

An Empirical Study of the Long Duration of Continuous Integration Builds

Taher Ahmed Ghaleb · Daniel Alencar
da Costa · Ying Zou

Author pre-print copy. The final publication is available at Springer via:
<http://dx.doi.org/10.1007/s10664-019-09695-9>

Abstract Continuous Integration (CI) is a set of software development practices that allow software development teams to generate software builds more quickly and periodically (e.g., daily or even hourly). CI brings many advantages, such as the early identification of errors when integrating code. When builds are generated frequently, a *long* build duration may hold developers from performing other important tasks. Recent research has shown that a considerable amount of development time is invested on optimizing the generation of builds. However, the reasons behind *long* build durations are still vague and need an in-depth study. Our initial investigation shows that many projects have build durations that far exceed the acceptable build duration (i.e., 10 minutes) as reported by recent studies. In this paper, we study several characteristics of CI builds that may be associated with the *long* duration of CI builds. We perform an empirical study on 104,442 CI builds from 67 GitHub projects. We use mixed-effects logistic models to model *long* build durations across projects. Our results reveal that, in addition to common wisdom factors (e.g., project size, team size, build configuration size, and test density), there are other highly important factors to explain *long* build durations. We observe that rerunning failed commands multiple times is most likely to be associated with *long* build durations. We also find that builds may run faster if they are configured (a) to cache content that does not change often or (b) to finish as soon as all the required jobs finish. However, we observe that about 40% of the studied projects do not use or misuse such configurations in their builds.

Taher Ahmed Ghaleb
School of Computing
Queen's University, Kingston, Canada
E-mail: taher.ghaleb@queensu.ca

Daniel Alencar da Costa
Department of Information Science
University of Otago, New Zealand
E-mail: danielcalencar@otago.ac.nz

Ying Zou
Department of Electrical and Computer Engineering
Queen's University, Kingston, Canada
E-mail: ying.zou@queensu.ca

In addition, we observe that triggering builds on weekdays or at daytime is most likely to have a direct relationship with *long* build durations. Our results suggest that developers should use proper CI build configurations to maintain successful builds and to avoid *long* build durations. Tool builders should supply development teams with tools to identify cacheable spots of the project in order to accelerate the generation of CI builds.

1 Introduction

Building software is the process of automatically transforming software artifacts (e.g., source code) into deliverables, such as executables and libraries [57]. Build systems (e.g., `make` [21] and `ANT`¹) allow developers to constantly re-build testable artifacts after performing code changes. Maintaining build systems requires substantial effort [35], since these build systems evolve along with software source code [41]. For example, modern build systems (e.g., `Maven`²) generate more code churn, which requires a higher maintenance effort [42]. Neglecting build maintenance may lead to a build breakage [55], which is mostly caused by dependency-related issues [55, 58, 59]. Thanks to the advent of the Continuous Integration (CI) practices [23], builds can be generated more frequently (e.g., daily or even hourly), which allows the earlier identification of errors [62].

CI builds can be broken due to several reasons, such as compilation errors, dependency errors, configuration errors, or test failures [51, 64]. Xia and Li [66] and Ni and Li [47] introduced prediction models to predict build failures with AUC values of over 0.80 and 0.85, respectively. Studies also show that tests are central to the CI process [7], while process factors, build stability, and historical committer statistics are among the best indicators of a build failure [47, 51].

Nevertheless, When adopting CI, the time invested in building a project should be as short as possible, since the frequency of generated builds increases in magnitude [11]. In such a context, *long* build durations generate a negative overhead to the software development process, since developers would need to wait for a long time before engaging in other development activities [28, 50, 52]. In addition, the energy cost of running CI builds increases as the build duration increases, which is an emerging concern [29]. Hilton et al. [28] found that developers may feel disappointed when builds take a long time to be generated. Such challenges encouraged researchers to study the approaches for reducing the durations of builds. For example, Ammons [3] proposed an approach to split builds into a set of incremental mini-builds in order to speed up the build generation process. Recent user studies suggest that the most acceptable build duration is 10 minutes [11, 28]. Bisong et al. [10] introduced a number of regression models that aim to estimate the build duration. The authors were motivated by the study performed by Laukkanen and Mäntylä [36], which has demonstrated the lack of quantitative analysis on build durations.

Although Bisong et al. [10] studied the build duration, the factors associated with *long* build durations were not investigated. In addition, the models presented by Bisong et al. were not entirely suitable for predictions, since post-build factors were used in their models (e.g., the number of tests runs, test duration, and CI latency). Moreover, the models were built to estimate the individual durations of

¹ <http://ant.apache.org>

² <http://maven.apache.org>

build jobs instead of the perceived durations of builds. Nevertheless, the reasons behind *long* build durations are still vague and need an in-depth study. In addition, our exploration of 104,442 CI builds of 67 projects reveals that 84% of CI builds last more than the acceptable 10 minutes duration [11,28].

Therefore, we empirically investigate which factors are associated with *long* build durations. This investigation is important to help software engineers to reduce the duration of CI builds. We perform our study using 67 GitHub projects that are linked with Travis CI, a distributed CI build service. To collect our data, we use TravisTorrent [8], which is a publicly available dataset that contains information about CI builds of 1,283 projects. Despite the relatively small sample of projects in our study, we should note that our dataset contains well-known and previously studied projects (e.g., **rails**, **jrubby**, and **openproject**). Moreover, our study selects projects with high variations of build durations where the problem of *long* build durations may occur. To this end, we select 67 projects that have a build durations Median Absolute Deviation (MAD) above 10 minutes [11,28]. To capture the different characteristics of the studied projects in relation with *long* build durations, we use mixed-effects logistic models to model *long* build durations.

Goals and research questions.

In terms of the empirical software engineering research method, we should note that the goals and research questions (RQs) of our study are *exploratory* in nature [17]. In particular, the goal of this study is to empirically conduct an *exploratory* research to (a) study the frequency of *long* build durations; (b) investigate the most important factors that may have an association with *long* build durations; and (c) gain insights about CI build configurations and practices that may help developers to mitigate *long* build durations. Based on the above goals, we address the following *exploratory* RQs:

RQ₁: What is the frequency of long build durations?

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

RQ₂: What are the most important factors to model long build durations?

We classify build durations to be either *short* or *long* based on the *lower* and *upper* quantiles of the perceived build duration. Then, we use mixed-effects models to study the association of various factors on *long* build durations, taking into consideration the fixed and random effects. Our top-performing model obtains a good discrimination power (i.e., AUC value of 0.87). The model shows that *long* build durations can be explained using less obvious factors (e.g., caching, rerunning failed commands, and the day or time of triggering CI builds).

RQ₃: What is the relationship between long build durations and the most important factors?

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

Observations and implications.

Our results reveal that configuring CI builds to avoid build breakages may have a strong association with *long* build durations. For example, increasing the number of times to rerun failing commands will most likely be associated with *long* build durations. However, rerunning failing commands reduced the ratio of build failures by a median of 3% only. Our results suggest that developers should use proper CI build configurations to maintain successful builds with *short* build durations. Tool builders should supply development teams with tools to identify cacheable spots of the project in order to accelerate the generation of CI builds. Knowing the current workload of servers at the time of triggering a CI build may help developers to expect delays in running their builds.

Paper organization.

The rest of this paper is organized as follows. Section 2 presents background material about CI builds. Section 3 introduces the experimental setup of our empirical study. Section 4 discusses the results and findings of our studied RQs. Section 5 discusses the implications of our findings for developers, researchers, tool builders, and CI services. Section 6 describes threats to the validity of our results. Section 7 presents the related literature on CI builds. Finally, Section 8 concludes the paper and outline avenues for future work.

2 Background

In this section, we introduce an overview of continuous integration in general, and how it is implemented in Travis CI.

Continuous integration (CI) is a set of development practices that automate the builds of software projects every time a team member submits changes to a central code repository [23]. CI increases the frequency of code integration, testing, and packaging. For example, software builds can be generated daily or even hourly. Smaller and more frequent commits can help developers to reduce the debugging effort and keep track of the development progress. Hence, CI encourages developers to break down development tasks into smaller activities to improve software quality and to reduce any potential risks [16].

CI emerged in the eXtreme Programming (XP) community [6], the most widely used agile methodology. Nowadays, CI is broadly adopted in the GitHub pull-based development model. As reported by Bernardo et al. [9] and Vasilescu et al. [63], CI enables development teams to address more pull requests than before. CI automates the build process, including unit and regression tests, under a unified infrastructure (i.e., Web frameworks and build tools), which frees developers from the burden of maintaining their own local integration environments. Unified structure prevents the “it works on my machine” syndrome before projects are deployed [44]. Travis CI,³ AppVeyor,⁴ and CircleCI⁵ are examples of GitHub-compatible, cloud-based CI tools.

CI has a well-defined life-cycle when generating builds. Fig. 1 depicts the main phases of the CI build life-cycle. A CI build is normally triggered once a developer pushes a commit or submits a pull request to a remote repository in a version

³ <https://travis-ci.org>

⁴ <https://appveyor.com>

⁵ <https://circleci.com>

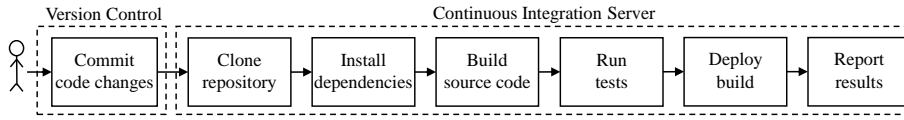


Fig. 1: The CI build life-cycle

control system. A CI build can also be (re)started manually. The CI building process starts by cloning the remote repository into the CI server. After installing all the required dependencies, the production code is built, followed by running the unit and integration tests. Finally, if all the previous phases are successful, the build is deployed and the development team is notified with the results. If any of the phases fails, the building process stops and a feedback is sent to the development team.

Travis CI is an open source, distributed, CI tool that supports more than 25 programming languages. Every GitHub repository can be configured to use Travis CI to automatically generate CI builds. Travis CI provides a free service for open source projects and a paid service for private projects. Travis CI offers different subscription plans based on customer needs.

Travis CI maintains a customizable build life-cycle. In Travis CI, every programming language has a default configuration, which normally consists of all the CI build phases but combined into two main build phases (*install* and *script*) and an optional *deploy* phase [7]. In the *install* phase, the remote repository is cloned and all the dependencies are installed. In the *script* phase, the software is built and tests are run. In the *deploy* phase, the software is packaged and deployed to a continuous deployment provider. Travis CI employs a configuration file, i.e., *.travis.yml*, in the root directory of the Git repository to allow development teams to customize the build phases. For example, developers can define the preferred build environment(s), test commands, and dependencies. In addition, Travis CI allows to write custom instructions and options that are run *before*, *after*, or *in-between* the *install*, *script*, and *deploy* phases. The building process in Travis CI may be interrupted at any point due to an error, a failure, or a manual cancellation. A build is considered successful if it passes all CI build phases.

A build on Travis CI may consist of multiple jobs. Each job is responsible for achieving a certain task, such as running the build on a specific runtime environment. Multiple build jobs may target multiple integration environments. Build jobs are run independently of each other. A CI build (and each build job) may have one of the following statuses: *passed*, *errored*, *failed*, or *canceled*. A CI build (or a build job) is marked as *passed* if it is successful in running all the build phases. The build fails if any of the build phases or build jobs fail. The *errored* status indicates that there is a problem with installing build dependencies, while the *failed* status reveals that the project encountered a failure when compiling or testing the software. The *canceled* status indicates that the build was manually stopped from running. Developers can mark some of the build jobs as `allow_failures`, which means that their failure does not impact the overall status of the build.

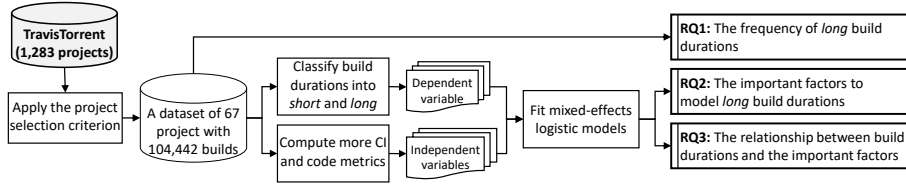


Fig. 2: Overview of our study

3 Experimental Setup

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

3.1 Data Collection

Fig. 2 gives an overview of our study. Our study is based on data collected from TravisTorrent [8]. TravisTorrent, in its 11.1.2017 release, stores CI build data of 1,283 projects. These projects are written in three programming languages: Ruby, Java, and JavaScript. Each entry in TravisTorrent represents a build job. For example, if a build is triggered using x jobs, TravisTorrent records x data entries for that build. All the x entries are recorded using the same build *id* and a different job *id*. Nevertheless, the values of the selected factors, including build duration values, are the same in all the job entries for a given build. Considering that our study focuses on *long* durations of builds, we keep a single entry for each build *id* and discard the rest of the entries of a build. We keep the information about how many jobs in each build. Each job has its own duration reported by Travis CI. However, given the fact that build jobs can run in parallel on Travis CI, using the sum of durations of all build jobs could be misleading (i.e., the perceived build duration is likely smaller than the sum of durations of all jobs). Therefore, we collect the perceived build duration represented by the difference between the time when the build started and the time when the build finished.

We identify a criterion to select the subject projects in our study based on the high variations of build durations. Given that we are interested in studying the factors that are associated with *long* CI build durations, we need to study the projects that have higher variation in their build durations. For example, if the variation of build durations of a given project is relatively small (e.g, one or two minutes), developers may disregard that variation, since it could simply be caused by the load on CI servers. Therefore, we select projects that have a variation of build durations above 10 minutes [11,28]. We use the Median Absolute Deviation (MAD) [31] to measure the absolute deviation from the median of a given distribution of build durations. The higher the MAD, the greater the variation of the build durations in each of the subject projects. We obtain the start and finish timestamps for each build of these projects from Travis CI in order to compute the perceived build duration. We compute the MADs of the perceived build durations for each project. We filter out the projects for which the build duration MADs are less than 10 minutes. 67 projects survive this criterion.

Table 1 provides an overview of the 67 projects that satisfy the selection criterion. The studied projects form a variety of domains, including, but not limited to, programming languages, tools, applications, and services. In addition, these

Table 1: An overview of the studied projects

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

projects are of different sizes in terms of lines of code and development teams. The number of builds of the subject projects is 104,442. We clone the Git repository of each studied project to compute CI build metrics. For example, we compute the experience of the developers who triggered the builds in terms of (a) the number of commits and (b) the number of days of development. We also analyze the

Travis CI configuration file (i.e., *.travis.yml*) of each build to compute metrics related to the build configuration that we use in the models.

3.2 Data Processing

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

3.2.1 Classification of Build Durations

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

3.2.2 Selection of Independent Variables

This step is concerned with the selection of the factors that are used as independent variables in the models.

Factors obtained from TravisTorrent: There are 61 data columns in TravisTorrent [8], in its 11.1.2017 release, that represent the characteristics of CI builds. In our analyses, we only consider the *build starting timestamp* and exclude the other timestamps (e.g., *the creation date of the first commit* and *the creation date of the PR*). In addition, we exclude the data columns in TravisTorrent that:

- do not represent factors, such as *build ids*, *pull request numbers*, *commit hashes*, and *test names*.
- contain values produced after the build is run, such as *the number of tests ran*, *build status*, and *setup time*. These factors are not suitable for modeling *long* build durations, since we cannot obtain their values before starting the build [10].
- have high percentages of *zero* or *NA* values (e.g., *the number of comments on a commit*), since they can impact the results of the regression models.

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

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

- ***SLOC delta***: We compute this factor by calculating the difference between the number of source lines of code of each build and its preceding build in the same branch.
- ***Day of week & Day or night***: We compute these two factors by deriving the day and the hour when the builds were started. Build starting timestamps are in the UTC time zone that is used by Travis CI.
- ***Lines of .travis.yml***: We compute this factor by counting the number of instruction lines in the Travis CI build configuration file (i.e., *.travis.yml*) of each build. We exclude the blank and comment lines.
- ***Configuration files changed***: We compute this factor by counting the number of project configuration files that have been changed by all the build commits. We consider a file as a project configuration file if it has one of the following extensions: *.yml*, *.xml*, *.conf*, *.rake*, *.rspec*, *.ruby-version*, *Gemfile*, *Gemfile.lock*, *Rakefile*, and *.sh*. To get the number of configuration files changed, we perform a *diff* at each build commit and count the number of unique files that end with the specified extensions.
- ***Configuration lines added/deleted***: We compute these two factors by counting the number of lines added or removed to/from all project configuration files. We perform a *diff* at each build commit on all the configuration files changed to get the total number of lines added/removed.
- ***Caching***: We compute this factor by analyzing the *.travis.yml* file to check whether it contains the *cache*: instruction.
- ***Fast finish***: We compute this factor by analyzing the *.travis.yml* file to check whether it contains the *fast_finish*: instruction.
- ***Travis wait***: We compute this factor by analyzing the *.travis.yml* file to check whether it implements *travis_wait*: in any of the build instructions.
- ***Retries for failed commands***: We compute this factor by analyzing the *.travis.yml* file to search for the maximum number of times to rerun failed commands using either a *--retry* command option or the *travis_retry* instruction.
- ***Author experience***: We compute two metrics as proxies for the experience of the developers who triggered the builds. We first get the name of the developer who authored the commit that triggered the build. Then, we search through all the commits that precede the build-triggering commit. We obtain the list of commits that were authored by the same developer who triggered the build. After that, we count the number of commits that each developer has and the number of days between the commit that triggered the build and the first commit of the developer in the repository. If a build was triggered by a commit that was authored by multiple developers, we obtain the experience factors for the author with the largest number of commits and days.

In Table 2, we categorize the final set of factors into five dimensions: CI factors (8), code & density factors (7), commit factors (8), file factors (7), and developer factors (4). We use these factors as independent variables in the models. We present a detailed description of each factor in the last column of Table 2.

3.2.3 Correlation and Redundancy Analysis

Regression models can adversely be affected by the existence of highly correlated and redundant independent variables [13]. Therefore, we perform correlation and

Table 2: Dimensions of factors used as independent variables in our mixed-effects logistic models

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

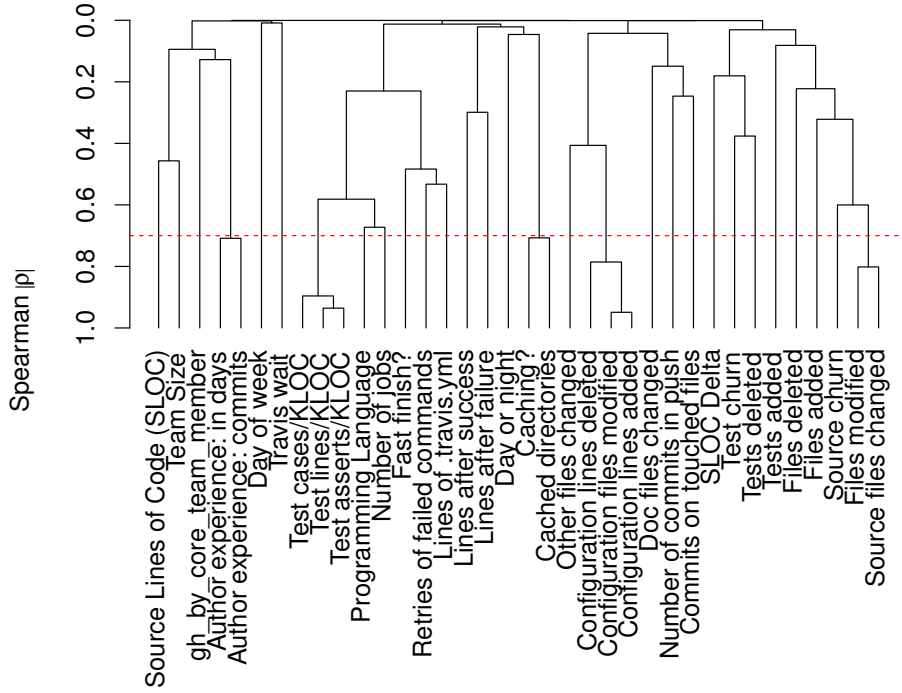


Fig. 3: The hierarchical clustering of independent variables in the studied projects

redundancy analyses for the independent variables used in our models. We follow the guidelines that are provided by Harrell [27] to train regression models.

Correlation Analysis: In this step, we employ the Spearman rank ρ clustering analysis [54] to remove highly correlated variables in each of the subject projects. To this end, we use the `varclus` function from the `rms`⁶ R package. For each pair of independent variables within all clusters that have a correlation of $|\rho| > 0.7$, we remove one variable and keep the other in the models. If two variables are highly correlated, we keep one variable in our models and remove the other variable. According to the principle of parsimony in regression modeling, simple explanatory variables should be preferred over complex variables [61]. Considering that our explanatory variables are equally simple (e.g., in terms of computation), we keep the variables that are more informative about the building process. For example, the *Lines of .travis.yml* is highly correlated with the *Number of jobs*. Therefore, we keep the *Lines of .travis.yml*, since it conveys more information about the build. Similarly, the *Test lines/KLOC* is highly correlated with the *Test cases/KLOC*. Therefore, we keep the *Test cases/KLOC*, since it has more specific details than the *Test lines/KLOC*.

In Fig. 3, we show the dendrogram of the hierarchical clustering of independent variables for the subject projects. In this dendrogram, we observe five clusters of highly correlated variables ($|\rho| > 0.7$). In Table 3, we present the highly correlated

⁶ <https://cran.r-project.org/web/packages/rms/rms.pdf>

Table 3: Selected variables of the highly correlated variables in the projects

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

variables and the selected variable within each cluster. We observe that there is an additional cluster (i.e., cluster 6) presented in Table 3. Such a cluster is formed after performing the variable selection in cluster 5.

Redundancy Analysis: In this step, we perform a redundancy analysis on the remaining 26 independent variables (i.e., those that survive the correlation analysis step). Redundant variables can distort the relationship between the other independent variables and the dependent variable (i.e., *short* or *long* build duration) [27]. To this end, we use the `redun` function from the `rms` R package, which models each independent variable using the remaining independent variables. If an independent variable can be estimated by other independent variables with an $R^2 \geq 0.9$, we discard such a variable [27]. By performing the redundancy analysis, we observe that our dataset has no redundant variables.

4 Experimental Results

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

RQ1: *What is the frequency of long build durations?*

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

ated with only *passed* builds. It can also suggest that *long* build durations might be associated with other important factors even if the build is *errored* or *failed*.

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

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

- We use the Kruskal-Wallis test [34] to investigate whether a *long* duration is associated with the *passed*, *errored*, *failed*, or *canceled* build statuses. The Kruskal-Wallis test is the non-parametric equivalent of the ANOVA test [22] to check whether there are statistically significant differences between three or more distributions of build durations. Considering that the Kruskal-Wallis test does not indicate which build status has significantly different build durations with respect to others, we use the Dunn test [15] to perform individual comparisons. For example, the Dunn test indicates whether the build durations that belong to the *passed* builds are statistically different when compared to the build durations that belong to the *failed* builds. To counteract the problem of multiple comparisons [14], we use the Bonferroni-Holm correction [30] along with our Dunn tests to adjust our obtained *p-values*.
- We use Cliff’s delta effect-size measures [12] to verify how significant is the difference in magnitude between the values of two distributions. The higher the value of the Cliff’s delta, the greater the magnitude of the difference between distributions of build durations. For instance, a significant *p-value* but a small Cliff’s delta means that, although two distributions do not come from the same population, their difference is not significantly large. We use the thresholds provided by Romano et al. [53] to perform our comparisons: *delta* < 0.147 (*negligible*), *delta* < 0.33 (*small*), *delta* < 0.474 (*medium*), and *delta* ≥ 0.474 (*large*).
- We use beanplots [33] to compare the distributions of build durations of the different projects. The vertical curves of beanplots summarize and compare the distributions of different datasets. The higher the frequency of data within a particular value, the thicker the bean is plotted at that particular value on the *y* axis.

Findings. *We observe that over 40% of the builds in our dataset took over 30 minutes to run.* We also observe that only 16% of the builds in our dataset have durations under 10 minutes. Fig. 4 summarizes the distributions of build durations of the 67 studied projects. In particular, Fig. 4 shows the distributions of the *minimum*, *lower quantile*, *median*, *upper quantile*, and *maximum* build durations of all the 67 projects. We observe that the median build duration varies in the studied projects, ranging from 8 minutes to 90 minutes (the overall median build duration is 20 minutes). We also observe that the 10-minute build duration is expressed by the median build duration of the *lower quantile* distribution. In

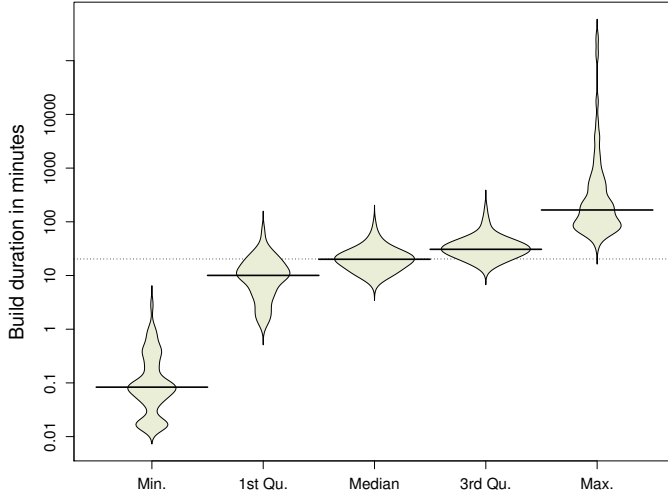


Fig. 4: The distributions of build durations of the studied projects

addition, the distributions of the *max* and *min* build durations are highly right-skewed (skewness values of 6.3 and 4.7, respectively). The median build duration of the *max* duration distribution is 2.77 hours. The overlap between the different distributions of build durations suggests that we should consider (a) modeling the build duration differences between the studied projects and (b) varying the threshold of classifying build durations as *short* or *long*.

In Table 4, we show statistics about the build durations of the top four projects in terms of their number of builds. We observe in Table 4 that the median build durations of the projects range between 24 and 36 minutes. We also observe that the majority of build durations in **rails** (i.e., 90%) and **jruby** (i.e., 70%) are in the range of 1–10 hours, with lower quantiles of around 2 and 0.7 hours and upper quantiles of 6 and 5 hours, respectively. Moreover, **jackrabbit-oak** has lower and upper quantiles of about 0.3 and 0.7 hours, respectively. Both **jackrabbit-oak** and **openproject** do not experience extremely long build durations as opposed to **rails** and **jruby**.

The build duration does not always increase over time. In Fig. 5, we show how build durations evolve over time for the top four projects. We show the durations of *passed* builds of such projects. We observe that build durations fluctuate as opposed to an increasing or decreasing trend of build durations over time. The fluctuation of build durations over time indicates that there are other possible factors that may have an association with the increase or decrease of build durations. We aim to investigate these factors in our following RQs.

Durations of passed builds are not always longer than the durations of errored and failed builds. Table 5 shows the number and percentage of projects in which the build durations are significantly different between build statuses. By intuition, since *passed* builds complete all the build phases, they are expected to be longer than builds with any other statuses. Nevertheless, we observe in Table 5 that durations of *passed* builds are not always the longest durations amongst build statuses. Hilton et al. [29] performed a similar analysis on a different dataset of

Table 4: Statistics of build durations of the top four projects (statistics of build durations of the full set of projects is available in our online appendix [1])

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

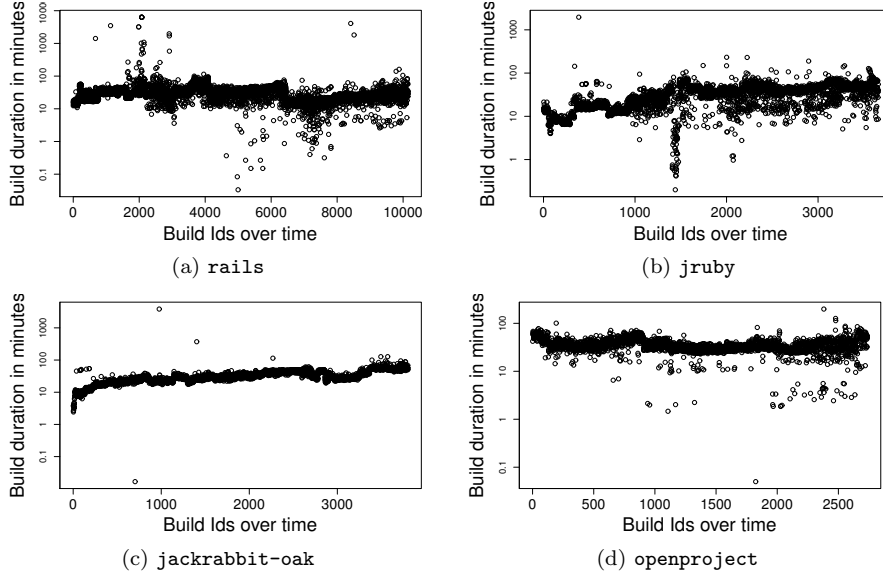


Fig. 5: Build durations over time for the top four projects (line plots for the full set of projects is available in an online appendix [1])

Table 5: The number and percentages of projects where build durations are significantly different from one build status to another.

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

34,544 projects. They also found that *passed* builds run faster than *errored* and *failed* builds. Their speculation of the results suggests that many of the *passed* builds that run faster may not have generated meaningful results.

Our project-wise analysis of the results reveals that durations of *passed* builds are not significantly longer than durations of *failed* and *errored* builds in more than 50% of the projects. In addition, *passed* builds run faster than *failed* and/or *errored* builds in 17 (i.e., 25%) projects (shown in Table 6). In 5 projects, *passed*

Table 6: List of projects where *passed* run faster than *failed* and/or *errored* builds

Project name	Passed <i>faster than</i> Failed	Passed <i>faster than</i> Errored
ark	✓	✓
canvas-lms		✓
capbara		✓
celluloid	✓	✓
celluloid-io	✓	✓
chef	✓	
diaspora		✓
druid		✓
jackrabbit-oak		✓
moped	✓	✓
openproject	✓	
promiscuous	✓	✓
ruboto		✓
skylight-ruby	✓	
twitter-cldr-rb	✓	
wicked		✓
xtreemfs		✓

builds run faster than both *failed* and *errored* builds. In 4 projects, *passed* builds run faster than *failed* builds only, whereas *passed* builds run faster than *errored* builds only in 8 projects. On the other hand, our results show that there is no significant difference between the build durations of *failed* and *errored* builds in 60% of the projects. Moreover, we observe that durations of *errored* builds are significantly longer than *failed* builds in 27% of the projects.

Gallaba et al. [24] found that the build status data may have noise. In particular, the status of a build may not always reflect the statuses of all the build jobs. For example, a *passed* build may contain *broken* (i.e., *failed* and *errored*) jobs that are ignored by developers (e.g., marked as `allow_failures`). In addition, a *broken* build may contain jobs that are *passed*. Therefore, we check if our results are sensitive to these types of noise in the build status data. Specifically, we consider that a build is (a) really *passed* if all jobs are *passed* and (b) really *broken* if all jobs are *broken*. Our cleaned dataset contains 64% of the total builds of our original data. We find that our observations hold for the majority of the studied projects. In particular, we observe that (1) the durations of *failed* builds are longer than the durations of *passed* in 6 projects, (2) the durations of *errored* builds are longer than the durations of *passed* in 8 projects, and (3) the durations of *errored* builds are longer than the durations of *failed* builds in 9 projects. Therefore, the noise in build statuses does not have a significant impact on our results. Moreover, the main focus of our study is to analyze *long* build durations. Therefore, the noise observed by Gallaba et al. does not impact our main observations.

We analyze a sample of the builds of the studied projects to investigate the reasons why the durations of *broken* builds could be longer than the durations of *passed* builds. In particular, we analyze builds of the *diaspora*⁷ project. We find that the median duration of *passed* builds is 5.6 minutes, whereas the median durations of *failed* and *errored* builds are 12.6 and 13.2 minutes, respectively. We manually analyze the build logs of the broken build jobs of the *diaspora*

⁷ <https://github.com/diaspora/diaspora>

project to investigate the reasons behind such results. We find that 43% of build failures in such jobs were due to commands that took longer than a certain limit of time to execute and were terminated by Travis CI. For example, running the `./script/ci/build.sh` script in build #2671⁸ of *diaspora* took longer than 25 minutes and was terminated by Travis CI. As a result, job #3 *failed* and took 5-13 minutes longer than the other *passed* jobs of the build. In addition, connection and test timeouts were the reasons behind the failures of 12% and 2% of the build jobs, respectively. Moreover, we observe that, in 13% of the build job failures, Travis CI retried the failing commands two to three times with no success. For example, build #4037⁹ of *diaspora* reran the command `bundle install ...` twice. However, job #4 *failed* while taking 8-24 minutes longer than the other *passed* jobs of the build.

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

RQ₂: *What are the most important factors to model long build durations?*

Motivation. RQ1 shows that CI build durations may behave differently across projects. The fluctuating trend of build durations over time suggests that there are factors that may have an association with the increase or decrease of build durations. In this RQ, we aim to understand the different factors that may have an association with *long* build durations. We take into consideration both common wisdom factors (e.g., project size, build configuration size, team size, and test density) and other factors that may have an association with *long* build durations.

Approach. Our dataset contains builds from 67 studied projects. Such projects are very different in terms of size and domain. As the results of RQ1 suggest, *long* build durations are different from one project to another. Therefore, we use a mixed-effects regression to control the variation between projects in terms of build durations. Mixed-effects logistic models allow to assign (and estimate) a different intercept for each project [65]. Considering that we aim to study the relationships between *long* build durations and the factors listed in Table 2, we particularly use the generalized mixed-effects models for logistic regression. Generalized mixed-effects models are statistical regression models that contain both fixed and random effects [20]. Fixed effects are variables with constant coefficients and intercepts for every individual observation. Random effects are variables that are used to control the variances between observations across different groups (i.e., projects). Our mixed-effects logistic models assume a different intercept for each project [39]. Traditional regression models, in contrast, use fixed effects only, which disregard the variances of build durations across projects.

Equation 1 shows the equation of the mixed-effects logistic model. In Eq. 1, Y_g denotes the binary build duration (i.e., *long* or *short*); β_0 demonstrates the constant intercept; X_i represents the independent variables; β_i represents the coefficients of each X_i ; ϵ_g indicates the errors; and θ_g represents the intercepts that vary across each project. We use the `glmer` function in the `lme4` R package to use

⁸ <https://travis-ci.org/diaspora/diaspora/builds/4033669>

⁹ <https://travis-ci.org/diaspora/diaspora/builds/10766342>

mixed-effects logistic models. We use the *binomial* distribution, *Laplace* approximation, and the *bobyqa* optimizer as parameters to the `glmer` function.

$$Y_g = \beta_0 + \theta_g + \sum_{i=1}^n \beta_i X_i + \epsilon_g \quad (1)$$

Significant independent variables are marked with asterisks in the output of the mixed-effects logistic models using the **ANOVA** test [49]. An independent variables is significant if it has $Pr(< |\chi^2|) < 0.05$. $Pr(< |\chi^2|)$ is the *p-value* that is associated with the χ^2 -statistical test. The χ^2 (Chi-Squared) values show whether the model is statistically different from the same model in the absence of a given independent variable according to the degrees of freedom in the model. The higher the χ^2 , the higher the explanatory power of an independent variable. We use *upward* (\nearrow) and *downward* (\searrow) arrows to indicate whether a variable has a direct or an inverse relationship, respectively, with the *long* build duration.

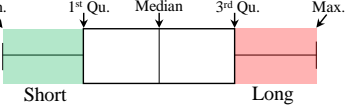
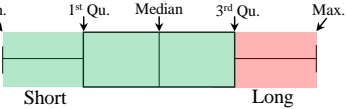
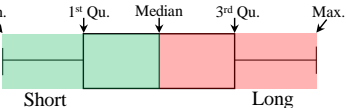
We compute the number of Events Per Variable (EPV) in the models. EPV shows the likelihood of a regression model to overfit [48]. EPV values represent the ratio of the number of builds with *long* durations to the number of independent variables. A dataset with an EPV above 10 is less risky to run into an overfitting problem [48].

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

- The Area Under the Curve (AUC) evaluates the diagnostic ability of the mixed-effects logistic models to discriminate *long* build durations [26]. AUC is the area below the curve created by plotting the true positive rate (TPR) against the false positive rate (FPR) using all the possible classification thresholds. The value of AUC ranges between 0 (worst) and 1 (best). An AUC value that is greater than 0.5 indicates that the explanatory model outperforms a random predictor.
- The *marginal* R^2 is a measure of the goodness-of-fit of our mixed-effects models. It represents the proportion of the total variance explained by the fixed effects [46]. Higher values of the *marginal* R^2 indicate that fixed effects can well explain the dependent variable (in our case, *long* build durations).
- The *conditional* R^2 is a measure of the goodness-of-fit of the mixed-effects models. It represents the proportion of the variance explained by both fixed and random effects [46]. Higher values of the *conditional* R^2 indicate that the proportion of the variance that is explained by both fixed and random effects is higher than the proportion of the variance that is explained by fixed effects only. A high difference between the values of *conditional* and *marginal* R^2 suggests that the random effects significantly help to explain the dependent variable.

We conduct a sensitivity analysis using three scenarios of different classification thresholds for *long* build durations. We perform the sensitivity analysis to study how the model is sensitive to the classification threshold for *long* build durations. Table 7 presents the thresholds for classifying build durations into *short* and *long* using three classification scenarios. For each classification scenario, we present the obtained number of builds with *short* and *long* and the number of independent

Table 7: Classification scenarios of build durations

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

variables that survive the correlation and redundancy analyses. We explain the three classification scenarios below:

- **Scenario 1:** Build durations below the *lower* quantile are considered *short*, while the durations above the *upper* quantile are considered *long*. This scenario classifies 25% of the builds as *short* and 25% of the builds as *long*.
- **Scenario 2:** Build durations below the *upper* quantile are considered *short*, while the durations above the *upper* quantile are considered *long*. This scenario classifies 75% of the builds as *short* and 25% of the builds as *long*.
- **Scenario 3:** Build durations below the median are considered *short*, while the durations above the median are considered *long*. This scenario classifies 50% of the builds as *short* and 50% of the builds as *long*.

Findings. *Our mixed-effects logistic models maintain a good discrimination performance when classifying long build durations using different duration thresholds.* Table 8 shows the performance of the mixed-effects logistic models in terms of AUC, *marginal* R^2 , and *conditional* R^2 . Our results indicate that the models maintain a high performance in all the classification scenarios for *long* build durations. We observe that the model using *Scenario 1* obtains a good AUC value of 0.87 for discriminating *long* build durations. In the second and third scenarios, the models obtain AUC values of 0.78 and 0.79, respectively. The *conditional* R^2 values of all the three models are higher than the values of the *marginal* R^2 values by 22%, 36%, and 19% respectively. The EPV values in the three scenarios are 932.61, 870.43, and 1,740.27, respectively. Such high EPV values indicate that the models are less likely to be overfitting. In our subsequent analyses, we use the results obtained by the top performing modeling scenario (i.e., *Scenario 1*).

Build durations have a strong association with CI build factors, such as (1) the build triggering time, (2) the number of times to rerun failing commands, (3) caching, and (4) finishing as soon as the required jobs finish. Table 9 presents the variable importance results obtained from the model

Table 8: Performance of the mixed-effects logistic models

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

fit using the best performing scenario (i.e., *Scenario 1*). All the independent variables are sorted based on their χ^2 values in a descending order. For each independent variable, we show its estimated coefficient (estimated coefficients of the days of week are presented in Table 10), its χ^2 value, the p -value (represented by $Pr(< \chi^2)$), its significance to model *long* build durations, and whether each independent variable has a direct or an inverse association with *long* build durations (represented by the *upward* and *downward* arrows). We observe that, as expected, the common wisdom factors (i.e., SLOC, lines of *.travis.yml*, team size, and test cases/KLOC) have significantly strong association with *long* build durations. However, we observe other less obvious factors to explain *long* build durations (e.g., caching, rerunning failing commands, time of triggering the build, and developer’s experience). We also observe that commit-level factors have a weak association with *long* build durations (e.g., tests added/deleted, files added/deleted, and the number of commits in a push). To better investigate the explanatory power of the less obvious factors to explain *long* build durations, we fit our top performing model without using the common wisdom factors (i.e., SLOC, lines of *.travis.yml*, team size, and test cases/KLOC). We observe that the model also maintains a good performance (i.e., the AUC values 85%). In addition, the model becomes more sensitive to project variances (i.e., the *conditional R^2* is higher than the *marginal R^2* by 0.46). Moreover, the less-obvious important factors preserve their explanatory power in both models (i.e., with and without the common wisdom factors).

RQ3: *What is the relationship between long build durations and the most important factors?*

Motivation. RQ2 suggests that *long* build durations are strongly associated with several important factors. In this RQ, we use the results obtained from our top performing model to study whether each important factor has a direct or inverse relationship with *long* build durations. In addition, we perform manual analyses on the builds of the subject projects to gain insights on the relationship between *long* build durations and the most important factors.

Approach. We use the values, generated by the top performing model, of the estimated coefficients of the independent variables. Estimated coefficients can be positive or negative. A positive coefficient indicates that the variable has a direct relationship with *long* build durations. A negative coefficient indicates that the variable has an inverse relationship with *long* build durations. We use the sign of the estimated coefficient of each variable to produce *upward* and *downward* arrows. The *upward* and *downward* arrows represent a direct or an inverse relationship between a variable and *long* build durations. We use the odds ratios [2] to measure the association of the dependent variable with the presence/absence of a binary independent variable (or the increase/decrease of a continuous independent variable) while holding the other variables at a fixed value. For example, odds ra-

Table 9: Results of the mixed-effects logistic model – sorted by χ^2 descendingly

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

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

*Individual coefficients for each day of the week are presented in Table 10

Table 10: Estimated coefficients obtained from the mixed-effects logistic model for each *Day of week*

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

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

tios can explain how the *long* duration differs between builds that use and builds that do not use caching. We compute odds ratios by taking the exponentiation of the estimated coefficients obtained from the model for each independent variables. For the *Day of week* independent variable, the odds ratio for each day of week is computed over the reference day (i.e., *Friday*).

Findings. *Configuring CI builds to finish as soon as the required jobs finish is most likely to be associated with short build durations.* The `fast_finish` setting in Travis CI allows builds to finish as soon as the status of the required build jobs is determined. In other words, the build is finished and its status is determined without the need to wait for jobs that are marked as `allow_failures`.

Such a feature shows a high importance in producing short build durations. Our results reveal a strong inverse association between fast build finishing and *long* build durations (χ^2 of 1343.62). Looking at the negative estimated coefficient (i.e., -0.72) of the *Fast Finish* independent variable, we observe that builds that are configured to finish as soon as the required jobs finish have significantly shorter durations than builds that are not configured with the `fast_finish` setting ($p\text{-value} = 2.2e^{-16}$). The odds of having a *long* build duration for builds with the `fast_finish` setting is 51% lower than the odds for builds without the `fast_finish` setting.

To gain more insights about builds that are configured to perform fast finishing, we analyze the `.travis.yml` file of the studied projects. We consider (a) builds that were triggered after supporting fast-finishing on Travis CI¹⁰ (i.e., November 27th, 2013); and (b) builds that have `allow_failures` jobs. We find that 37 of the studied projects have builds that contain `allow_failures` jobs. However, we observe that only 16 (i.e., less than half) of these 37 projects have their builds configured with the `fast_finish` setting. In 10 projects, builds with the `fast_finish` setting run faster than builds without that setting (a median difference of 9 minutes). We investigate the projects that experienced no major reductions of build durations after enabling the `fast_finish` setting. We observe that the more `allow_failures` jobs a build have, the more likely for the `fast_finish` setting to speed up the build generation. For example, the `killbill`¹¹ project experiences the least benefit of the `fast_finish` setting among all the other projects. In `killbill`, the median percentage of the `allow_failures` jobs to the total number of jobs is 20%. On the other hand, the `ruboto`¹² project had the maximum reduction of build durations after enabling the `fast_finish` setting. In `ruboto`, the median percentage of the `allow_failures` jobs to the total number of jobs is 53%. Therefore, development teams should consider enabling the `fast_finish` setting to receive feedback about their builds as soon as the required jobs finish.

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

To gain more insights about why builds might experience *long* build durations even with the use of caching, we analyze the `.travis.yml` file of the studied projects. We consider builds that were triggered after December 17th, 2014 (i.e., after Travis CI introduced the caching feature for open source projects¹³). We find that 42 (i.e., 63%) of the projects have the caching feature enabled in their builds. We observe that caching was actively used in 30 out of the 42 projects (i.e., caching was

¹⁰ <https://blog.travis-ci.com/2013-11-27-fast-finishing-builds>

¹¹ <https://github.com/killbill/killbill>

¹² <https://github.com/ruboto/ruboto>

¹³ <https://blog.travis-ci.com/2014-12-17-faster-builds-with-container-based-infrastructure/>

enabled in more than 80% of the builds). We observe that caching reduced the build duration by a median of 11 minutes for only 13 of these projects. Moreover, we investigate the projects that have no notable reduction in the durations of builds that perform caching. We observe that, in some projects (e.g., `killbill`¹⁴ and `flink`¹⁵) caching was enabled in the build configuration without specifying the content to cache. In other projects (e.g., `vanity`¹⁶ and `openproject`¹⁷), we observe that caching was applied mostly to `bundler`,¹⁸ a `gem` for dependency management, rather than specific directories. Since `bundler` maintains frequent updates, caching it is less likely to have a significant reduction to build durations. As a consequence, caching content that changes more often can introduce an overhead to the build generation process, since Travis CI may need to upload the cache frequently.

Maintaining a stable build status has a strong association with long build durations but with a negligible reduction in the build failure ratio.

Our results reveals that there is a trade-off between *long* build durations and the attempts of developers to maintain *passing* builds. In particular, developers may configure their builds to rerun failing commands multiple times to avoid having many build failures. However, we observe that such a configuration has a strong association with *long* build durations. It is true that allowing builds to rerun a failing command several times may help to reduce the ratio of build failures. However, developers should take into consideration that the more times a command fails, the more duration the build would take. Most of Travis CI internal build commands can be wrapped with `travis_retry` to reduce the impact of network timeouts.¹⁹ Looking at the positive estimated coefficient (i.e., 0.61) of the *Retries of failed commands* independent variable, we observe that the more reruns of a failed command in a build the longer the duration of that build. Such an association between the number of retries of failed commands and *long* build durations is significant ($\chi^2 = 667.21$ and $p\text{-value} < 2.2e^{-16}$). In addition, a one-unit increase in the number of times of rerunning failed commands increases the odds of having a *long* build duration by 84%.

To gain more insights about builds that rerun failing commands, we analyze the projects that have explicit configuration instructions for specifying the number of times to rerun failing commands. We find 13 of the studied projects with such a configuration in their builds. We analyze projects that have at least 10% of their builds configured to retry failing commands multiple times. We observe that the median duration of builds that are configured to rerun failing commands is 13 minutes more than the builds without such a configuration. Although we observe that rerunning failing commands several times reduced the ratio of build failures in 60% of the projects, we find that the median reduction is only 3%. We manually investigate a sample of builds of the `jruby`²⁰ project. Such builds were configured to retry failing commands for 3 times. We find that the duration of build # 11843²¹

¹⁴ <https://github.com/killbill/killbill>

¹⁵ <https://github.com/apache/flink>

¹⁶ <https://github.com/assaf/vanity>

¹⁷ <https://github.com/opf/openproject>

¹⁸ <https://bundler.io>

¹⁹ https://docs.travis-ci.com/user/common-build-problems/#travis_retry

²⁰ <https://github.com/jruby/jruby>

²¹ <https://travis-ci.org/jruby/jruby/builds/108164066>

of `jruby` is more than the duration of its preceding²² and succeeding²³ builds by 27 and 49 minutes, respectively. Although the commit²⁴ that triggered build # 11843 only updated the copyright year, the build was *errored*. The majority of the jobs of build # 11843 reran failing commands for 2–3 times. Therefore, developers should carefully study the number of times to rerun failing commands, since it can generate unnecessary waiting durations in broken builds.

Builds are more likely to have longer durations if they are triggered on weekdays or at daytime. We observe that the day of week factor is one of the important factors to model *long* build durations ($\chi^2=214.59$ with a *p-value* $< 2.2e^{-16}$). Table 10 shows the individual estimated coefficients obtained from the mixed-effects logistic model for each day of the week. It is clear from Table 10 that *Saturday* and *Sunday* have an inverse relationship with *long* build durations, whereas the other weekdays have a direct relationship with *long* build durations. The estimated coefficient results implies that the odds of having *long* build durations on *Saturday* and *Sunday* is 36% and 40% lowers than the odds for *Friday* (*p-values* of $4.46e^{-14}$ and $1.23e^{-15}$, respectively). However, the odds of having *long* build durations on the other weekdays are 9 – 14% higher than the odds for *Friday* (*p-values* of 0.011-0.037). Such a finding suggests that the servers of Travis CI have a higher workload on weekdays. Existing research has also found an association between the *Day of week* and buggy code changes [19, 56]. Furthermore, builds are more likely to have longer durations when they are triggered during the day. Looking at the estimated coefficient (i.e., -0.14) of the *Day or night* independent variable, we observe that builds triggered at night have a significant inverse relationship with *long* build durations (*p-value* $= 4.01e^{-07}$). This suggests that the odds for having *long* durations of builds triggered at night is 13% lower than the odds for triggering builds during the day. Hence, builds are most likely to run faster because Travis CI’s servers have lower workloads at night.

5 Discussion

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

5.1 Developers

Developers should consider optimizing their core tests in addition to the removal of unnecessary tests. Developers acknowledge that tests are significantly associated with *long* build durations [28]. Much of developers’ effort is usually invested to identify and remove unnecessary tests. However, developers should consider that the important software tests cannot be ignored. For important tests, the performance of test cases may be improved by reducing brittle assertions (i.e., assertions that depend on uncontrolled inputs) and unused inputs (i.e., inputs controlled by a test but not checked by an assertion) [32]. Tests can be optimized using a proper management of test dependencies [60]. Employing test case minimization techniques can improve the efficiency of software testing while

²² <https://travis-ci.org/jruby/jruby/builds/108161963>

²³ <https://travis-ci.org/jruby/jruby/builds/108165671>

²⁴ <https://github.com/jruby/jruby/commit/30d975e6abdb1bdab1b80b0bfbd83313f139f8a2>

maintaining an effective coverage [37,38]. As a result, such techniques may help to reduce build durations. There exists a rule of thumb of restricting a test case to only one assertion [4]. Employing a single test assertion per test case improves fault localization [67] and test readability [43], but may have an association with *long* build durations. If the build duration is a very important factor to a development team, sacrificing the test readability factor might be a wiser choice. For example, developers may combine several test cases into a single comprehensive test case if such tests share similar characteristics. Developers may use proper explanatory messages for test assertions to distinguish the test assertions of each functionality in the case of test failures. Nevertheless, developers should take into consideration that removing a few tests in a commit (or a push of commits) will less likely have an association with the build duration. For example, the commit level factors, such as the *Test churn*, *Tests added*, and *Tests deleted*, have a weak association with *long* build durations. Therefore, development teams should consider performing test optimization whenever they produce a new release of the project.

To gain more insights about how the test density may be associated with the *long* build duration, we perform a manual analysis for sample test cases of the **structr**²⁵ project. We find five test cases (i.e., test methods) that contain 33 – 60 test lines and 5 – 9 test asserts. Although each of such test cases targets a certain system functionality, we observe duplicate code and asserts between them. For example, we find a test case for the functionality of moving a file to an arbitrary directory and another test case for moving a file to the root directory. Having a separate test case for each functionality of the system helps to locate test failures. However, if reducing the build duration is more important, refactoring the test code (e.g., by resource inlining or reducing the test data) [43,60] would be a wiser option.

Not all build jobs are parallelized. Developers should realize that, for free subscriptions of Travis CI, there is a limit of 5 jobs to run simultaneously. If a build has more than five jobs, only five of them would be running in parallel. Once one of the jobs finishes, another job can start running along with the four running jobs, and so on. Development teams can maintain paid subscriptions depending on how many concurrent jobs are needed to run. On the other hand, build jobs may be configured to run in stages. A build stage may contain a set of jobs that can run in parallel. Jobs of a build stage do not run in parallel with jobs of other build stages. Instead, jobs of next build stages wait for the jobs of previous build stages to finish. Hence, maintaining build stages indicates that, even though parallelizing jobs significantly helps to reduce build duration, the number of jobs still matters when it comes to the build duration. Future research should study how open source projects (i.e., free CI subscribers) may become more costly than projects with a paid service based on the gain in terms of build durations.

5.2 Researchers

Test optimization and prioritization may be useful to reduce the build duration. Researchers should explore ways to identify tests that may perform similarly (i.e., semantic test clones) in order to potentially reduce build durations. Developers add more tests whenever a new system functionality is introduced to

²⁵ <https://github.com/structr/structr>

the project. Hence, due to the frequent additions of tests, developers may neglect writing efficient test cases. In addition, due to parallel development activities, it could be hard for developers to identify whether a test is a duplicate of another existing test. Therefore, researchers may explore ways to prioritize software tests from a CI perspective [18,40].

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

5.3 Tool builders

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

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

5.4 CI services

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

CI services should utilize the current behavior of builds to suggest possible build (re)configurations. Existing research shows that development teams misuse CI configurations [25]. Misusing a CI configuration may unintentionally be associated with *long* build durations. For example, developers may configure builds to update dependencies in every run to avoid breaking the build in the case of unexpected dependency updates. Therefore, it is better for services CI to optionally perform such an update only if a dependency is recognized to be not up to date. It is also better to send feedback to development teams about the possible (re)configuration performed on their builds. The feedback may also incorporate information about the reasons why recently triggered builds have longer build durations than the previously triggered builds.

6 Threats to Validity

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

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

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

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

6.3 External Validity

External threats are concerned with our ability to generalize our results. Our study is based on builds that are collected from a set of 67 projects. Therefore, we cannot generalize our conclusions to other projects with different characteristics. Nevertheless, our study selects projects with high variations of build durations where the problem of *long* build durations may occur. To this end, we select the projects that have a build durations MAD above 10 minutes [11, 28]. However, despite the relatively small sample, our dataset contains well-known and previously studied projects (e.g., **rails**, **jruby**, and **openproject**) Projects with lower build durations MADs (i.e., less than 10 minutes) are less likely to suffer from *long* build durations. Still, future work should investigate whether lower MADs of build durations would produce different results as compared to our findings. Moreover, a replication of our work using projects written in other programming languages is required to reach more general conclusions.

7 Related Work

This section presents the related research about CI builds while highlighting the contributions of our work.

7.1 Studying CI build status

Existing research has investigated the reasons behind CI build failures [7, 68, 51, 64]. A study by Beller et al. [7] shows that tests are central to the CI build process, since they highly impact the build duration and the build status. Another study by Zolfagharinia et al. [68] indicates that build status could be impacted by operating systems or runtime environments in which the code should be integrated. Rausch et al. [51] studied the causes of build errors and failures and found 14 common error categories of CI builds. They also found that process factors (e.g., the complexity of changes) have a strong impact on build statuses, in addition to the history build stability (i.e., failing ratio). Another study by Vassallo et al. [64] derived a taxonomy of build failures and found that, although open source and industrial projects have differences in their design and their ways for reporting build failures, they share common failing patterns. None of such studies investigated how build status is associated with *long* build durations. In our study, we analyze build durations per build statuses to investigate whether *long* build durations are associated with build statuses. We also study the most important factors to model *long* build durations.

Other studies introduced prediction models to predict the build status [66, 47]. Xia and Li [66] built 9 prediction models to predict the build status. They validated their models using a cross-validation scenario and an online scenario. Their models achieved a prediction AUC of over 0.80 for the majority of the projects they studied. They found that predicting a build status using the online scenario performed worse than that of cross-validation, with a mean AUC difference of 0.19. They observed that the prediction accuracy falls down due to the frequent changes of project characteristics, development phases, and build characteristics across different version control branches. Ni and Li et al. [47] used cascade classifiers to predict build failures. Their classifiers achieve an AUC of 0.75, which outperforms other classifiers, such as decision trees and Naive Bayes. They also observed that historical committer and project statistics are the best indicators of build status. In our study, we use mixed-effects logistic models to study the association of various factors with *long* build durations across 67 projects. We fit the models to *long* build durations instead of build statuses. We analyze the explanatory power of the most important factors to explain the *long* build durations.

7.2 Studying CI build duration

Studies in the literature have investigated build durations from different perspectives [52, 50, 3]. A study by Rogers [52] observes that a *long* build duration considerably interrupts the development process. The author suggests strategies to keep complex projects coping with CI, such as accumulating the integration to be performed once a week. It is also argued that every project should maintain a certain limit of build durations and keep up with that. Similarly, Rasmusson [50] studied how feedback and team spirits are negatively impacted when a build takes a *long* duration. Another study proposed an approach to split large builds into smaller

builds and adopt the so-called *incremental builds* [3]. Incremental builds enhanced the overall build duration. However, none of such studies have investigated the factors that may be associated with *long* build durations. In our work, we study *long* durations of CI builds. Instead of studying how *long* durations of CI builds impact the development process, we gain insights on how developers can optimize their code or configure their builds to reduce the *long* build durations.

Brooks [11] highlighted that *long* build durations affect the development flow and lead to changing the frequency and size of commits. According to reports of expert developers of a private company, a build duration of 2 minutes is considered optimal, while is acceptable to take up to 10 minutes. Nonetheless, we cannot generalize such an assumption, since builds of complex software systems (e.g., Linux) may definitely require more time to process. On the other hand, building and testing tools may impact the duration of builds, since they require running internal libraries or other dependent tools [45]. Additionally, users have a limited control over such tools. All such studies lack an empirical evidence of the frequency of builds with *long* durations in software projects and the factors that may be associated with such a latency. In our work, we empirically study the factors that have a strong association with *long* build durations by employing mixed-effects logistic models.

The work by Bisong et al. [10] is relevant to our work in the sense that it investigates the duration of CI build generation. We summarize the distinctions between our study and the study conducted by Bisong et al. [10] in the following:

- Bisong et al. evaluated 13 prediction models for build durations and reported which models are best to estimate build durations using a dataset of 10,000 builds collected from different projects in TravisTorrent. In our study, we model *long* build durations using a mixed-effects logistic model, which take into consideration the variance in build duration across projects.
- The models presented by Bisong et al. were not entirely suitable for modeling build durations, since post-build factors were used in their models (e.g., the number of tests runs, test duration, and CI latency). In our study, we exclude such kind of factors from our models to make them suitable to be used for prediction. We also compute more factors, in addition to TravisTorrent factors, to study their relationship with *long* build durations.
- The models presented by Bisong et al. use the build duration factor introduced by TravisTorrent as a dependent variable. However, such duration values are misleading, since they do not represent the perceived build duration but rather the summation of the durations of all build jobs. In our study, we compute the perceived build durations using the build starting time and build finishing time provided by Travis API.
- Bisong et al. reported the results of the top-performing models in estimating build duration. However, there was no analysis about the factor that may be associated with *long* build durations. In our study, we leverage the results of the mixed-effects model to analyze the most important factors to model *long* build durations. Furthermore, we perform manual analyses to gain insights about the relationship between long build durations and the most important factors of our model.

7.3 Other studies about CI builds

A study by Atchison et al. [5] performed a time-series analysis of the history of CI builds. They observed a steady growth in the number of builds over time. They also observed weekly and seasonal trends of how builds are generated by software projects. Their approach was able to estimate the number of builds to be generated in the future, with an average accuracy of 0.86. However, that study did not investigate how build durations evolve over time. In our study, we observe that build durations do not increase over time. We classify build durations into *short* and *long* and study the frequency of *long* build durations.

8 Conclusion

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

- About 40% of build durations take over 30 minutes to run.
- Build durations may increase or decrease over time, which indicates that there exist important factors that have a strong association with such a fluctuation.
- Durations of *passed* builds are not always longer than durations of *errored* or *failed* builds.
- Triggering CI builds during the day or on weekdays is most likely to be associated with *long* build durations.
- Short build durations are associated with builds that are configured (a) to cache content that does not change often or (b) to finish as soon as all the required jobs finish. However, misusing such configurations may not help to reduce the build duration.
- There is a tradeoff between maintaining stable build statuses and long build durations. In particular, configuring builds to rerun failing commands multiple times has a strong association with *long* build durations with a negligible reduction in the build failure ratio.

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

References

1. Online Appendix. https://taher-ghaleb.github.io/papers/emse_2018/appendix.html
2. Agresti, A.: Tutorial on modeling ordered categorical response data. *Psychological bulletin* **105**(2), 290 (1989)
3. Ammons, G.: Grexmk: speeding up scripted builds. In: *Proceedings of the international workshop on Dynamic Systems Analysis*, pp. 81–87. ACM (2006)
4. Astels, D.: One Assertion Per Test. <http://www.artima.com/weblogs/viewpost.jsp?thread=35578>. Visited on February 05, 2018

5. Atchison, A., Berardi, C., Best, N., Stevens, E., Linstead, E.: A time series analysis of TravisTorrent builds: to everything there is a season. In: Proceedings of the 14th International Conference on Mining Software Repositories, pp. 463–466 (2017)
6. Beck, K.: Extreme programming explained: embrace change. addison-wesley professional (2000)
7. Beller, M., Gousios, G., Zaidman, A.: Oops, my tests broke the build: An explorative analysis of Travis CI with GitHub. In: Proceedings of the 14th International Conference on Mining Software Repositories, pp. 356–367 (2017)
8. Beller, M., Gousios, G., Zaidman, A.: Travistorrent: Synthesizing travis ci and github for full-stack research on continuous integration. In: Proceedings of the 14th International Conference on Mining Software Repositories, pp. 447–450 (2017)
9. Bernardo, J.H., da Costa, D.A., Kulesza, U.: Studying the impact of adopting continuous integration on the delivery time of pull requests. In: Proceedings of the 15th International Conference on Mining Software Repositories, pp. 131–141. ACM (2018)
10. Bisong, E., Tran, E., Baysal, O.: Built to last or built too fast?: evaluating prediction models for build times. In: Proceedings of the 14th International Conference on Mining Software Repositories, pp. 487–490 (2017)
11. Brooks, G.: Team pace keeping build times down. In: Proceedings of the AGILE Conference, pp. 294–297. IEEE (2008)
12. Cliff, N.: Dominance statistics: Ordinal analyses to answer ordinal questions. *Psychological Bulletin* **114**(3), 494 (1993)
13. Domingos, P.: A few useful things to know about machine learning. *Communications of the ACM* **55**(10), 78–87 (2012)
14. Dunn, O.J.: Multiple comparisons among means. *Journal of the American Statistical Association* **56**(293), 52–64 (1961)
15. Dunn, O.J.: Multiple comparisons using rank sums. *Technometrics* **6**(3), 241–252 (1964)
16. Duvall, P.M., Matyas, S., Glover, A.: Continuous integration: improving software quality and reducing risk. Pearson Education (2007)
17. Easterbrook, S., Singer, J., Storey, M.A., Damian, D.: Selecting empirical methods for software engineering research. In: Guide to advanced empirical software engineering, pp. 285–311. Springer (2008)
18. Elbaum, S., Rothermel, G., Penix, J.: Techniques for improving regression testing in continuous integration development environments. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 235–245. ACM (2014)
19. Eyolfson, J., Tan, L., Lam, P.: Do time of day and developer experience affect commit bug-giness? In: Proceedings of the 8th Working Conference on Mining Software Repositories, pp. 153–162. ACM (2011)
20. Faraway, J.J.: Extending the linear model with R: generalized linear, mixed effects and nonparametric regression models, vol. 124. CRC press (2016)
21. Feldman, S.I.: MakeA program for maintaining computer programs. *Software: Practice and experience* **9**(4), 255–265 (1979)
22. Fisher, R.A.: Statistical methods for research workers. Genesis Publishing Pvt Ltd (1925)
23. Fowler, M., Foemmel, M.: Continuous integration. http://www.dccia.ua.es/dccia/inf/asignaturas/MADS/2013-14/lecturas/10_Fowler_Continuous_Integration.pdf (2006)
24. Gallaba, K., Macho, C., Pinzger, M., McIntosh, S.: Noise and heterogeneity in historical build data: an empirical study of travis ci. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, pp. 87–97. ACM (2018)
25. Gallaba, K., McIntosh, S.: Use and Misuse of Continuous Integration Features: An Empirical Study of Projects that (mis) use Travis CI. *IEEE Transactions on Software Engineering* (2018)
26. Hanley, J.A., McNeil, B.J.: The meaning and use of the area under a receiver operating characteristic (ROC) curve. *Radiology* **143**(1), 29–36 (1982)
27. Harrell, F.E.: Regression modeling strategies, with applications to linear models, survival analysis and logistic regression. GET ADDRESS: Springer (2001)
28. Hilton, M., Nelson, N., Tunnell, T., Marinov, D., Dig, D.: Trade-offs in continuous integration: assurance, security, and flexibility. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, pp. 197–207. ACM (2017)
29. Hilton, M., Tunnell, T., Huang, K., Marinov, D., Dig, D.: Usage, costs, and benefits of continuous integration in open-source projects. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, pp. 426–437. ACM (2016)

30. Holm, S.: A simple sequentially rejective multiple test procedure. *Scandinavian journal of statistics* pp. 65–70 (1979)
31. Howell, D.C.: Median absolute deviation. *Wiley StatsRef: Statistics Reference Online* (2014)
32. Huo, C., Clause, J.: Improving oracle quality by detecting brittle assertions and unused inputs in tests. In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 621–631. ACM (2014)
33. Kampstra, P., et al.: Beanplot: A boxplot alternative for visual comparison of distributions. *Journal of Statistical Software* **28** (2008)
34. Kruskal, W.H., Wallis, W.A.: Use of ranks in one-criterion variance analysis. *Journal of the American statistical Association* **47**(260), 583–621 (1952)
35. Kumfert, G., Epperly, T.: Software in the DOE: The Hidden Overhead of “The Build”. Tech. rep., Lawrence Livermore National Lab., CA (US) (2002)
36. Laukkanen, E., Mäntylä, M.V.: Build waiting time in continuous integration: an initial interdisciplinary literature review. In: *Proceedings of the Second International Workshop on Rapid Continuous Software Engineering*, pp. 1–4 (2015)
37. Lei, Y., Andrews, J.H.: Minimization of randomized unit test cases. In: *Software Reliability Engineering, 2005. ISSRE 2005. 16th IEEE International Symposium on*, pp. 10–pp. IEEE (2005)
38. Leitner, A., Oriol, M., Zeller, A., Ciupa, I., Meyer, B.: Efficient unit test case minimization. In: *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pp. 417–420. ACM (2007)
39. Lewis, A.J.: *Mixed effects models and extensions in ecology with R*. Springer (2009)
40. Liang, J., Elbaum, S., Rothermel, G.: Redefining prioritization: continuous prioritization for continuous integration. In: *Proceedings of the 40th International Conference on Software Engineering*, pp. 688–698. ACM (2018)
41. McIntosh, S., Adams, B., Hassan, A.E.: The evolution of Java build systems. *Empirical Software Engineering* **17**(4-5), 578–608 (2012)
42. McIntosh, S., Nagappan, M., Adams, B., Mockus, A., Hassan, A.E.: A large-scale empirical study of the relationship between build technology and build maintenance. *Empirical Software Engineering* **20**(6), 1587–1633 (2015)
43. Meszaros, G.: *xUnit test patterns: Refactoring test code*. Pearson Education (2007)
44. Meyer, M.: Continuous integration and its tools. *IEEE software* **31**(3), 14–16 (2014)
45. Mokhov, A., Mitchell, N., Peyton Jones, S., Marlow, S.: Non-recursive make considered harmful: Build systems at scale. In: *Proceedings of the 9th International Symposium on Haskell*, pp. 170–181. ACM (2016)
46. Nakagawa, S., Schielzeth, H.: A general and simple method for obtaining R² from generalized linear mixed-effects models. *Methods in Ecology and Evolution* **4**(2), 133–142 (2013)
47. Ni, A., Li, M.: Cost-effective build outcome prediction using cascaded classifiers. In: *Proceedings of the 14th International Conference on Mining Software Repositories*, pp. 455–458 (2017)
48. Peduzzi, P., Concato, J., Kemper, E., Holford, T.R., Feinstein, A.R.: A simulation study of the number of events per variable in logistic regression analysis. *Journal of clinical epidemiology* **49**(12), 1373–1379 (1996)
49. Pinheiro, P.: Linear and nonlinear mixed effects models. *r package version 3.1-97*. <http://cran.r-project.org/web/packages/nlme> (2010)
50. Rasmusson, J.: Long build trouble shooting guide. *Proceedings of the Extreme Programming and Agile Methods-XP/Agile Universe Conference* pp. 557–574 (2004)
51. Rausch, T., Hummer, W., Leitner, P., Schulte, S.: 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*, pp. 345–355 (2017)
52. Rogers, R.O.: Scaling continuous integration. In: *Proceedings of the International Conference on Extreme Programming and Agile Processes in Software Engineering*, pp. 68–76. Springer (2004)
53. Romano, J., Kromrey, J., Coraggio, J., Skowronek, J.: Should we really be using t-test and cohen’sd for evaluating group differences on the nsse and other surveys. In: *Annual meeting of the Florida association of institutional research* (2006)
54. Sarle, W.: The VARCLUS Procedure. *SAS/STAT User’s Guide* (1990)
55. Seo, H., Sadowski, C., Elbaum, S., Aftandilian, E., Bowdidge, R.: Programmers’ build errors: a case study (at google). In: *Proceedings of the 36th International Conference on Software Engineering*, pp. 724–734. ACM (2014)

56. Śliwerski, J., Zimmermann, T., Zeller, A.: When do changes induce fixes? In: ACM sigsoft software engineering notes, vol. 30(4), pp. 1–5. ACM (2005)
57. Smith, P.: Software build systems: principles and experience. Addison-Wesley Professional (2011)
58. Sulír, M., Porubán, J.: A quantitative study of Java software buildability. In: Proceedings of the 7th International Workshop on Evaluation and Usability of Programming Languages and Tools, pp. 17–25. ACM (2016)
59. Tufano, M., Palomba, F., Bavota, G., Di Penta, M., Oliveto, R., De Lucia, A., Shybyanyk, D.: There and back again: Can you compile that snapshot? *Journal of Software: Evolution and Process* **29**(4) (2017)
60. Van Deursen, A., Moonen, L., van den Bergh, A., Kok, G.: Refactoring test code. In: Proceedings of the 2nd international conference on extreme programming and flexible processes in software engineering (XP2001), pp. 92–95 (2001)
61. Vandekerckhove, J., Matzke, D., Wagenmakers, E.J.: Model comparison and the principle. In: The Oxford handbook of computational and mathematical psychology, vol. 300. Oxford Library of Psychology (2015)
62. Vasilescu, B., Van Schuylenburg, S., Wulms, J., Serebrenik, A., van den Brand, M.G.: Continuous integration in a social-coding world: Empirical evidence from github. In: Proceedings of the International Conference on Software Maintenance and Evolution (ICSME 2014), pp. 401–405. IEEE (2014)
63. Vasilescu, B., Yu, Y., Wang, H., Devanbu, P., Filkov, V.: Quality and productivity outcomes relating to continuous integration in GitHub. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, pp. 805–816. ACM (2015)
64. Vassallo, C., Schermann, G., Zampetti, F., Romano, D., Leitner, P., Zaidman, A., Di Penta, M., Panichella, S.: A Tale of CI Build Failures: an Open Source and a Financial Organization Perspective. In: Proceedings of the 33rd International Conference on Software Maintenance and Evolution (2017)
65. Winter, B.: A very basic tutorial for performing linear mixed effects analyses. arXiv preprint arXiv:1308.5499 (2013)
66. Xia, J., Li, Y.: 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, pp. 311–315 (2017)
67. Xuan, J., Monperrus, M.: Test case purification for improving fault localization. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 52–63. ACM (2014)
68. Zolfagharinia, M., Adams, B., Guéhéneuc, Y.G.: 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, pp. 312–322 (2017)