Studying the Interplay between the Durations and Breakages of Continuous Integration Builds

Taher A. Ghaleb, Safwat Hassan, and Ying Zou

Abstract—The Continuous Integration (CI) practice allows developers to build software projects automatically and more frequently. However, CI builds may undergo long build durations or frequent build breakages, which we refer to as *build performance*. Both long durations and frequent breakages of CI builds can impede developers from engaging in other development activities. Prior research has conducted independent studies on build durations or build breakages. However, there is little attention to the possible interplay between reducing build durations and build breakages. In particular, it is unclear from prior studies (i) whether and how build performance is influenced by the context of projects; (ii) whether the actions to reduce build durations would reduce or increase build breakages; and (iii) whether fixing build breakages would lead to longer or faster builds. It is important for developers to understand the practices that make both timely and passing CI builds. In this paper, we conduct experimental and survey studies on the practices that can have dual or inverse associations with two *build performance* measures: *build durations* and *build breakages*. To this end, we extend an existing dataset called TravisTorrent to exclude inactive projects and collect recent builds of active projects. As a result, we study $924,616$ CI builds from 588 GitHub projects that are linked with Travis CI. In addition, we survey developers who contributed to the projects in our dataset to get their feedback on our experimental observations. First, we investigate project-level metrics and find that project characteristics have a significant association with build durations and breakages. In addition, we investigate how build-level metrics are associated with both build durations and breakages and observe an evident interplay between them. In particular, we observe that actions to fix build breakages (e.g., retrying or waiting for build commands) not only increase build durations but also do not guarantee passing builds. We also find that improving the build performance of a project is dependent on the current build durations and breakages of that project. Furthermore, we analyze how build performance changes over time and observe nearly a third of projects in which one performance measure is sacrificed in favor of the other, especially when not possible to achieve both together. The majority of our experimental observations are confirmed by survey results, which provide useful insights though some survey responses disagree with some of our experimental observations. Our work (a) provides developers with development and building practices to maintain timely and passing CI builds, and (b) encourages researchers to highlight any potential dual or inverse side effects when reporting actionable findings about CI builds.

Index Terms—Continuous Integration (CI); Build performance; Build duration; Build breakage; Empirical software engineering; Mining software repositories; Questionnaire survey

1 Introduction

Continuous Integration (CI) is a development practice that allows software developers to run software builds automatically and more frequently. Software developers adopt CI to obtain early feedback on the changes submitted to the code base [1]. With CI builds, developers are more concerned about *build durations* and *build breakages*, which we refer to as *build performance*. The *build duration* is the time a build takes to run. The *build breakage* is an indication that a build has not completed successfully (e.g., due to errors or failures).

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 [2]. Moreover, CI resources can be excessively consumed if builds take longer or are re-triggered more than once after fixing build breakages [3]. Therefore, it is important to understand the best practices that enable developers to maintain both timely and successful builds.

Prior research has conducted independent analyses on either build durations or build breakages. Previous studies have investigated the impact of long build durations on software development [4–6] and proposed recommendations to reduce the build duration [5, 7–9]. Previous studies have also proposed approaches to model and predict build breakages [10–13]. Recent studies [14, 15] have performed independent analyses to study the evolution and frequency of build durations and build breakages in software projects.

Despite the valuable insights provided 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 and how build performance is influenced by the context of projects; (ii) whether the actions to reduce build durations cause more or fewer build breakages; and (iii) whether fixing build breakages leads to longer or faster builds. Developers tend to fix build breakages regardless of the possible negative impact on build durations (e.g., by increasing the build time limit to avoid timeout-related build breakages[1]). Therefore,

- *T. A. Ghaleb is with the School of EECS, University of Ottawa, Ottawa, Canada. E-mail: tghaleb@uottawa.ca*
- *S. Hassan is with the Faculty of Information, University of Toronto, Canada. E-mail: safwat.hassan@utoronto.ca*
- *Y. Zou is with the Department of Electrical and Computer Engineering, Queen's University, Canada. E-mail: ying.zou@queensu.ca*

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

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).

In this paper, we conduct both experimental and survey studies on the practices that can have dual or inverse associations with two *build performance* measures: *build durations* and *build breakages*. First, we extend an existing dataset called TRAVISTORRENT [16], in which the last build was dated on August 31, 2016. To do this, we exclude inactive projects and collect recent builds of active projects until July 2020, which resulted in $924,616$ CI builds from 588 GITHUB projects linked with TRAVIS CI. In addition, we survey 139 developers who contributed to the projects in our dataset to get their feedback on our experimental observations. To model the interplay between the two build performance measures, we use build durations and build breakage ratios as dimensions to group the studied projects into four quadrants (or states): *dominantly timely/passing*, *dominantly long/passing*, *dominantly timely/broken*, and *dominantly long/broken*. Then, we perform project-level and build-level analyses of each quadrant to explore factors associated with build durations and breakages. For each experimental observation, we seek justifications from survey respondents (i.e., software developers with CI experience) about whether and why they agree or disagree with the observation.

Research Questions (RQ_s). We study the interplay between build durations and build breakages by addressing the following exploratory research questions:

RQ_1: What project characteristics are associated with build performance? Previous studies report inconsistent findings on CI builds [17, 18]. 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 project context (e.g., being a large and active project with extensive tests and configuration) has a strong association with build performance, which is confirmed by about half of the survey respondents. As suggested by survey respondents, developers are encouraged to change build configuration only when needed (e.g., to optimize or fix problems related to build performance). Survey responses suggest to "*Never change a winning horse*" and "*If it ain't broke, don't fix it*". Researchers on CI should also pay careful attention to the differences between projects when studying build durations and build breakages.

RQ_2: What build-level metrics are significantly 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, we model each quadrant with respect to the other quadrants using 40 build-level metrics and verify the results with those obtained from the developer survey. We find that both experimental and survey results suggest that workarounds (e.g., command retrials) should be avoided and developers should rather fix actual causes of build issues. However, there is some level of discrepancy, especially for factors having non-obvious associations with build performance (e.g., multi-architecture). Moreover, inconsistent with our experimental results, survey respondents do not see that less experienced developers break fewer builds unless the changes made are not much impactful.

RQ_3: What actions should developers take 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 factors (e.g., build configurations and server workload) are more associated with build performance than development factors (e.g., change complexity). However, one size does not fit all (i.e., a CI feature might perform inconsistently across projects). This means that actions to improve the build performance of a project is dependent on where that project stands in terms of build duration and frequency of build breakages. While survey respondents disagree with some experimental observations (e.g., caching rather introduces more breakages), they confirm that there is a possible trade-off between optimizing build durations and fixing build breakages, thus alarming developers to be more careful when dealing with different alternatives of build configurations.

RQ_4: How frequently does build performance change over time? Experiencing ups and downs in the build performance of a project at a certain point in time can indicate a change to the development process. In this RQ, we identify nine patterns in which projects may undergo changes to build performance. We find that projects may encounter ups and downs in the build performance over time, but as survey respondents confirm, changes in build performance may not happen for every code change. Nearly a third of projects sacrifice one performance measure in favor of the other, especially when not possible to achieve both together. Changes in build performance are mainly attributed to changes in development practices, such as involving contributions from developers with external expertise. Developers should constantly explore ways to improve CI builds that are commonly adopted by other projects.

In summary, our experimental results show an evident interplay between build durations and breakages, with dual or inverse side effects. The majority of our experimental observations are confirmed by survey results, which provide useful insights though some survey responses disagree with some of our experimental observations. In particular, improving CI builds is contingent on the context of a project and where it stands in terms of build performance. We highlight our key findings in relation to the interplay between build durations and breakages and discuss their implications for developers, researchers, and CI service providers. Feedback from survey respondents indicates that, though build performance could be improved using workarounds, developers should instead focus on addressing the root causes of CI build problems.

Overall, this paper makes the following contributions:

- We extend a publicly available dataset of CI builds, called TRAVISTORRENT (last build dated on *August 31, 2016*), to include CI builds till *July 17, 2020*. We make the updated dataset publicly available [19].
- We conduct a qualitative study using a user survey of 139 developers to help verify our quantitative findings on the interplay between the durations and breakages of CI builds. The survey results help us justify our findings and understand their practical implications.

- We characterize the project-level build performance by modeling the interplay between build durations and breakages. Project-level characteristics help developers understand the best CI practices that suit their projects.
- We highlight the factors associated with build durations and breakages with side effects. Our developer survey further verifies the experimental observations and provides insights into their impact in practice.
- We identify the patterns of build performance changes over time, and the factors associated with such changes. Survey respondents agree that ups and downs in build performance are common in practice.

Paper organization. The rest of this paper is organized as follows. Section 2 provides background about CI. Section 3 describes the experimental setup of our study. Section 4 presents the results and findings of our studied RQs. Section 5 discusses the implications of our findings. Section 6 presents threats to the validity of our results. Section 7 reviews the related literature on CI builds. Finally, Section 8 concludes the paper and suggests possible future work.

2 Background

Continuous Integration (CI) allows developers to generate software builds automatically and more frequently and get early feedback on their code changes [1]. CI builds are triggered when commits are pushed to a remote repository or via pull requests. CI build process starts by fetching the source code from the remote repository to the CI server, installing the dependencies, building the production code, and then running unit/integration tests. Builds pass if all phases run successfully or fail if errors are encountered. TRAVIS CI[2] and CIRCLECI[3] are examples of GITHUB-compatible CI services.

TRAVIS CI is a cloud-based CI service that is widely adopted by GITHUB projects [20]. TRAVIS CI maintains a customizable build lifecycle consisting of two main build *phases* (i.e., `install` and `script`) and an optional `deploy` phase. In the `install` phase, the remote repository is cloned and all 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. TRAVIS CI allows developers to customize the building process through a configuration file (i.e., `.travis.yml`). For example, developers can configure the CI building machine (e.g., specifying the operating system), tune parameters (e.g., timeout threshold), and enable CI features (e.g., CI caching).

Builds in TRAVIS CI can run multiple independent *jobs* in parallel or in sequence. A CI build (and each build job) may have one of the following status indicators: *started*, *passed*, *errored*, *failed*, or *canceled*. A CI build (or a build job) is *passed* if all the build phases are successful. CI builds may break if any of the build phases or build jobs is *errored* or *failed*. The *errored* status indicates a problem with the `install`, whereas the *failed* status indicates a problem with the `script` phase. The *canceled* status indicates that the build is manually interrupted. We refer to builds that have an *errored* or *failed* status as *broken* builds. A breakage in one build job breaks the entire build. However, marking a build job as *allow_failures* leaves the CI build status unaffected when that job is broken. A build is normally marked as finished if all its jobs are completed, but it can be configured with *fast_finish* to finish as soon as a job has already failed or only remaining jobs are allowed to fail.[4]

3 Experimental Setup

This section presents the setup of our empirical study. We explain how we collect and process the data for our studied RQs and how we perform a survey study to support experimental observations.

3.1 Data Collection

Figure 1 shows an overview of our study. Our study is based on data collected from TRAVISTORRENT [16], a commonly used dataset to study CI builds [11, 21, 22]. TRAVISTORRENT contains builds from $1,283$ projects: 886 Ruby, 393 Java, and 4 JavaScript projects. We exclude the 4 JavaScript projects, since they are not 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 no longer available in GITHUB, (b) stopped using TRAVIS CI, or became less active (i.e., having less than 50 builds [16] 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 TRAVIS CI API[5] and collect the corresponding build metrics from GHTORRENT.[6] We exclude *started* and *canceled* builds, since they are incomplete. In total, our updated dataset contains $924,616$ builds. These builds span across various project branches, with main branches having the majority (72%) of all builds in our dataset.

For each build, we compute the actual build duration by taking the difference between the build start and finish time [9]. Prior CI research considers both *errored* and *failed* builds as *broken* [10, 13, 21, 23]. Hence, we consider all build breakages in the dataset equally regardless of being *errored* or *failed*. Moreover, our study assumes all build breakages to be deterrents, since they make developers wait for whoever broke the build to fix it, thus delaying software releasing. Therefore, we disregard the fact that some breakages might be desirable (e.g., when experimenting with new features or bug fixing on separate branches) or unnecessary (e.g., intermittent breakages due to flaky tests). Identifying these kinds of breakages can be subjective and challenging [13, 24], which is out of the scope of this study. Besides, we clone the GIT repository of each project to compute additional project-level and build-level metrics (e.g., build configuration and developer experience). We also analyze build configuration files (i.e., *.travis.yml*) to compute metrics about build configurations, such as adopted CI features. Moreover, we collect CI metrics about builds (e.g., build integration environments) from TRAVIS CI.

2. https://travis-ci.org
3. https://circleci.com
4. https://blog.travis-ci.com/2013-11-27-fast-finishing-builds
5. https://docs.travis-ci.com/api
6. https://ghtorrent.org

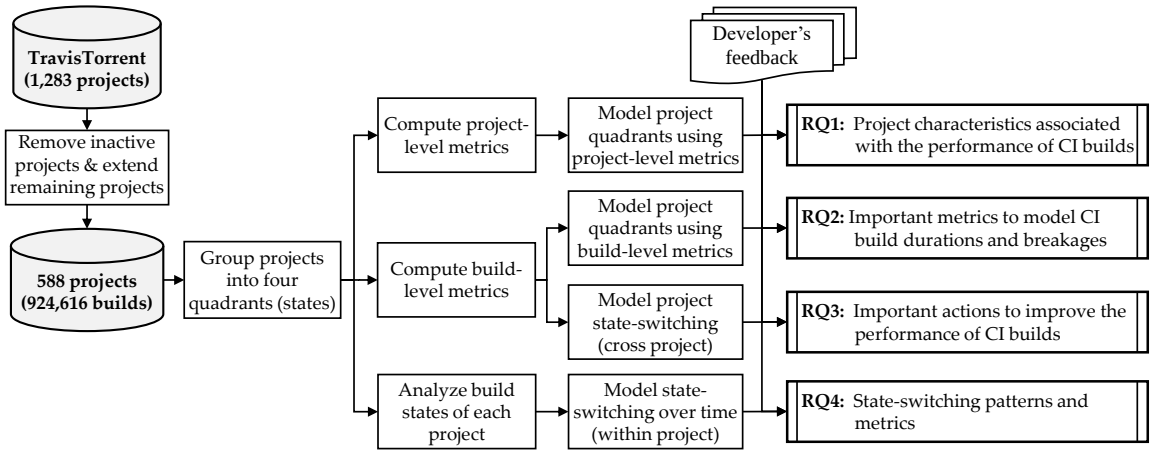


Figure 1: Overview of our study

3.2 Data Processing

This section describes how we process our collected data.

3.2.1 *Grouping the studied projects into quadrants*

Each build in our dataset has two performance measures representing (i) *build duration* (continuous values) and (ii) *build status* (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 *build breakage ratio* as a proportion of broken builds of a project over 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 using the overall median value of *build breakage ratios* and *median build durations* of all projects as thresholds. The median measure is robust as it is not heavily influenced by outliers [25]. Figure 2 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 long/passing '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.

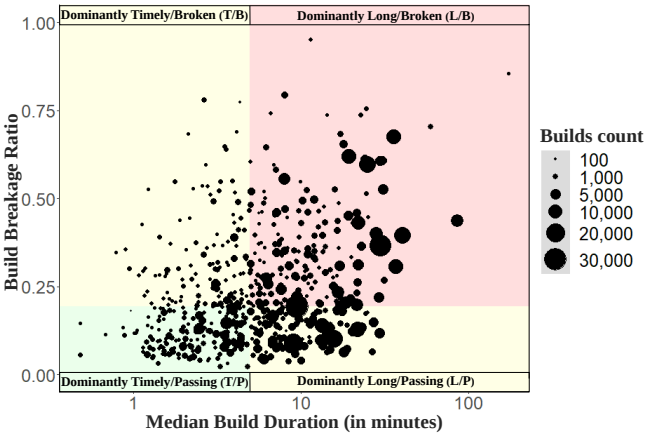


Figure 2: Quadrants of the studied projects using the median build duration (x-axis) and the breakage ratio (y-axis)

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 projects, we also produce quadrants for each quarter of the project lifetimes. 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.

3.2.2 *Computing project-level metrics*

Build performance can be affected by the characteristics of software projects [18]. 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 experience different CI build performance. We study whether the performance of Ruby builds differs from that of Java builds.
- *Project Maturity:* Mature projects can have a different CI experience from those projects that are still in 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 build performance.
- *Project Reputation:* Projects desire to generate acceptable CI builds as much as maintain a better reputation. We study whether the reputation of projects (e.g., project stars) is associated with the build performance.

For each facet, 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 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 (RQ_1).

3.2.3 Computing 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 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 (RQ_2) and how to switch projects to other states (RQ_3). 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. We also use build-level metrics to perform within-project analysis to explore significantly changed metrics across the lifetime of the project (RQ_4).

3.2.4 Performing correlation and redundancy analyses

We use the computed project-level and build-level metrics as independent variables to fit our logistic regression models (See Section 4: *RQ_1*, *RQ_2*, and *RQ_3*). We first exclude the highly correlated independent variables, since they can adversely affect regression models [26]. We follow the guidelines provided by Harrell [27] on regression modeling. In particular, we use the Spearman rank ρ clustering analysis [28] (using the `varclus` function from the `rms`[7] *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 [29].

For project-level metrics, we exclude '*# of builds*', since it is highly correlated with '*# of commits per lifetime*'. We also exclude '*# of build jobs*', since it is highly correlated with '*# unique build environments*'. Moreover, we exclude '*# of forks*', since it is highly correlated with '*# of stars*'. For build-level metrics, we exclude (a) '*Configuration lines added*' and '*Configuration lines deleted*', since they are highly correlated with '*Configuration files changed*', (b) '*Source files changed*', since it is highly correlated with '*Source churn*', (c) '*Jobs removed*', since it is highly correlated with '*Jobs added*', (d) and finally '*Author experience: # of commits*', since it is highly correlated with '*Author experience: # of days*'.

Finally, we perform a redundancy analysis on the remaining variables, since redundant variables can distort regression models [27]. We use the `redun` function from the `rms` *R* package to identify 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.

3.3 Conducting a developer survey

Alongside our experimental observations, we further seek feedback and insights from software developers with CI experience on the possible interplay between build durations and build breakages. In particular, we conduct a user study with sample developers who have contributed to the projects in our study to confirm/validate our experimental observations. To do this, we use the GitHub API[8] to retrieve the names and email addresses of developers having commits in the studied projects. Then, we sort developers based on their latest activity on GitHub and send survey invitations to the most recently active developers (a total of 4,366 developers) who have public Email addresses (the invitation letter can be found in our replication package [19]). These developers contributed to the vast majority (94%) of the studied projects, taking into consideration that one developer can have contributions across multiple projects. Our questionnaire survey consists of a combination of Likert scale and open-ended questions, presented in Table 3. The survey is delivered online through Qualtrics and has two groups of questions: (1) questions to gain insights about the demographics of survey respondents; and (2) questions to verify the experimental observations of our study. To compensate developers for their time, we offer 10$ Amazon gift cards to 30% randomly drawn survey respondents who complete the questionnaire and share their contact information to receive the prize. Our invitation Email did not reach 366 (8.4%) developers due to having obsolete Emails. In total, we receive 224 anonymous responses to our survey (i.e., 5.6% response rate after excluding the unreachable Emails). 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).

Figure 3 shows a summary of the demographics of our survey respondents. The majority of the respondents are males (96%), between 25 and 45-years-old (80%), from Europe and North America (87%), are maintainers or active contributors of software projects (64%), have over ten years of software development experience (81%), have over five years of Travis CI experience (54%), and use other CI services, such as CircleCI and Jenkins (42%). We exclude two survey respondents who have no experience with Travis CI or other CI services. Compared to the *State of the Octoverse* report (2022) as a baseline,[9] which reports the demographics of GitHub users, we find that the top country of our survey respondents is the United States,

7. https://cran.r-project.org/web/packages/rms/rms.pdf

8. https://docs.github.com/en/rest/commits/commits

9. https://octoverse.github.com/2022/global-tech-talent

Table 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
	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

Table 2: Description of the build-level metrics used in our logistic regression models

	Build-level metric	DT*	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 work 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 caching is enabled or used in the build
	Is `fast_finish` enabled	C	Whether `fast_finish` is enabled in the build
	Is `docker` used	C	Whether a `docker` container is used to run build jobs on
	Is `sudo` enabled	C	Whether `sudo` is enabled in the build
	Operating system (OS)	C	The operating system(s) used to run the build jobs on
	OS distribution	C	The distribution of the operating systems used to run the build jobs on
	Jobs added	N	Number of jobs added to the build
	Jobs removed	N	Number of jobs 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) of 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	(intra-project) Whether the developer who triggered the build is a core member of the project to which the build belongs
	Developer experience: # of days	N	(intra-project) Number of days the developer who authored the build has been contributing to the project to which the build belongs
	Developer experience: # of commits	N	(intra-project) Number of commits the developer who authored the build has in the project to which the build belongs
	Developer total activities	N	(inter-project) Number of developer activities (commits, issues, pull requests, and reviews) across GitHub projects in the past three months
	Developer % of commits	N	(inter-project) Ratio of the commits of the developer to the total activities
	Developer % of issues	N	(inter-project) Ratio of the issues raised by the developer to the total activities
	Developer % of open pull requests	N	(inter-project) Ratio of open pull requests of the developer to the total activities
	Developer % of merged pull requests	N	(inter-project) Ratio of merged pull requests of the developer to the total activities
	Developer % of unmerged pull requests	N	(inter-project) Ratio of unmerged pull requests of the developer to the total activities
	Developer % of reviews	N	(inter-project) Ratio of reviews performed by the developer to the total activities

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

thus matching what is reported in the report. Therefore, all conclusions of this study are only representative of the study population.

We integrate the results of our developer survey with the experimental observations of the studied RQs. For each observation, we ask participants two questions: a Likert-scale question and an open-ended question. Likert scale questions ask about the extent to which developers agree or disagree with an observation from score 1 (strongly disagree) to score 5 (strongly agree). Open-ended questions ask developers about any justification of the experimental observations. To analyze the level of agreement of developers with each observation, we calculate the percentage of each score for each question. To understand the rationale behind the scores, we manually analyze the responses to open-ended questions related to each score and report the common justifications and provide quotations of representative responses. For open-ended questions that require categorization of the responses (e.g., Section 2), card sorting sessions [30] are used to perform an open coding of the responses to extract common themes and statements. Card sorting is performed by the first and second co-authors, where they collaboratively perform manual labeling of responses, discuss any uncertainty in the responses, and merge similar labels into common groups.

4 Experimental Results

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

4.1 RQ_1: What project characteristics are associated with build performance?

Motivation. Ståhl and Bosch [18] suggested that the variations in the observations reported by prior studies on build durations and breakages could be due to contextual differences in the studied projects. However, the role of context in the overall build performance remains unexplored. 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 2). In particular, we fit a multinomial logistic regression model [31] using the `multinom` function provided by the `nnet`[10] 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 and comparing 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. This helps us highlight the most significant project characteristics associated with build performance. Then, we use the ANOVA test [32] 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 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 [33] (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.

10. https://cran.r-project.org/web/packages/nnet/nnet.pdf

Findings. Figure 4 shows the results of grouping projects into quadrants for each quarter of their development lifetime. Table 4 shows the results of the multinomial regression model we fit on the four quadrants of projects depicted in Figure 2, taking into consideration that the *Timely/Passed* quadrant is the reference for comparing other quadrants.

Observation 1.1. ***The majority (80%) of projects retain their build performance for over three quarters of their lifetime.*** Looking at Figure 4, we find that the overall median build duration has steadily increased over time from 3.4 minutes (1^{st} quarter) to 5.8 minutes (4^{th} quarter), with an average difference of $+0.8$ minutes per quarter. The increase in build durations is expected as software projects become more complex as they evolve. However, we find that the build breakage ratio has decreased over time from 20% (1^{st} quarter) to 17% (4^{th} quarter), with an average difference of minus 1% per quarter. In addition, we observe that 28% of projects retain the same quadrant for all lifetime quarters and 53% of 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 their entire lifetime (as shown in Figure 2), particularly the results of the multinomial regression model in Table 4.

Observation 1.2. ***Overall, about half of survey respondents acknowledge the association of project characteristics with build performance.*** As Figure 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. Looking at Figure 6, we observe that the common project characteristics associated with build performance indicated by respondents include tests, size, configuration, dependencies, age, and team. These characteristics mainly relate to project maturity, which are already accounted for in our model. Most importantly, survey respondents raised many testing-related characteristics that can have a direct association with build performance, including the architecture, complexity, size, level (e.g., unit or integration), and execution frequency of software tests. However, we observe no responses referring to CI activity (e.g., frequency of CI building or configurations) and only a few (five) responses related to programming languages. This suggests that developers might be less aware of CI-specific optimization opportunities for their builds.

Table 3: Survey instrument (questionnaire)

#	Question	Answer
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 TRAVIS CI?	[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 TRAVIS CI?	[User, Maintainer, Active contributor, Occasional contributor, Other]
$Q_{1.7}$	What other CI services have you used in addition to TRAVIS CI? (Select all that apply)	[CircleCI, Appveyor, TeamCity, Other]

Section 2: Opinion about our experimental observations
In this section, we wish to get your generous feedback on the observations that we obtain from conducting statistical analyses using an empirical experiment on 500+ GitHub projects that are linked with TRAVIS CI.

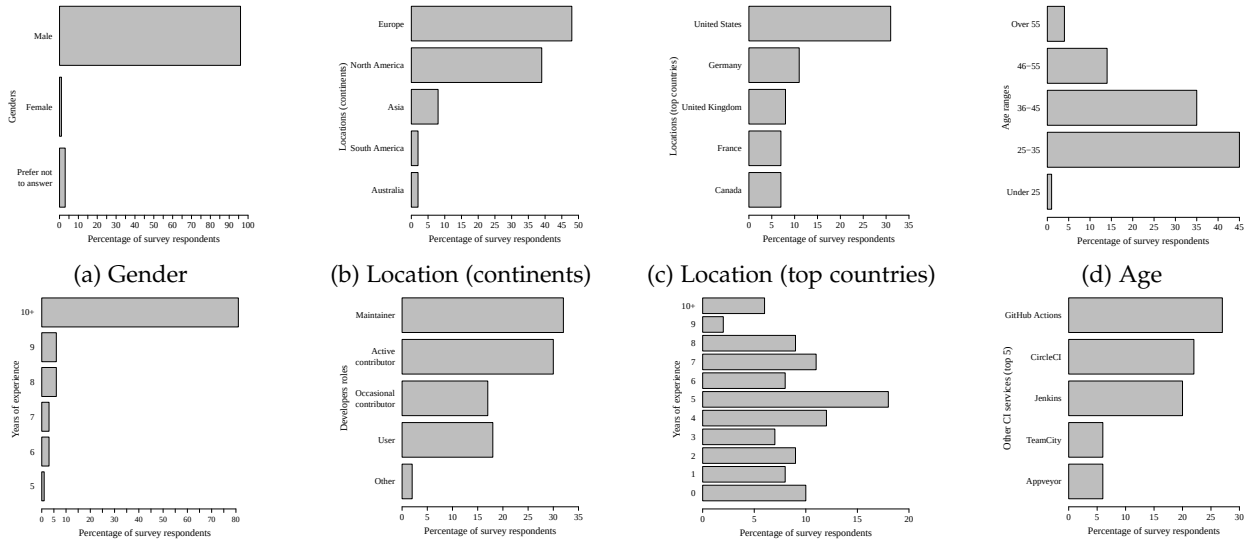
Below, you will find observations from our experimental analysis. Please let us know the extent to which you *Agree* or *Disagree* with each finding and how you would justify your answer or explain the possible reasons for each observation, based on your experience.

#	Observation	Likert Scale [1: strongly disagree, 2: disagree, 3: neutral, 4: agree, 5: strongly agree]	Justification/Explanation open-ended response
$Q_{2.1}$	Project characteristics (e.g., project size, build complexity, activity, etc.) are associated with build performance. If you agree, explain what project characteristics are associated the most.	1 ○ 2 ○ 3 ○ 4 ○ 5 ○	
$Q_{2.2}$	Modifying the build configuration more frequently can be associated with build durations and breakages.	1 ○ 2 ○ 3 ○ 4 ○ 5 ○	
$Q_{2.3}$	Younger projects (i.e., those that are at their early development stages) are likely to have more build breakages but faster builds.	1 ○ 2 ○ 3 ○ 4 ○ 5 ○	
$Q_{2.4}$	Commits submitted by occasional and less experienced developers are associated with making builds faster and passing.	1 ○ 2 ○ 3 ○ 4 ○ 5 ○	
$Q_{2.5}$	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 ○	
$Q_{2.6}$	Using non-default operating systems or Linux distributions is associated with long and frequently broken builds.	1 ○ 2 ○ 3 ○ 4 ○ 5 ○	
$Q_{2.7}$	Actions to optimize builds are restricted by the current build durations and breakages of a project.	1 ○ 2 ○ 3 ○ 4 ○ 5 ○	
$Q_{2.8}$	The *fast_finish* configuration speeds up feedback of broken builds, but not passing builds.	1 ○ 2 ○ 3 ○ 4 ○ 5 ○	
$Q_{2.9}$	Besides making builds faster, CI caching can also be associated with a reduced number of build breakages.	1 ○ 2 ○ 3 ○ 4 ○ 5 ○	
$Q_{2.10}$	CI builds triggered on weekends are more likely to be faster and passed.	1 ○ 2 ○ 3 ○ 4 ○ 5 ○	
$Q_{2.11}$	Build durations and build breakages tend to follow a similar pattern from one build to another (i.e., if a previous build is long or broken, the following build will most likely be long or broken).	1 ○ 2 ○ 3 ○ 4 ○ 5 ○	
$Q_{2.12}$	Build performance (durations and breakages) change over time throughout the project lifetime.	1 ○ 2 ○ 3 ○ 4 ○ 5 ○	

Table 4: Project-level stepwise multinomial model results — detailed results can be found in our replication package [19]

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



(a) Gender (b) Location (continents) (c) Location (top countries) (d) Age

(e) Development experience (f) Role (g) Travis CI experience (h) Other CI services

Figure 3: Demographics of survey respondents

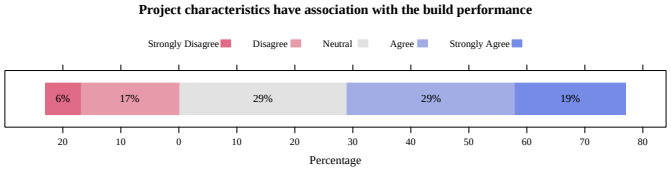


Figure 5: Developers' feedback about the association of project characteristics with build performance (Q2.1)

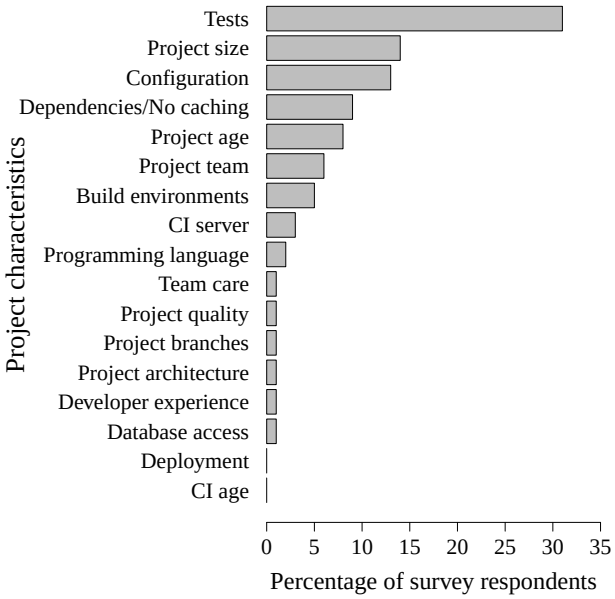
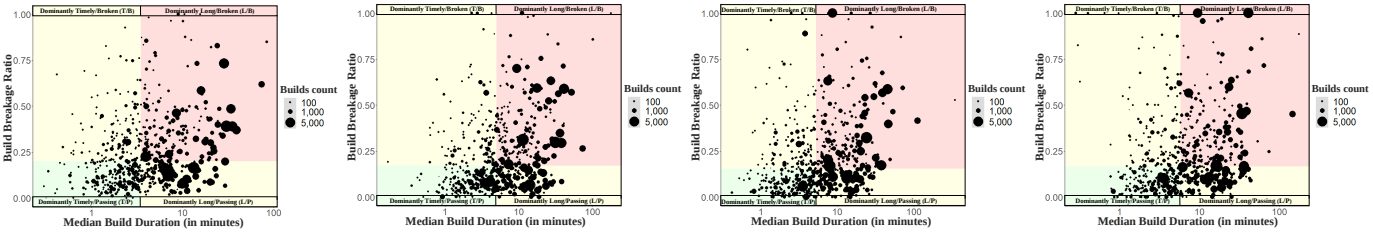


Figure 6: Common project characteristics associated with build performance based on survey respondents (Q2.1)

Observation 1.3. ***Overly configuring CI builds is highly associated with longer build durations and frequent build breakages.*** Our model results show that the build configuration frequency has a significant association with long and frequently broken builds. Analyzing build configuration frequencies of the projects in our dataset reveals that projects in which builds pass timely tend to be configured 36% less frequently than other projects. As per Figure 7, 47% of survey respondents agree with our observation, but there are two points of view regarding such an association. On the one hand, 26% of responses indicate that changing the build configuration is likely 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 that changing 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*". Overall, survey respondents recommend changing the build configuration only on demand (i.e., when CI problems are encountered because of the build configuration itself); "*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*".



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

Figure 4: Grouping projects into quadrants over the development lifetimes (quarter-wise)

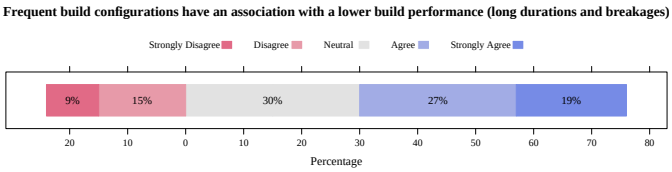


Figure 7: Developers' feedback about the association of frequent build configurations with build performance (Q2.2)

Observation 1.4. ***The more build environments a project has, the more likely for the project to suffer from long and broken builds.*** We observe that projects with long and broken builds have significantly more (a median of $1.7x$) integration environments than timely and passed builds. Multiple CI environments can complicate build maintenance and make builds more prone to long durations and intermittent breakages [9, 13]. Previous research reported that most of the changes to the CI build configuration are related to build environments [3, 34]; "*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 environment for that job can negatively affect the build performance. Survey respondents indicate that the main issue regarding the declined build performance of multi-job builds is mainly due to bad parallelization practices employed by developers; e.g., "*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 run on a feature branch (which was merged in fast-forward-mode)*".

Observation 1.5. ***Large and active projects are likely to have longer and more frequent build breakages.*** Vasilescu et al. [35] reported that, in older projects, pushed commits lead to more build breakages than pull requests. However, our model reveals no association of build performance with the project's chronological age (i.e., in terms of time). We observe that, regardless of the chronological age of projects, having 100 more commits per year (i.e., more active) increases the odds of having long and broken builds by 34%. This result is confirmed by survey respondents who indicate that "*The age of the project has no bearing on this whatsoever.*"; "*I'm not sure if I've seen a correlation here*". Still, we observe from Figure 8 that half of the respondents agree that younger projects can have better build performance. However, our analysis of the justifications shows that respondents tend to rather refer to the size of projects when referring to age "*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.*". Therefore, given that frequent committing is recommended in CI, developers should do so very carefully. For example, committing changes without taking into account the side effects of such changes on existing tests or configurations may introduce unnecessary overhead that would not occur if changes were committed with more care. Therefore, developers are encouraged to keep in mind any potentially accumulated increase in build durations or breakages when making changes to the code base.

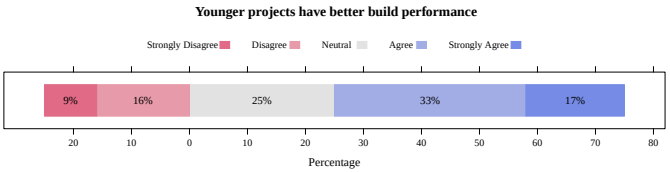


Figure 8: Developers' feedback about the association of project age with build performance (Q2.3)

RQ_1 summary. As confirmed by survey respondents, the context of a project (e.g., being a large and active project with extensive tests and configuration) has a strong association with frequently long and broken builds. Developers are encouraged to change build configuration only when needed (e.g., to optimize or fix problems related to build performance).

4.2 RQ_2: What build-level metrics are significantly associated with build durations and/or breakages?

Motivation. Findings of RQ_1 suggest that project characteristics are strongly associated with build performance. However, builds within the same project may have different characteristics. In this RQ, we perform an intra-project analysis aiming to understand which build-level metrics have a strong association with build durations and breakages.

Approach. Similar to RQ_1, we fit a multinomial logistic regression model using quadrants as 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.

Findings. Table 5 presents the results obtained from the models of each of the four quadrants.

Observation 2.1. ***The experience and role of developers have significant associations with build performance.*** A prior study [10] 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 9, 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*

Table 5: Results of the build-level stepwise multinomial model — results of all variables are in our replication package [19]

Build-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 activities	***	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

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 activities (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*".

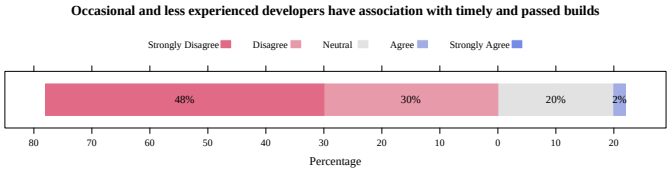


Figure 9: Developers' feedback about the association of the experience of developers with build performance (Q2.4)

Observation 2.2. ***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.[11,12] However, such configurations are likely to delay builds [9]. Besides delaying builds, we observe that command retrials do not guarantee passing builds. Our results show that retrying and waiting for commands or tests are associated with both long and 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 a sample of the logs of the builds that use `travis_wait` shows that breakages occur after waiting for commands 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,[13] we find that commands may create child processes that are not tracked by `travis_wait`, thus making the build timeout when the main process becomes silent for ten minutes.

Almost half of survey respondents agree that waiting for or retrying commands is not always helpful in fixing build breakages (Figure 10) and should rather be a last resort; For example, some 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*". Besides, command retrial "*is a strategy to address something that's unreliable, so it makes sense that it would often not be successful*". Nevertheless, other respondents recommend using such configurations only when build 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 arguments of respondents who disagree with our observation shows that they 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 such workarounds for all commands, and should rather consider providing real fixes to failing commands.

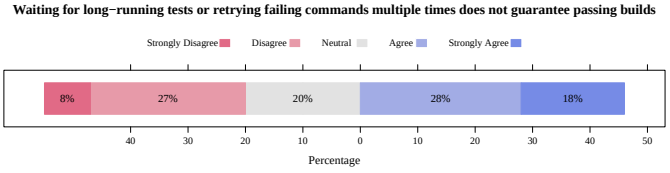


Figure 10: Developers' feedback about the association of retrials and waiting for commands/tests with build performance (Q2.5)

Observation 2.3. ***Using multiple or non-default build architectures (i.e., operating system) has a negative associa-***

11. https://docs.travis-ci.com/user/common-build-problems/#build-times-out-because-no-output-was-received
12. https://docs.travis-ci.com/user/common-build-problems/#timeouts-installing-dependencies
13. https://github.com/travis-ci/travis-ci/issues/8526

tion with build performance. Developers change the build architecture very often [34]. However, failure to choose a proper architecture may impact the overall CI building experience [36]. 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[14] of TRAVIS CI indicates that running builds on newer versions of `OSX` is likely to cause 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 11). Yet, 47% of respondents have a disagreement with this association. While our analysis of the responses shows that respondents do not find this association to be obvious, we observe that they had no prior experience using multiple architectures in their CI builds. Still, based on the analyzed responses, some operating systems are 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.

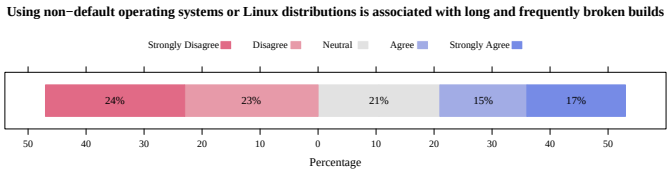


Figure 11: Developers' feedback about the association of build architecture with build performance (Q2.6)

RQ_2 summary. Both experimental and survey results suggest that workarounds (e.g., command retrials) should be avoided and developers should rather fix actual causes of build issues. However, there is some level of discrepancy, especially for factors having non-obvious associations with build performance (e.g., multi-architecture). Moreover, inconsistent with our experimental results, survey respondents do not see that less experienced developers break fewer builds unless the changes made are not much impactful.

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

4.3 RQ_3: What actions should developers take to optimize build performance?

Motivation. RQ_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 an undesirable quadrant (e.g., long/broken builds) to a better quadrant (e.g., timely/passed). While 57% of survey respondents acknowledge that build breakages can sometimes be unnecessary (e.g., intermittent failures due to flaky tests), 88% of respondents confirm that they tend to fix build breakages regardless of the impact on build duration. Therefore, In this *RQ*, we study the most important actions that can have dual and/or side effects on the build performance.

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 2. RQ_1 suggests that project characteristics are strongly associated with build performance. Hence, we use Generalized Linear Mixed Models (GLMM) for logistic regression [37] 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 [38]. We fit six mixed-effects 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_1 and RQ_2, we use the ANOVA and odds ratios to analyze the relationship of the dependent variable with the independent variables of each model.

Table 6: Results obtained from the six switching mixed-effects logistic models

Metric	L/B ⟼ T/P			L/B ⟼ L/P			L/B ⟼ T/B			L/P ⟼ T/P			L/P ⟼ T/B			T/B ⟼ 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

Findings. Table 6 presents the results obtained from the six models to switch projects to better build states. We summarize the most important actions that are likely to switch projects to better quadrants in Figure 12.

Observation 3.1. ***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 to switch dominantly long/broken projects to be dominantly timely/passed (e.g., *committing on weekends* or *fewer command retrials*) do not apply when the project is dominantly timely/broken. Similarly, the actions for switching a project to a dominantly timely/passed quadrant from a dominantly timely/broken quadrant (e.g., `fast_finish`) do not apply when the project is dominantly long/passing. While 43% of survey respondents agree with this conclusion, as Figure 13 depicts, analyzing the responses of respondents reveals that the restriction in actions is likely 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 assess the performance of their builds during the past periods (i.e., the last k builds) before proceeding with build performance improvement.

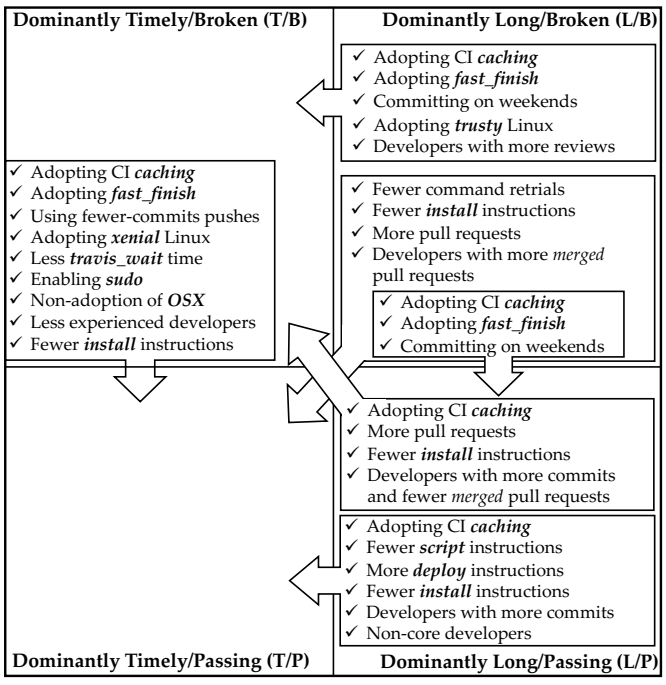


Figure 12: Summary of the quadrant-switching actions

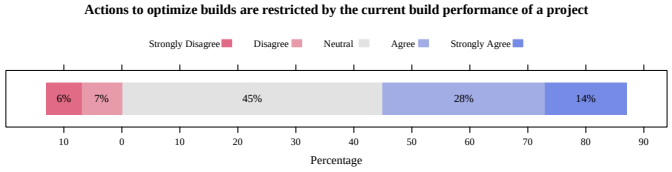


Figure 13: Developers' feedback about the restricted actions to improve build performance (Q2.7)

Observation 3.2. ***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 [9]. 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 develop-

ers 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 there is no need to wait for `allow_failures` jobs, since they do not affect the overall build status. As Figure 14 shows, 59% of survey respondents neither agree or disagree with our observation, while 27% of them agree with our observation. Yet, our analysis of the responses shows that the `allow_failures` configuration is not recommended as it does not allow revealing all possible build 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 have a clear rationale to either allow a build to fail or simply exclude failing build jobs.

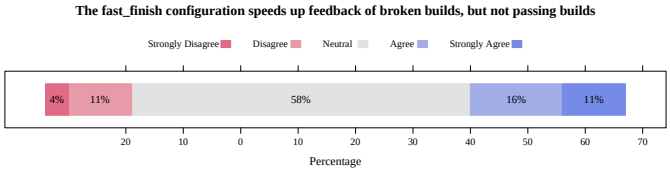


Figure 14: Developers' feedback about the association of `fast_finish` with only timely broken builds (Q2.8)

Observation 3.3. ***CI caching has an inverse association with build breakages in addition to build durations.*** Prior research has reported that caching less frequently changing content is highly associated with timely builds [9]. Our results reveal that, besides speeding up builds, the odds of having passed builds increases by 139% when caching is adopted. About 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 dependency installation [10]. We observe from Figure 15 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 a relatively higher percentage (43%) of disagreement. Disagreeing respondents indicate that such an association is not obvious. Our analysis of disagreeing responses shows arguments indicating 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 an eye on build breakages that might be caused due to outdated or corrupt dependencies in the cache.

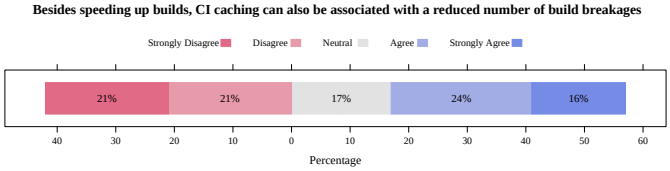


Figure 15: Developers' feedback about the association of `caching` with only timely broken builds (Q2.9)

Observation 3.4. ***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 [9] but not with build breakages [10]. 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.[15,16]. We find that the TRAVIS CI team is aware of the overhead caused due to triggering builds during rush hours. Although 27% of respondents agree with our observation, other 30% of respondents have uncertain opinions. Overall, respondents commonly agree that, on weekends, CI resources are 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*". Yet, this might not always be the case as other respondents indicate that "*as a paid travis subscriber, we don't get throttled.*". Hence, developers are recommended to reduce the number of operations in their builds to ensure both timeliness and passing regardless of triggering time.

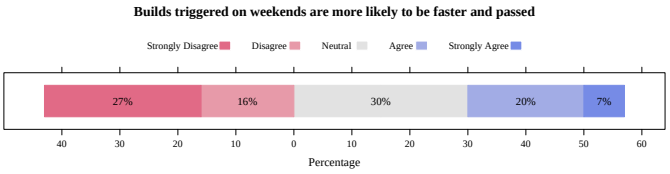


Figure 16: Developers' feedback about the association of `day of week` with build performance (Q2.10)

RQ_3 summary. One size does not fit all (i.e., a CI feature might perform inconsistently across projects). While survey respondents disagree with some experimental observations (e.g., caching rather introduces more breakages), they confirm that there is a possible trade-off between optimizing build durations and fixing build breakages, thus alarming developers to be more careful when dealing with different alternatives of build configurations.

4.4 RQ_4: How frequently does build performance change over time?

Motivation. Findings of RQ_3 suggest that development and build practices are associated with build performance. However, little is known about how build durations and breakages evolve and what makes build performance undergo ups and downs. It is important to understand what practices change when the build performance changes over time. In this RQ, we perform an intra-project analysis to investigate the patterns, frequency, and metrics of projects switching between different build performance states over time.

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

15. https://github.com/travis-ci/travis-ci/issues/2072
16. https://github.com/travis-ci/travis-ci/issues/8489

- We identify the state switching between every pair of subsequent builds.
- We convert the project lifetime into four quarters using the median, 1^{st}, and 3^{rd} quantiles. Then, we identify the most frequent state of each quarter as well as the state switching between the quarters of each project.

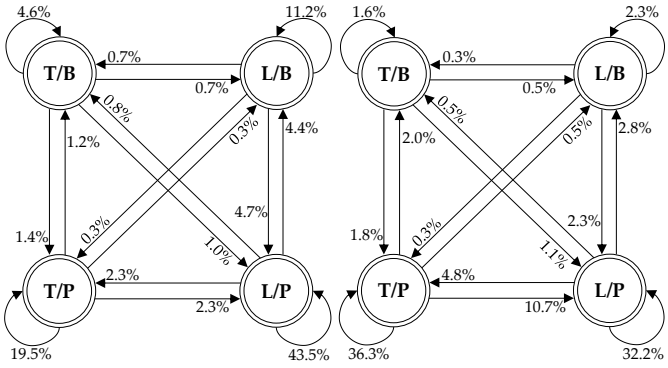
We use the rules presented in Table 7 to summarize project state switching as *positive*, *negative*, or *strategic*. Though build breakages can sometimes be helpful to inform developers about bugs, we consider in our analysis long build durations and frequent build breakages as undesirable performance states, thus switching to them is considered *negative*, whereas switching from them is considered *positive*. A *strategic switch* happens when developers decide to sacrifice one performance measure (e.g., timeliness) for another (e.g., passing). If two subsequent builds or quarters have the same states (e.g., both are positive), we combine their states into a single state. 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 nine evolutionary patterns (see Table 8) for projects to switch from one state to another state or remain in the same state throughout the development lifetime. In addition, we model the statistical change in metrics between every two state switches of each project using logistic regression.

Findings. Figure 17 shows (a) the ratios of build-to-build state switches and (b) the ratios of quarter-to-quarter state switches across the studied projects. Table 8 presents nine patterns in which projects undergo build state-switching over time (more details about the state-switching sub-patterns can be found in our replication package [19]).

Observation 4.1. ***The majority (i.e., 79%) of builds in our dataset follow the build state of former builds.*** Prior research has reported that build breakages follow former breakages [10, 13]. We observe that not only the breakage but also the duration of CI builds is likely to be similar to former builds. Nevertheless, 21% of build pairs have different states. Prior research has reported that build breakages follow former breakages [10, 13]. We observe that not only the breakage but also the duration of CI builds follow former builds. Nevertheless, there are 21% of build pairs that encounter different build performances. 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.e., 2.3%) in both directions. Looking at Figure 18, survey respondents mostly (68%) agree that builds tend to have a similar performance to former builds, in terms of durations and breakages, since

Table 7: The rules used to tag the state switching of projects

Switch	Symbol	Previous quadrant	Current quadrant
Positive	$\oplus$	Any	Timely/Passing
Negative	$\ominus$	Any Timely/Passing	Long/Broken Any
Strategic	$\odot$	Long/Passing Timely/Broken	Timely/Broken Long/Passing



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

Figure 17: Percentages of state-switching cases

Table 8: Patterns of quadrant state-switching over time

Pattern	Trend	Projects #	Projects %
$\oplus$	Steadily Positive	206	35%
$\odot$	Steadily Strategic	187	32%
$\oplus \mapsto \odot$	Degrading	111	19%
$\odot \mapsto \oplus$	Improving	35	06%
$\odot \mapsto \ominus$	Degrading	17	03%
$\ominus \mapsto \odot$	Improving	16	03%
$\ominus$	Steadily Negative	9	02%
$\oplus \mapsto \ominus$	Degrading	5	01%
$\ominus \mapsto \oplus$	Improving	2	00%

"*Not every change changes the build system*"; "*... unless there are non-deterministic tests*". Therefore, it is important for developers to understand how to revert CI builds from an undesired build state to a better state.

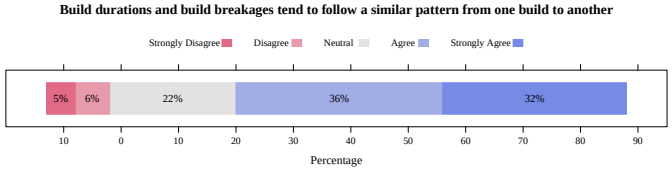


Figure 18: Developers' feedback about build performance over time (Q2.11)

Observation 4.2. ***Developers mostly maintain both timely and passing builds (35% of projects), but sacrifice one performance measure in favor of the other, especially when not possible to achieve both together (32% of projects).*** Looking at Table 8, we observe that about two-thirds of the studied projects maintain a steady build performance throughout the development lifetime. In particular, most (35%) of the projects maintain a positive build performance state, whereas build states of 32% of the projects have been steadily strategic over time (i.e., sacrificing build durations for passing, or vice versa). Of the 187 steadily strategic projects, 166 (89%) are steadily long/passing, whereas only 2% are steadily timely/broken, with the rest being switched from timely/broken to long/passing (6%) or vice versa (3%). Such results indicate that developers tend to be in favor of passing over timeliness if both cannot be achieved together. We find only five projects (2%) with steadily negative build performance during their lifetime. Lastly, we find it rare to jump from a negative to a positive build state directly (occurred in only two projects). This result emphasizes the importance of RQ_3 findings, which

highlight the actions that help switch projects to better build performance states. Nevertheless, build performance has undergone degradation and improvement in 23% and 9% of the projects, respectively. This result suggests that build performance can easily get worse, thus posing a challenge for developers to return it back to its normal state. The majority (56%) of survey respondents indicate that long and broken builds tend to improve over time (see Figure 19). Regardless, developers are encouraged to inspect their builds periodically, especially when taking longer or breaking more frequently than usual, rather than relying on other developers to do so.

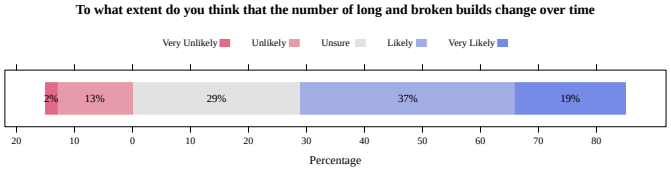


Figure 19: Developers' feedback about the likelihood of changed build durations and breakages over time (Q2.12)

Observation 4.3. ***Build state-switching over time is mostly attributed to significant changes in the experience of developers contributing to the projects.*** 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 these projects. Our statistical modeling of build state-switching shows that there are significant differences between the characteristics of builds before and after the switch (the full list of statistical testing results of build state-switching can be found in our replication package [19]). We observe that metrics related to developer experience (both intra- and inter-project) are the most common metrics with a significant direct association among state switches. In addition, we observe that the most significant metrics (e.g., the top 20 metrics) of intra-project state switches differ from the inter-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.

RQ_4 summary. Build performance tends to be steadily positive or strategic. Yet, as our experimental and survey results show, projects may still encounter ups and downs in build performance over time, but not for every build. Nearly a third of projects sacrifice one performance measure in favor of the other, especially when it is not possible to achieve both together. Changes in build performance are mainly attributed to changes in development practices, such as involving contributions from developers with external expertise. Therefore, developers are encouraged to constantly explore ways to improve CI builds that are commonly adopted by other projects.

4.5 Highlights of the study results

Our experimental and survey results show an evident interplay between build durations and breakages, with dual or inverse side effects. In particular, build performance is contingent on the context of each project, given that project characteristics (e.g., project maturity, development activity, and CI configuration) are key in shaping the duration and status of CI builds. Deciding what actions to take to reduce the build duration or fix build breakages also depends on where a project stands in terms of build performance. This makes actions to have possibly inconsistent effects on build performance across projects.

Below, we highlight the key findings in relation to the interplay between build durations and breakages.

- Retrying failed commands can fix build breakages but, in return, may slow down builds, and thus should be avoided in favor of real build fixes.
- CI caching can reduce build breakages in addition to build durations, but may cause occasional breakages if corrupted or outdated.
- Running builds under various environments makes it prone to long build durations and also build breakages.
- Allowing builds to fast finish infers partial build passing, thus skipping failures under some environments.
- Developers with external expertise help gain better build performance, but may pay less attention to build durations in favor of passing builds.
- Projects may have strategic decisions in which preference is given to passing over timely builds or vice versa.

These findings have direct implications for developers, researchers, and CI service providers, which are discussed in the following section, in which we encourage researchers and CI service providers to supply developers with more tools and guidelines to increase their awareness of the possible interplay between CI build durations and breakages.

5 Implications

5.1 Implications for developers

Fixing build issues and monitoring their effectiveness rather than ignoring them. CI builds may encounter intermittent delays or breakages. In response, developers tend to ignore the real causes and, instead, retry or wait for failing or prolonged commands, which is also reported as a Continuous Delivery (CD) smell [39]. However, our results (Observation 2.2) show that such practices can result in an interplay between build durations and breakages, since besides increasing the build duration, builds may eventually break after retrials or waiting. For example, developers may fix a dependency installation issue by configuring the build to retry installation commands multiple times, but according to Observation 2.2, such retrials are likely to increase the build duration with no evidence that build breakages are resolved. Therefore, developers are encouraged to invest in developing permanent and concrete solutions to this kind of problems to improve the build performance, thus ensuring that build fixes really address the outstanding build issues.

Assessing where a project stands before making any build optimization decisions. Our results (Observation 3.1) suggest that CI is not "one size, fits all", which means that actions to improve the build performance of one project might not necessarily apply to other projects. Project context and where it stands in terms of build performance can limit the

actions to reduce build durations and breakages, thus giving less room to control interplay. Hence, developers should pay attention to 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.

Reducing both build breakages and durations. Developers rely on the documentation provided by CI service providers when configuring builds. However, developers may be unaware of the consequences of such configurations and their possible interplay on the durations and breakages of CI builds. For example, dependency caching is known to speed up builds, but our results (Observation 3.3) show that caching can also reduce build breakages, as it skips dependency installation, but can cause occasional breakages if outdated. This alarms developers to be careful when dealing with such configurations.

Staying vigilant and up-to-date with CI practices. Build performance can improve or decline at any point of time. Therefore, developers should regularly keep an eye on build durations and breakages to spot any anomalous change. In addition, our results (Observation 4.2) show that developers tend to sacrifice build durations for passing, or vice versa, for about a third of the projects, thus indicating an evident interplay. Nevertheless, recovering from such situations can be possible if developers stay up-to-date with the common CI configurations and recommendations used in practice. For example, as Observation 2.3 suggests, the OS distribution can play a role in the perceived build performance. Not specifying the OS distribution makes TRAVIS CI use the default one, which in turn can change from time[17] to time[18]. Changes in default OS distributions can be accompanied by changes in build performance, thus encouraging developers to regularly check CI updates, since such updates are typically not communicated to developers.

5.2 Implications for researchers

Awareness of the importance of project characteristics when reported research findings. While much research has studied build durations and breakages, only a few studies have considered what builds belong to which projects (e.g., using project-wise statistical testing [10] or mixed-effects regression [9, 13]). Other studies mixed builds from different projects together. According to our results (Observation 1.2), project characteristics can limit the options available to developers to improve build performance, thus increasing the chances of having interplay between build durations and breakages. Therefore, we encourage researchers to consider variances among projects to help developers identify the best practices that fit their projects.

Highlighting side effects when reporting actionable findings. Researchers should make clear any possible interplay, both 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 (e.g., Observations 2.2 and 3.3). For example, as prior research reported [9, 40], CI caching helps reduce the build duration. However, our experimental and survey results show a conflicting interplay with respect to caching. In particular, based on our experimental results (Observation 3.3), CI caching may also reduce build breakages, whereas our survey results indicate that caching can rather cause occasional build breakages (e.g., due to outdated cached dependencies). Therefore, exploring all prospects of a CI issue may help identify undesirable side effects when taking a certain action.

Approaches for detecting/reporting the inefficient adoption of CI configurations. Our results (Observation 3.2) suggest that not all projects can benefit from certain development or CI practices. 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 identify whether certain CI actions achieve their anticipated benefits, with the possible interplay if any, given that the current build performance of a project can impede such a practice from performing well.

5.3 Implications for CI services

CI service providers should continuously improve CI documentation. Despite the details/examples provided by CI services (e.g., TRAVIS CI[19]), certain CI configurations may to unexpected build performance, as our results show (e.g., Observation 2.2), expressed by a possible interplay between build durations and breakages. Hence, side effects and misconfiguring CI builds could be a result of unclear, inadequate, or misunderstood CI documentation.[20] Therefore, CI services should maintain clear and updated documentation to avoid misleading developers when configuring builds.

6 THREATS TO VALIDITY

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

6.1 Construct Validity

Construct threats to validity are concerned with the degree to which our analyses measure what we claim to analyze. In our experimental 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, survey respondents did not respond to some open-ended questions, which might not provide enough evidence about a certain experimental observation. To address this issue, we distinguish the responses of respondents who agree from the responses of those who disagree with an observation.

Build breakages in our dataset refer to both errored and failed builds, which can be caused due to different reasons. However, prior CI research considers error and failed builds as broken [10, 13, 21, 23]. In addition, metrics used in our regression models can capture the differences between build errors and failures. For example, if a metric is associated

17. https://docs.travis-ci.com/user/precise-to-trusty-migration-guide
18. https://docs.travis-ci.com/user/trusty-to-xenial-migration-guide
19. https://docs.travis-ci.com
20. https://github.com/travis-ci/travis-ci/issues/5700

with only build errors but not build failures, it would show no significance in our models. Yet, distinguishing factors that may co-occur with build errors from those that co-occur with build failures is interesting but remains to be investigated in future research. Moreover, our study does not make distinction between builds of different project branches, which might make the generalization of median build durations and breakage ratio across branches misleading. Though some branches might be created to experiment with new features or bug fixing, developers still need to investigate the causes of any anomalous build performance among these branches to ensure well-performing builds when merging with the main branch. We should also note that builds belonging to the main branches of the projects in our dataset are dominant, representing over two thirds (72%) of all builds, and there is no sufficient evidence that build breakage ratios differ across branches. Therefore, we believe that controlling for branching in our experiments is less likely to affect our overall conclusions, but is worth investigating in the future. Finally, while build durations may vary within a project, the main objective of our study is to construct a holistic view of the overall build performance in the studied projects. Moreover, we accounted for differences in build stages (installation, script, and deployment), environments, or test density between different project branches by including relevant metrics that correspond to them in our regression models (see Table 1). Future research should further extend our analyses to investigate interplay across build environments or project branches.

6.2 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 experimental 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.

The respondents of our survey study are software developers who have experience with TRAVIS CI and GITHUB. However, the responses we obtained might not reflect current CI practices. To mitigate this issue, we targeted developers who have recent contributions to the projects in our dataset. Moreover, the questions about the possible association of project characteristics with both build durations and breakages might make respondents hesitant, especially if one characteristic is believed to be associated with one performance measure but not the other. However, we expect respondents to agree or disagree with this observation depending on how many characteristics they believe to have such a relationship. If one characteristic is related but another characteristic is not, then we expect respondents to give a 'Neutral' response. Finally, our survey does not define some terms (e.g., *build performance* and *fast_finish*) as we assume respondents to have experience with CI. Nevertheless, this could lead respondents to have misconceptions about the experimental observations in question. To address this issue, we perform a manual analysis of open-ended responses, thus enabling the identification of any misconceptions that might have occurred.

6.3 External Validity

External threats are concerned with our ability to generalize our results. Our experimental study is based on builds 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. Hence, a replication of our study using projects written in other programming languages or linked with other CI services is important to reach more general conclusions. Moreover, experimental observations are verified using a developer survey, which is responded to by 139 respondents. Though such a response rate can be relatively low, which could be due to our large number of open-ended questions, we ensure that the developers we contacted span across the majority (94%) of the studied projects. Given the anonymity of the responses, we are unable to differentiate between respondents and non-respondents to the survey. Nevertheless, the survey results show that respondents cover a variety of demographics, in terms of development and CI experience, roles, and locations, thus making our observations supported by a wide range of viewpoints. Finally, some of the survey questions might introduce bias to the analyzed responses. In particular, one of our demographic questions confuses gender with sex and does not provide an option for non-binary genders and used '*Other*' and '*Prefer not to answer*' instead. However, there was no intentional preference towards a specific gender in our survey study, but we rather followed a similar practice in the literature [41, 42]. We expect respondents of non-binary genders to choose '*Other*'. Further, all our survey questions were reviewed and approved by the General Research Ethics Board at the academic institution where this research is conducted.

7 RELATED WORK

In this section, we summarize research related to build durations and breakages while highlighting our contributions.

7.1 Studies on CI build durations

Much research has studied build durations [4–9, 14, 15]. Rasmusson [4] reported that team spirits are negatively impacted when builds take longer to generate. Rogers [5] reported that long build durations can considerably interrupt software development. Brooks [6] and Hilton et al. [8] reported that ten minutes is the most acceptable build duration that developers may desire. However, recent studies [9, 15] reported that only 16% of CI builds have durations under ten minutes. Moreover, prior studies reported recommendations for developers to reduce the build duration, such as performing the integration once a week [5] or splitting large builds into smaller builds [7] Hilton et al. [8] reported that developers reduce the duration of CI builds by eliminating unnecessary dependencies or

tests. Ghaleb et al. [9] have investigated the reasons behind long build durations and reported the best practices to generate timely builds.

Despite the valuable research on build durations, it is unclear whether reducing the build duration would have consequences on build breakages. In our study, we investigate how maintaining timely builds may have a positive or negative side effect on build breakages.

7.2 Studies on CI build breakages

Prior research has studied build breakage modeling [10–12, 43, 44]. Jin and Servant [45] have proposed an approach to reduce the cost of CI builds while by running fewer builds (those that are likely to fail), while skipping builds for changes that are likely to pass. The authors found that project size, in terms of lines of code and tests, and CI lifespan are the most associated factors with build breakages. Rausch et al. [10] have studied the impact of 16 metrics on build breakages. The authors observed that the complexity of changes and historical breakage ratio are strongly associated with build breakages. Saidani et al. [46] have used evolutionary search to predict build breakages. The authors found that team size and types of changed files are associated with build breakages. Jain et al. [43] have performed a causation analysis of build breakages and also found that larger team sizes make more breakages. In addition, the authors observed that their findings are not sensitive to differences in programming languages or projects. Conversely, Islam et al. [44] found that project and team sizes are not associated with build breakages. Instead, the authors found that the building tools and complexity of changes are associated with build breakages.

Despite the valuable research on build breakages, little is known about whether fixing a build breakage may impact the build duration. In our study, we model build breakages with build durations together to investigate how likely they impact each other.

7.3 Studies on CI build durations and breakages

Previous research has studied build durations and build breakages independently. For example, Vassallo et al. [14] have studied (a) the evolution of build durations and (b) the frequency of breakages in the *master* repository branches. Wagner et al. [15] have performed independent analyses on the proportion and frequency of build durations and build breakages. Ghaleb et al. [9] have studied how build durations of *broken* and *passed* builds can vary. Pan et al. [47] have evaluated techniques and features used to reduce the build duration by selecting and/or prioritizing tests that are likely to break the build and found that build history is significantly more important than code coverage and complexity.

In summary, independent analyses on build durations and breakages may lead to unexpected build performance. In our study, we investigate both the build duration and build breakage and the possible interplay between them.

8 Conclusion

In this paper, we conduct an empirical study to investigate the interplay between reducing build durations and build breakages. In particular, 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. Moreover, we conduct a user study with software developers who contribute to the projects in our dataset, and thus have experience with CI, to get their feedback on our experimental observations. We summarize the key findings of our study as follows:

- Actions to improve the build performance of a project are dependent on the context and current build performance of that project.
- Project characteristics (e.g., project maturity, age, and build configuration ratio) have significant associations with build performance.
- Metrics that are commonly known to be associated with one build performance measure (e.g., the build duration) can also have positive or negative associations with the other measure (e.g., the build breakage).
- Developers should keep an eye on CI builds as build performance is subject to change as a result of changes to software development or building practices.

In summary, our experimental results show an evident interplay between build durations and breakages, with dual or inverse side effects. The majority of our experimental observations are confirmed by survey results, which provide useful insights, though some survey responses disagree with some of our experimental observations.

Feedback from survey respondents indicates that, though build performance can be improved using workarounds, developers should instead focus on addressing the root causes of CI build problems. Therefore, we encourage researchers and CI service providers to supply developers with more tools and guidelines to increase their awareness of the possible interplay between CI build durations and breakages.

Future work. We aim in the future to further extend our study to include more CI practices from industrial projects to investigate whether our findings would hold. We also plan to expand our empirical analyses to explore CI building practices in other commonly adopted CI services, such as GitHub Actions, CircleCI, Jenkins. Finally, future research should control for project branching to investigate whether our findings hold across different branches and under different software development paradigms.

References

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

[2] 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.

[3] 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*, pages 426–437. ACM, 2016.

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

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

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

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

[8] 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*, pages 197–207. ACM, 2017.

[9] Taher Ahmed Ghaleb, Daniel Alencar Da Costa, and Ying Zou. An empirical study of the long duration of continuous integration builds. *Empirical Software Engineering*, 24(4):2102–2139, 2019.

[10] 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*, pages 345–355, 2017.

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

[12] 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*, pages 311–315, 2017.

[13] 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*, 47(9):1998–2011, 2021.

[14] Carmine Vassallo, Sebastian Proksch, Harald C Gall, and Massimiliano Di Penta. Automated reporting of anti-patterns and decay in continuous integration. In *Proceedings of the 41st International Conference on Software Engineering*, pages 105–115. IEEE Press, 2019.

[15] 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*, pages 1–10, 2019.

[16] 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*, pages 447–450, 2017.

[17] 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.

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

[19] Studying the Interplay between the Durations and Breakages of Continuous Integration Builds (Replication Package). https://taher-ghaleb.github.io/papers/tse_2022/interplay/replication_package.html.

[20] Johannes Nicolai. GitHub welcomes all CI tools. https://github.blog/2017-11-07-github-welcomes-all-ci-tools. Visited on January 20, 2021.

[21] 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*, pages 356–367, 2017.

[22] 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*, pages 139–142. IEEE, 2017.

[23] Islem Saidani, Ali Ouni, and Mohamed Wiem Mkaouer. Improving the prediction of continuous integration build failures using deep learning. *Automated Software Engineering*, 29(1):1–61, 2022.

[24] Sakina Fatima, Taher A Ghaleb, and Lionel Briand. Flakify: A black-box, language model-based predictor for flaky tests. *IEEE Transactions on Software Engineering*, 2022.

[25] 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.

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

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

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

[29] 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.

[30] Donna Spencer. *Card sorting: Designing usable categories*. Rosenfeld Media, 2009.

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

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

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

[34] Keheliya Gallaba and Shane McIntosh. Use and misuse of continuous integration features: An empirical study of projects that (mis) use travis ci. *IEEE Transactions on Software Engineering*, 46(1):33–50, 2018.

[35] 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*, pages 401–405. IEEE, 2014.

[36] Mahdis Zolfagharinia, Bram Adams, and Yann-Gaël Guéhéneuc. Do not trust build results at face value: an empirical study of 30 million CPAN builds. In *Proceedings of the 14th International Conference on Mining Software Repositories*, pages 312–322, 2017.

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

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

[39] 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.

[40] Keheliya Gallaba, Yves Junqueira, John Ewart, and Shane Mcintosh. Accelerating continuous integration by caching environments and inferring dependencies. *IEEE Transactions on Software Engineering*, 2020.

[41] Prem Devanbu, Thomas Zimmermann, and Christian Bird. Belief & evidence in empirical software engineering. In *Proceedings of the 38th international conference on software engineering*, pages 108–119, 2016.

[42] Fernando Kamei, Igor Wiese, Gustavo Pinto, Márcio Ribeiro, and Sérgio Soares. On the use of grey literature: A survey with the brazilian software engineering research community. In *Proceedings of the 34th Brazilian Symposium on Software Engineering*, pages 183–192, 2020.

[43] 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.

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

[45] Xianhao Jin and Francisco Servant. A cost-efficient approach to building in continuous integration. In *IEEE/ACM 42nd International Conference on Software Engineering*, pages 13–25. IEEE, 2020.

[46] Islem Saidani, Ali Ouni, Moataz Chouchen, and Mohamed Wiem Mkaouer. Predicting continuous integration build failures using evolutionary search. *Information and Software Technology*, 128:106392, 2020.

[47] Rongqi Pan, Mojtaba Bagherzadeh, Taher A Ghaleb, and Lionel Briand. Test case selection and prioritization using machine learning: a systematic literature review. *Empirical Software Engineering*, 27(2):1–43, 2022.

Authors' Biographies

Taher A. Ghaleb is a Postdoctoral Research Fellow at the School of EECS at the University of Ottawa, Canada. Taher obtained his Ph.D. in Computing from Queen's University, Canada (2021). During his Ph.D., Taher held an Ontario Trillium Scholarship, a highly prestigious award for doctoral students. He worked as a research/teaching assistant since he obtained his B.Sc. in Information Technology from Taiz University, Yemen (2008) and M.Sc. in Computer Science from King Fahd University of Petroleum and Minerals, Saudi Arabia (2016). His research interests include continuous integration, software testing, mining software repositories, applied data science and machine learning, program analysis, and empirical software engineering.

Safwat Hassan is an Assistant Professor in the Faculty of Information, University of Toronto, Canada. He completed his Ph.D. degree at Queen's University in the Software Analysis and Intelligence Lab (SAIL). He worked as a software engineer for ten years in different corporations, including the Egyptian Space Agency (ESA), HP, EDS, VF Germany (outsourced by HP), and Etisalat. During his ten years in the software industry, he worked on multiple software systems, such as web-based systems and embedded systems. He also participated in diverse project roles (e.g., design service, customer support, and R&D) across multiple business domains (e.g., telecommunication, supply-chain, and aerospace). His research interests include data mining for software engineering, mobile app store analytics, software architecture, system anomaly prediction, continuous integration, and software performance analytics. More about Safwat Hassan can be read on his website: https://safwathassan.com

Ying (Jenny) Zou is the Canada Research Chair in Software Evolution. She is a professor in the Department of Electrical and Computer Engineering, and cross-appointed to the School of Computing at Queen's University in Canada. She is a visiting scientist of IBM Centers for Advanced Studies, IBM Canada. Her research interests include software engineering, software reengineering, software reverse engineering, software maintenance, and service-oriented architecture. More about Dr. Zou and her work is available online at https://seal-queensu.github.io