Characterizing Timeout Builds in Continuous Integration

Nimmi Weeraddana, *Student Member, IEEE*, Mahmoud Alfadel, *Member, IEEE*, and Shane McIntosh, *Senior Member, IEEE*

Abstract—Compute resources that enable Continuous Integration (CI, i.e., the automatic build and test cycle applied to the change sets that development teams produce) are a shared commodity that organizations need to manage. To prevent (erroneous) builds from consuming a large amount of resources, CI service providers often impose a time limit. CI builds that exceed the time limit are automatically terminated. While imposing a time limit helps to prevent abuse of the service, builds that timeout (a) consume the maximum amount of resources that a CI service is willing to provide and (b) leave CI users without an indication of whether the change set will pass or fail the CI process. Therefore, understanding timeout builds and the factors that contribute to them is important for improving the stability and quality of a CI service. In this paper, we investigate the prevalence of timeout builds and the characteristics associated with them. By analyzing a curated dataset of 936 projects that adopt the CircleCI service and report at least one timeout build, we find that the median duration of a timeout build (19.7 minutes) is more than five times that of a build that produces a pass or fail result (3.4 minutes). To better understand the factors contributing to timeout builds, we model timeout builds using characteristics of project build history, build queued time, timeout tendency, size, and author experience based on data collected from 105,663 CI builds. Our model demonstrates a discriminatory power that vastly surpasses that of a random predictor (Area Under the Receiver Operating characteristic Curve, i.e., $AUROC$ = 0.939) and is highly stable in its performance ($AUROC$ optimism = 0.0001). Moreover, our model reveals that the build history and timeout tendency features are strong indicators of timeout builds, with the timeout status of the most recent build accounting for the largest proportion of the explanatory power. A longitudinal analysis of the incidences of timeout builds (i.e., a study conducted over a period of time) indicates that 64.03% of timeout builds occur consecutively. In such cases, it takes a median of 24 hours before a build that passes or fails occurs. Our results imply that CI providers should exploit build history to anticipate timeout builds.

Index Terms—Build Systems, Continuous Integration, Timeout Builds, Statistical Models

1 Introduction

Continuous Integration (CI) is a software development practice that involves frequently integrating code changes into a shared repository [10]. The primary objective of CI is to provide developers with prompt feedback, enabling them to check if their modifications integrate seamlessly with the existing codebase and the changes other team members have been developing concurrently [14]. Prior work shows that the adoption of CI is associated with improvements in developer productivity [29, 49] and software quality [30, 50, 55].

Using a CI service that is suboptimally configured can impede timely feedback and waste computational resources [15, 56, 59]. To reduce waste and limit abuse of resources,[1] CI service providers often impose a time limit for CI builds. For example, in the CircleCI[2] context (a popular CI service provider), a build "times out" if the CI process has not produced output for a set time limit (imposed by default or configured by project teams). If the limit is exceeded, the build is terminated. Configuring a time limit prevents erroneous builds from consuming an indefinite amount of organizational or team CI resources due to common mistakes, such as infinite loops or (temporary) unavailability of network resources.

A CI build that times out does not deliver a definite signal to developers about the change set under scrutiny (i.e., do the changes integrate safely or not?) [14]. If timeout builds occur frequently, developers may not be able to progress on their work. One such example is shown in the Homebrew project,[3] where developers discussed the issue of observing frequent timeout builds and its consequence: several pull requests (PRs) being stalled in the backlog to be reviewed and/or integrated.

CI builds that time out also burden CI service providers with jobs that occupy an allocation of computational resources for an extended period of time without yielding meaningful outcomes for CI consumers [14]. To compensate for these wasteful jobs, CI service providers may need to over-allocate computational resources to their production environments in order to maintain their quality of service. This, in turn, increases the operational cost and introduces maintenance complexities.

Thus, our goal in this study is to investigate the prevalence and characteristics of timeout builds. In the initial stage of our study, we analyze 936 GitHub projects that reported at least one timeout build. We find that timeout builds last for a median of 19.7 minutes, which is more

• *N. Weeraddana, M. Alfadel, and S. McIntosh are with the Cheriton School of Computer Science, University of Waterloo, Canada.*
E-mail: {nrweeraddana, malfadel, shane.mcintosh}@uwaterloo.ca

Manuscript received date; revised date.

[1]https://support.circleci.com/hc/en-us/articles/360045268074
[2]https://circleci.com
[3]https://github.com/Homebrew/discussions/discussions/4075

than five times longer than the median duration of signal-generating builds [14] (i.e., builds that have either a fail or pass outcome).

To investigate the factors that characterize the outcome of CI builds, we fit and analyze regression models. For this analysis, we extract five families of features from a dataset of 105,663 CI builds that span 24 projects. Among these builds, 1,301 timed out. Below, we present our research questions and a preview of the corresponding answers:

(RQ1) How well can our models explain the incidences of timeout builds? To evaluate whether and how well our model distinguishes builds that will time out from those that will not, we estimate the *discriminatory power* of the model using the Area Under the (Receiver Operating characteristic) Curve ($AUROC$) [21], its *calibration* using the Brier score [4], and its ability to balance the precision-recall tradeoff using the Area Under Precision-Recall Curve ($AUPRC$) [46]. We assess the stability of these fitness scores using a bootstrap-derived optimism penalty [11]. Our model achieves an $AUROC$ of 0.939—a discriminatory power that vastly surpasses that of naïve baselines, such as random guessing ($AUROC$ of 0.5). Moreover, the model achieves a Brier score of 0.008—a calibration score that suggests the risk estimates of the model are highly reliable. In terms of stability, our model has only a small optimism penalty of less than one percentage point in terms of both $AUROC$ and Brier score, suggesting that the model is unlikely to be overfitted to the data on which it was trained. Lastly, in terms of the balance between precision and recall, our model's $AUPRC$ of 0.319 outperforms the baseline performance ($AUPRC$ of 0.012).

(RQ2) What are the most influential features of our models of timeout builds? To assess the explanatory power of each (family of) feature(s), we perform Wald χ^2 (a.k.a., "chunk") tests [43]. We find that timeout builds are best characterized by the family of build history features, which contains the status of recent builds, their durations, and the proportion of the builds that have previously timed out. Indeed, build history features contribute to more than half (70%) of the explanatory power of the model. Furthermore, our model reveals that timeout tendency features (e.g., file tendency) are among the most powerful features to influence timeout builds, suggesting that change sets that include specific files or components are more often implicated in CI builds that time out than others. We also reveal that a non-negligible proportion (15%) of files more often appear in the change sets of timeout builds than signal-generating builds.

Inspired by the results of our model, we perform a longitudinal analysis of the occurrences of timeout builds. We find that the majority (64.03%) of the timeout builds occur in clusters (i.e., timeout builds tend to occur consecutively). A considerable proportion (20%) of these clusters comprise at least six builds. We also find that once a cluster of timeout builds is observed, it takes a median of 24 hours before a signal-generating build occurs.

To further understand the root causes of build timeouts, we conduct a thematic analysis of 79 GitHub discussions, encompassing 406 comments. This analysis uncover six primary causes for CI timeouts. Among these, the most frequent reason is inefficiencies in testing processes.

Our results highlight the importance of considering the historical context of builds for CI service providers and consumers to anticipate timeout builds. For example, CI providers could target efforts to enhance their infrastructure, optimize resource allocation, and fine-tune their systems to cater to specific demands of individual software projects, reducing the occurrence of timeouts. CI consumers can also leverage this knowledge to prioritize their attention to recent builds and their associated characteristics. Moreover, developers may identify timeout-prone files and focus their efforts on those files when troubleshooting timeouts. Finally, our longitudinal analysis suggests the existence of underlying systemic issues that contribute to prolonged periods of consecutive builds that time out, and our analysis reveals a catalog of potential root causes for timeouts. Developers can investigate timeout clusters to identify and address the root causes, leading to more stable CI processes.

2 Continuous Integration (CI) and Timeouts

This section provides a background about CI and timeouts.

CI Configurations. The instructions for executing a CI build (e.g., the machine specification, the order of test execution, etc.) are detailed in a configuration file. An illustrative example is CircleCI, where these settings are delineated in a `.circleci/config.yml` file located at the root of the project's repository.[4] This file contains all the necessary details to guide the CircleCI system on how to execute the builds, including the machine specifications, the order in which tests should be run, and other relevant build parameters.

CI Build Outcomes. CI builds are executed based on the specified configurations with the goal of providing clear feedback on the recent code changes. Nevertheless, the outcomes of these builds are not always straightforward. Our prior work classifies them into two categories [14]: (1) signal-generating builds, which executes until completion, resulting in either a pass or fail signal. A passing build indicates to the developers that the code changes have met the minimum required checks, while a failing build alerts users about the issues in the code changes [14]; (2) non-signal-generating builds, which terminates before completion due to user-initiated aborts, configuration errors, and infrastructure provisioning issues (e.g., timeouts). These builds do not provide developers with a meaningful signal about their code changes [14].

CI Logs. CI build outcomes are meticulously recorded in build logs, which serve as comprehensive records of the build process. These logs not only capture the outcomes but also provide crucial data for analysis and troubleshooting. For example, in the context of CircleCI, the build outcomes and their detailed logs are conveniently stored within a specific section of the CircleCI interface.[5] This is typically accessible through the project's dashboard on the CircleCI website, allowing users to easily review and analyze the results of each build. This accessibility plays a significant role in the maintenance and enhancement of the software development process.

[4]https://circleci.com/docs/configuration-reference/
[5]https://circleci.com/docs/audit-logs/

CI Timeout Builds and Time Limits Adjustment. CI timeout builds represent a distinct subset of non-signal-generating builds [14]. These occur when a build surpasses its allotted time frame, hinting at possible code inefficiencies like performance bottlenecks.[6] A few other studies [5, 9, 14, 58] have also considered timeouts as a special type of build outcome. Our study adds to prior work by explicitly focusing on characterizing timeouts.

Addressing such CI timeout builds requires adjusting time limits for timing out. In CircleCI, the default time limit for timing out is ten minutes,[7] but a developer can change this default limit. This is done by adding the `no_output_timeout` setting to the CI configurations in the `.circleci/config.yml` file. For instance, to set a 30-minute time limit, the `no_output_timeout` setting should be set to `30m`. Doing so ensures that if the job produces no output for 30 minutes, CircleCI will automatically terminate the job. This feature is useful for managing resource usage and detecting jobs that are stuck or taking unusually long to complete. Properly setting these time constraints is crucial to prevent extended, resource-intensive build processes.

An example of timeout builds is reported in an issue from the raft-tech/TANF-app project,[8] describing how timeout builds impact the project. The issue reports that 14 of the 33 nightly builds are timeout builds. To fix the issue, developers suggested increasing the `no_output_timeout`, which is a frequently recommended fix for CircleCI timeouts. This fix is not always optimal because each project using CircleCI is billed according to their use of "build-minutes" on the service. Increasing the time limit would increase service usage. Moreover, it is unclear how much the time limit should be increased to allow long-running commands to terminate. Another discussed option was to omit vulnerability scans from their CI process since those scans take a long time to complete and are silent during their execution, leading to the CircleCI timeouts. This is also suboptimal because the CI process of the project would no longer benefit from the early detection of vulnerabilities. The raft-tech organization is not the only one struggling with the timeout problem.

3 THE PREVALENCE OF CI TIMEOUT BUILDS

In this section, we report on an exploratory analysis of timeout builds in CircleCI. In particular, we analyze the frequency at which timeout builds occur (Section 3.1) and the quantity of waste that timeouts generate (Section 3.2).

3.1 The Frequency of Timeout Builds

To estimate how often timeout builds occur, we require a large and rich collection of build records from a realistic CI service. Since there has been an increase in the adoption of CI practices [19], a proliferation and a diversity of CI services exists in the market to cater increasing demands of collaborative software development and DevOps practices.

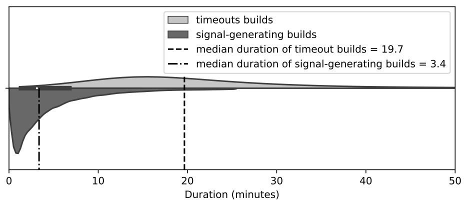


Fig. 1: The distributions of the duration of timeout and signal-generating builds.

As indicated in prior work [14], CircleCI is one of the (if not the) most popular CI service(s). In fact, CircleCI has 748k installations over an eight-year period [14]. Given its widespread usage and the advantages offered (e.g., support for multiple source code hosting services including GitHub and shareable packages of CI configurations) to consumers over other CI services,[9] we focus our analysis on CircleCI. For a detailed discussion of the advantages of CircleCI over other CI services, we encourage interested readers to read the comparison of CI services provided in our Online Appendix A. [10] We begin with a dataset curated by Gallaba et al. [14], which contains CircleCI builds of 7,795 open-source GitHub projects. This dataset contains the outcome (e.g., pass, failed, timeout, and canceled) for each of the CI builds that it contains. For our analysis, we select the projects that have at least one CI build with its outcome status being "timeout." Of the 7,795 projects in the dataset, we find that 936 (12%) projects contain at least one timeout build. Among these 936 projects, a median of four timeout builds is observed, suggesting that the distribution is skewed [41]. Indeed, we find that 10% of the projects account for 44 or more timeouts each, while an extreme 4% of the projects account for 100 or more timeouts each. Overall, this suggests that timeout builds are a relevant issue in the context of GitHub projects.

3.2 The Quantity of Timeout Waste

Fig. 1 shows the durations of signal-generating builds [14] (dark grey) and timeout builds (light grey) in the 936 projects that have at least one timeout. From the figure, we observe that a timeout lasts for a median of 19.7 minutes. This is almost fivefold longer than the median duration of signal-generating builds with fail or pass outcomes (3.4 minutes). In certain situations, the issue of timeouts is exacerbated. For example, in the Homebrew/linuxbrew-core project,[11] the median duration of a timeout build is 125.3 minutes, which is 21 times the median duration of a signal-generating build.

Furthermore, even though some projects are accountable for small proportions of timeouts in the dataset, a large quantity of waste is attributable to them. For example, coala/coala project[12] accounts for only 1.7% of the timeouts in the dataset, but it results in a total waste of 111 build hours. Besides, from the viewpoint of a CI provider, the impact is amplified; the higher the number of timeouts, the

[6]https://support.circleci.com/hc/en-us/articles/360007188574-Build-has-Hit-Timeout-Limit
[7]https://support.circleci.com/hc/en-us/articles/16616033407131-Max-Runtime-in-CircleCI-Server
[8]https://github.com/raft-tech/TANF-app/issues/1534
[9]https://circleci.com/docs/
[10]https://zenodo.org/records/10901318
[11]https://github.com/Homebrew/linuxbrew-core
[12]https://github.com/coala/coala

TABLE 1: Description and rationale of the selected features. "F." stands for family.

F.	Description/Rationale
Build history	**recent build status:** Whether the previous build timed out or not. Rationale: If a project has a recent history of timeout builds, the build is more likely to time out (inspired by the prior work [7, 44]).
	recent build duration: The duration of the prior build. Rationale: A project with a recent history of long build durations may be more likely to have the build take longer and be timed out (inspired by GitHub issues[13]).
	timeout ratio: The proportion of builds that timed out. Rationale: Inspired by studies that predict build outcomes [7], we expect greater timeout ratios to portend future timeouts.
When	**queued month, day, hour, and minute:** The moment when the build was queue for processing. Rationale: Build requests of particular times may be more/less prone to timeouts.
Size	**loc:** The number of lines of code within the files changed. Rationale: The size of the files changed may have an impact on the likelihood of the build to time out.
	insertions, deletions, files: The sum of inserted/deleted lines of code and the number of unique files touched. Rationale: The size of the change corresponding to the build may have some relation to timeout builds (inspired by similar studies on build failure prediction [40, 44]).
Author exp.	**changes to related files/changes to any file:** The sum of prior changes by the authors to (a) the files changed; and (b) any file. **lines added to/deleted from related files/any file:** The sum of prior lines of code added/deleted by the author to (a) the files changes; and (b) any file. Rationale: The familiarity of developers with the overall codebase and specific areas may have an impact on the timeout builds [40].
Timeout tendency	**author tendency:** The number of prior timeout builds that contain commits by the authors. Rationale: If the author tendency of a build is high, it has a high chance of timing out [52].
	file tendency: The number of prior timeout builds that contain changes to the files. Rationale: We hypothesize that the higher the file tendency of a certain build, the higher the chance of that build timing out [52].

more resources the provider must allocate to maintain the quality of service in the presence of increased uncertainty.

4 Study Design

The analysis in Section 3 demonstrates that timeout builds account for a large quantity of CI waste and motivates us to study the characteristics of timeout builds. In this section, we describe our procedures for curating our dataset (DC) and fitting our models (MF) to characterize timeout builds. Fig. 2 shows an overview of our study design.

(DC) Data Curation

We begin with the Gallaba et al. [14] dataset of CirclCI builds which explicitly labels passing builds, failing builds as well as timeout builds that we need for our study. Fig. 2 (DC) shows an overview of the two-step data curation procedure. Below, we describe each step.

(DC-1) Select projects. It is important to highlight that a substantial portion of the timeout builds in the dataset are contributed by specific projects. Additionally, GitHub accommodates projects that are still not yet mature. Conducting a closer examination of mature and disproportionately affected projects would yield valuable insights regarding the workload for the CI provider and raise awareness among project maintainers. To ensure a focused analysis isolating the key factors contributing to these issues, we apply the following filtering criteria to the projects from the original dataset.

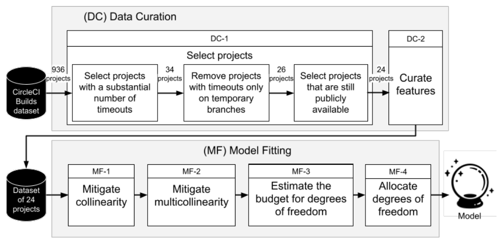

(DC) Data Curation
DC-1
Select projects
CircleCI Builds dataset
936 projects
Select projects with a substantial number of timeouts
34 projects
Remove projects with timeouts only on temporary branches
26 projects
Select projects that are still publicly available
24 projects
DC-2
Curate features
(MF) Model Fitting
Dataset of 24 projects
MF-1
Mitigate collinearity
MF-2
Mitigate multicollinearity
MF-3
Estimate the budget for degrees of freedom
MF-4
Allocate degrees of freedom
Model

Fig. 2: An overview of our study design.

- **Select projects with a substantial number of timeout builds.** Our initial dataset contains 936 projects that suffer from at least one timeout. To conduct our study on a set of projects that suffer from a meaningful number of timeouts, we sort the projects in descending order of the timeout frequency. Note that our study requires in-depth data extraction (e.g., collecting the commit and file records for each project). Therefore, to make the study feasible, we consider a subset of projects that account for more than half of all timeout builds collected from the entire dataset. This step yields 34 projects covering 54% of all timeout builds collected from the entire dataset. The selected projects have a rich development history with a median of 3,966 commits made by a median of 82 contributors. Moreover, the dataset includes popular projects, such as the palantir/atlasdb[14] project.
- **Remove projects having timeout builds only on temporary branches.** For each surviving project, we select the timeout builds that appear on its main/master branch. Doing so ensures that all selected commits corresponding to the builds are still accessible, mitigating inconsistencies caused by the removal of temporary branches. We find eight of the selected projects exclusively contain timeout builds on temporary branches, and we filter these projects out of further analysis. Thus, we find 26 projects that survive this step.
- **Select projects that are publicly available.** We need to extract project-related data, such as commits and lines of code. We select projects that are publicly available when conducting this work (22nd November 2022). We use the GitHub API to check the accessibility of projects. Doing so excludes two projects, leaving us with 24 projects eligible for our analysis.

Our final dataset consists of 105,663 CI builds spanning 24 projects. Of these builds, 1,301 are timeouts, while the remaining builds generate a pass/fail signal.

Note that our dataset is imbalanced in terms of the two classes — a common phenomenon in software engineering research [3]. Class balancing is particularly important in

[14]https://github.com/palantir/atlasdb

scenarios where the minority class is of greater interest than the majority, such as in fraud detection or rare disease diagnosis [28]. Class balancing tends to improve the recall (by fitting the model in an environment where the minority class is more prevalent than it actually is, the model is more likely to raise timeout alerts and catch timeout examples), but at the cost of precision (since the model is prone to raising plenty of timeout warnings, in real-world scenarios, where timeouts are rare, the model is likely to raise plenty of false alarms). In our case, even though the minority class (timeouts) is of great importance, we do not want to inflate the false positive rate. We are interested in modelling the characteristics of builds that are most likely timeouts and not the ones that have a little chance of being a timeout. Thus, leaving the classes imbalanced in our dataset creates a model that is reflective of real-world scenarios. That said, to see if there is a substantial change in the results after balancing the classes, we rerun our experiments separately in the following class balancing scenarios: (1) Oversampling with the Synthetic Minority Over-sampling TEchnique (SMOTE) [6], (2) Undersampling with the Random method [26, 42], and (3) Combining both SMOTE oversampling ans random undersampling. We do not observe substantial changes in our results. Thus, we use the dataset without any class balancing in the rest of this study. A detailed description of this analysis is available in our Online Appendix B.[10]

Furthermore, we find that the percentage of timeouts varies from one project to another. For example, the docker-atlassian-confluence/cptactionhank project accounts for 20.8% of the timeouts in our sample, whereas the median proportion across the 24 projects is 2.3%. To account for the impact of any bias introduced by this project, we rebuild our statistical model (Section 4 MF). We do not observe any substantial differences in the model fits. Interested readers can refer to our Online Appendices C and D[10] for more details.

(DC-2) Curate features. To determine a set of features that characterize timeout builds, we consult the related literature in the areas of build outcome prediction [7, 40, 44] and defect prediction [52]. Table 1 shows the initial list of 19 features that span five properties of a build, along with the rationale for each feature's impact in the context of build outcomes. We use Gallaba et al.'s dataset [14] and Git commit logs to extract these features; we describe this in detail below.

Features extracted using Gallaba et al.'s dataset [14]. For each build in our dataset, we extract build-history features, i.e., the most recent build outcome, the most recent build duration, and the timeout ratio, i.e., the ratio of the total number of timeout builds in a project to the total number of builds of the project that triggered before the build under analysis. We also extract queuing-related features, such as the build queued time (the month, day, and hour, and minute). We also extract features concerning the tendency of builds to timeout [52], such as the number of previous timeout builds involving commits by the same authors (author tendency) and the number of prior timeout builds with changes to the same files (file tendency). Build history and tendency features may not be precise at the beginning but will adapt to more accurately reflect values over time.

Features extracted from commit logs. To get the change set size-related data [40], we analyze the commit log of each project, and extract the unique number of files added, modified, or deleted, as well as the number of lines inserted or deleted in the commits associated with each build. We also extract the features that estimate the project-specific experience of the authors of the commits [40]. For instance, we derive features like the total number of prior commits made by the commit authors associated with each build.

(MF) Model Fitting

Our goal is to study what characterizes timeout builds. Therefore, we select a statistical regression modeling procedure, which, unlike other classification techniques, emphasizes interpretability. In fact, statistical models like logistic regression provide clear insights into how different factors influence outcomes, making them ideal for nuanced analysis.[15] Thus, we fit logistic regression models using the approach recommended by Harrell Jr. [24]. This approach relaxes linearity assumptions using restricted cubic splines to allow features to share complex relations with the outcome (i.e., the likelihood of inducing a timeout in our case). Fig. 2 (MF) provides an overview of the steps we follow.

(MF-1) Mitigate collinearity. Collinear features distort each others' importance in the model [38, 39]. Thus, we first check for collinearity among our features using Spearman's ρ rank correlation [48]. We choose a rank correlation instead of other types of correlation measures (e.g., Pearson) because the rank correlation can detect nonlinear correlations. Similar to prior work [17, 20, 51], we use $\rho = 0.7$ as our threshold, i.e., any pair of features with $\rho > 0.7$ should have one of the features removed prior to model interpretation.

The hierarchical overview of the correlations among the features is shown in Fig. E.1 in our Online Appendix E.[10] Selecting one feature from the pairs of features that have $\rho > 0.7$ eliminates the following six features: loc, files, lines added to the related files, lines added to any file, lines deleted from the related files, and lines deleted from any file. We provide details on reasons for choosing one over the other in our Online Appendix E.[10]

(MF-2) Mitigate multicollinearity. We perform a redundancy analysis on the surviving 13 non-correlated features to mitigate multicollinearity that can introduce noise in model interpretation. A feature may introduce multicollinearity if it can be modelled using the other features. We eliminate such redundant features using the `redun` function in `R`, which fits a set of 13 models that each explain one feature using the 12 other features. Features having model fits that exceed the threshold ($R^2 > 0.9$) are recommended for exclusion [21]. Applying `redun` to our set of features did not identify any additional features for exclusion. Note that reducing collinearity and multicollinearity helps to identify the independent contributions of each feature to the outcome [34].

(MF-3) Estimate the budget for Degrees of Freedom (DoF) [11, 22]. All of the features are allocated at least one DoF in our fit. A feature that is allocated a single DoF can only capture monotonic and linear relationships with the

[15]https://www.fharrell.com/post/stat-ml/

likelihood of a CI build timing out. Allocating additional DoF to features allows our model to capture nonmonotonic and nonlinear relationships with the likelihood of a CI build timing out [12]. On the other hand, spending additional DoF to fit our model increases its risk of overfitting (i.e., being too specifically tuned to the training data to apply to testing examples). This tradeoff between model expressiveness and the risk of overfitting is often balanced by respecting a DoF budget [23]. Following prior work [24, 25], the DoF budget for a logistic regression model can be estimated as $\frac{n}{15}$, where n is the number of records in the minority class. Thus, the DoF budget for our model is $\frac{1{,}301}{15} = 86$.

(MF-4) Allocate DoF. To expend our budget prudently, we assign more DoF to features that are most likely to have a nonmonotonic relationship with CI timeouts, as determined by Spearman's multiple ρ^2. Fig. F.1 in our Online Appendix F[10] shows the ρ^2 value for each feature. From the figure, we observe that the recent build status, timeout ratio, author tendency, file tendency, and recent build duration have higher ρ^2 values than the other features. Thus, we allocate three DoF for those features except for the status of the most recent build since it is a binary feature.

Finally, we fit our regression model to our data, applying restricted cubic splines [23] to the features with additional DoF. These splines smooth transitions between direction changes using cubic fits, while allowing tail regions to retain more linear (straight) shapes. We make our dataset and the replication package available online.[10]

5 Study Results

In this section, after an initial analysis of the fitness of our model (Section 5.1), we characterize timeout builds by analyzing the importance of the features (Section 5.2).

5.1 (RQ1) How well can our models explain the incidences of timeout builds?

Approach. We evaluate the fitness of our model according to (a) its discriminatory power, (b) the calibration of its risk estimates, (c) the stability of the fit, and (d) the ability to balance precision and recall particularly in datasets with imbalanced classes like ours. We estimate the discriminatory power of our model using the *Area Under Receiver Operating Characteristic Curve* ($AUROC$) [21], which plots the true positive rate against the false positive rate; $AUROC$ values of 0, 0.5, and 1 represent the worst discrimination, random guessing, and perfect discrimination, respectively. We estimate the calibration of the risk estimates that are produced by our model using the *Brier Score* [4], i.e., the mean squared error of the predicted probabilities. A Brier score of 0 indicates the perfect calibration, whereas a score of 1 indicates the worst. Finally, we estimate the stability of our model fitness using the bootstrap-calculated optimism [11]. We begin by obtaining a sample from our dataset using bootstrap sampling. Then, we refit our logistic regression model to this bootstrap sample with the same allocation of degrees of freedom used in the original dataset. Next, we calculate the $AUROC$ and Brier score of this bootstrap model when (a) reapplied to the bootstrap sample on which it was trained and (b) on the original sample. After that, we estimate the $AUROC$ optimism by subtracting the respective fitness measures in the bootstrap sample from that of the original data. We repeat this process for 1,000 bootstrap iterations, and report the mean optimism values. The closer the mean optimism values of these fitness scores are to zero, the greater the stability of the fit. Lastly, we calculate the precision and recall for various threshold values representing the likelihood of a CI build being classified as a "timeout build." Next, we compute the *Area Under Precision-Recall Curve* ($AUPRC$) to measure our model's ability to balance precision and recall across different probability thresholds in the context of our imbalanced-class dataset [46]. The $AUPRC$ is also a value between 0–1. Then, we compare our $AUPRC$ with a baseline approach. The baseline is determined by the positive class prevalence, i.e., $\frac{tp}{tp+tn}$ [46]. The baseline appropriate for a balanced class distribution is 0.5. However, for our dataset, the baseline is $\frac{1{,}301}{1{,}301+104{,}362} = 0.012$.

TABLE 2: Model fitness.

$AUROC$	Brier score	$AUPRC$
0.939	0.008	0.319

Results. Table 2 shows the results of our model's fitness, and we make the following observations based on the table.

Observation 1: Our model can discriminate between timeout and non-timeout builds effectively, with well-calibrated risk estimates. Our model achieves an $AUROC$ of 0.939, vastly surpassing the $AUROC$ of naïve baselines, such as random guessing ($AUROC$ of 0.5). Also, our model achieves a Brier score of 0.008—our model has a near-perfect calibration, and its risk estimates are highly reliable [4].

Observation 2: Our model is highly stable to bootstrap-simulated variability [24]. The mean optimism value for $AUROC$ measure is 0.0001, which is close to perfect optimism [24]. This shows that the $AUROC$ value calculated using the bootstrap samples and the $AUROC$ value computed using the original dataset are not substantially different. Similarly, the mean optimism value for the Brier score measure is -0.0002. Such small optimism penalties below 1% point suggest that the model is unlikely to be overfitted to the data on which it was trained.

Observation 3: Our model demonstrates a commendable balance between precision and recall. Our model achieves a $AUPRC$ of 0.319, surpassing the corresponding baseline of 0.012. This shows its effectiveness in distinguishing positive instances and minimizing false positives, especially crucial in our dataset of CI builds with imbalanced class distribution of timeout builds and other builds.

The discriminatory power and calibration of our model are excellent ($AUROC$ of 0.939, $AUPRC$ of 0.319, and Brier score of 0.008). Moreover, the fit is highly stable across different bootstrap samples (mean optimism values of 0.0001 and -0.0002 for $AUROC$ and Brier score, respectively). Lastly, our model successfully balances precision and recall, achieving an $AUPRC$ of 0.319, which surpasses the respective baseline.

TABLE 3: Importance of families.

Family		Overall	Nonlinear
Build history	D.F.	5	2
	χ^2	3,063.19 ***	440.14 ***
Timeout tendency	D.F.	5	3
	χ^2	20.77 ***	20.27 ***
Queued time	D.F.	4	-
	χ^2	6.79 ○	-
Author experience	D.F.	2	-
	χ^2	1.95 ○	-
Size	D.F.	2	-
	χ^2	0.15 ○	-
Entire model	D.F.	18	5
(all families)	χ^2	4,350.03 ***	645.04 ***

○ $p \geq 0.05$; *$p < 0.05$; **$p < 0.01$; ***$p < 0.001$

5.2 (RQ2) What are the most influential features of our models of timeout builds?

Approach. First, we estimate the importance of each family of feature(s) using the Wald χ^2 maximum likelihood (a.k.a., "chunk") tests [43]. The Wald χ^2 value indicates whether the model is statistically different from the same model in the absence of a given independent family of feature(s). The higher the Wald χ^2 value, the greater the explanatory power of the feature family in identifying timeout builds. While analyzing feature families allows understanding the collective impact of feature families on timeout builds, analyzing individual features allows understanding the specific contribution of each feature to timeout builds. Thus, we analyze the Wald χ^2 values of individual features as well.

Furthermore, we complement our analysis by plotting response curves[16] for the most important features to analyze the trend of the probability of a CI build timing out as the feature value varies. All other features are held constant at "typical" values, i.e., median for numeric features and mode for categorical features. These plots also show the 95% confidence intervals of the probabilities that are calculated based on 1,000 bootstrap iterations.

Results. Table 3 shows Wald χ^2 values for each of the families of features of our model. Note that the results are shown in two columns. The *Overall* column shows the explanatory power of all of the degrees of freedom that have been allocated to a family, while the *Nonlinear* column shows the explanatory power that the additional degrees of freedom provide. If no additional degrees of freedom have been allocated to a family, a dash (-) symbol is shown in the nonlinear column. The table shows that the ratio of the Wald χ^2 of all the nonlinear degrees of freedom to that of the entire model is $\frac{645.04}{4,350.03} = 0.15$, indicating that the magnitude of the contribution of additional degrees of freedom to our model is substantial and statistically significant ($p < 0.001$). Moreover, the two families (i.e., build history and timeout tendency) that are allocated additional degrees of freedom contribute a significant amount of explanatory power.

Observation 4: The build history family is the most important family for explaining the likelihood of timeout builds. Table 3 shows that the build history family has the highest Wald χ^2 value. Furthermore, the ratio of the Wald

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

TABLE 4: Importance of features.

Family	Feature		Overall	Nonlinear
Build history	recent build	D.F	1	-
	status	χ^2	1,215.49 ***	-
	timeout ratio	D.F	2	1
		χ^2	739.36 ***	287.02 ***
	recent build	D.F	2	1
	duration	χ^2	133.69 ***	128.98 ***
Timeout tendency	author tendency	D.F	3	2
		χ^2	12.16 **	11.74 **
	file tendency	D.F	2	1
		χ^2	8.89 *	8.22 **
Queued time	queued month	D.F	1	-
		χ^2	2.64 *	-
	queued day	D.F	1	-
		χ^2	2.83 °	-
	queued hour	D.F	1	-
		χ^2	0.84 ○	-
	queued minute	D.F	1	-
		χ^2	0.55 °	-
Author experi-ence	changes to	D.F	1	-
	related files	χ^2	1.67 ○	-
	changes to	D.F	1	-
	any file	χ^2	0.42 ○	-
Size	deletions	D.F	1	-
		χ^2	0.11 °	-
	insertions	D.F	1	-
		χ^2	0.05 °	-

○ $p \geq 0.05$; *$p < 0.05$; **$p < 0.01$; ***$p < 0.001$

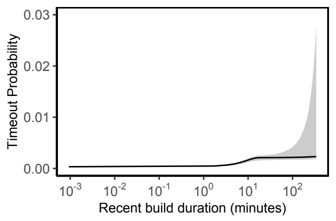


Fig. 3: Build duration vs. the probability of timing out.

χ^2 of build history to that of the entire model is $\frac{3,063.19}{4,350.03} = 0.70$, i.e., the build history family is the only family that can explain 70% of the variance of our model.

Furthermore, we show the importance of individual features in Table 4. The table shows that the recent build status, timeout ratio, and recent build duration are the most important features in the model. This indicates that projects with past timeout builds are more likely to accrue timeout builds. In fact, CI builds are often restarted in response to timeout builds [9]. Doing so without systematically addressing the cause of such timeouts may result in consecutive timeouts. To better understand the underlying relationship between past and future timeouts, we conduct a longitudinal analysis of the incidences of timeout builds in Section 6.

The response curves corresponding to build history features are shown in Fig. 3 and Fig. 4. Fig. 3 shows the response curve for the direction of the relationship between the duration of the most recent build and the probability of a CI build timing out. Accordingly, as the duration of the recent build increases, the probability of the current build timing out increases. While this appears to be a weak relationship in the figure (when compared to the strongest features of our model), the chunk test for this feature, as shown in Table 4, yields an explanatory power of 133.69 out

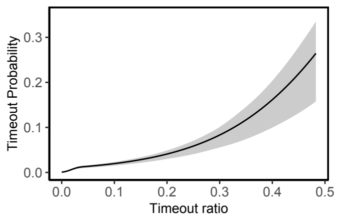


Fig. 4: Timeout ratio vs. the probability of builds timing out.

of 4,350.03, with a significant p-value (< 0.001).

Fig. 4 shows that as the timeout ratio increases, the probability of the current build timing out increases exponentially. This suggests that projects that experienced a high proportion of timeouts in the past will likely continue to suffer from a high rate of timeouts in the future. Note that the 95% confidence intervals are narrow for small values of the explanatory features but tend to broaden as the feature values increase. Having broader confidence intervals does not invalidate our model, but is instead a reflection of how the data in our sample supports the trend (i.e., there are fewer samples supporting the broader areas of the curve).

Observation 5: Timeout tendency is the second strongest family in explaining timeout builds. Table 4 shows that both forms of timeout tendency—file tendency and author tendency—have contributed statistically significant amounts of distinct explanatory power. In particular, the importance of file tendency to the model suggests that changes in certain files have a tendency to lead to timeout builds. A similar trend is observed for author tendency as well. To gain a richer perspective on the relationship between timeout builds and file tendency, we inspect the files that are associated with timeout builds. Of these files, we find that 15% appear more in the change sets of timeout builds than signal-generating builds. That is, the number of timeout builds associated with these files (median = 4) is greater than the number of signal-generating builds associated with them (median = 2). For example, `Branch-SDK/src/main/java/io/branch/referral/Defines.java` is one such file in BranchMetrics/android-branch-deep-linking-attribution project.[17] This file had been associated with 136 timeouts, which is three times more than the number of signal-generating builds associated with the file. A close inspection of the content of the file reveals that the file defines several JSON[18] keys, request paths, link parameters, etc. Overall, the file provides an essential way to access and handle various keys used in the system. Furthermore, we find that 24 artifacts in the project rely on those keys. Thus, such a file may have a broad impact on the system when it is changed, which can consequently impact the build outcome.

Lastly, we find that timeout tendency and build history features together account for a significant portion (4,297.41) of the model's total explanatory power (4,350.03), providing evidence of the gain when fitting the model only by considering strong families of features.

[17]https://github.com/BranchMetrics/android-branch-deep-linking-attribution
[18]https://www.json.org/json-en.html

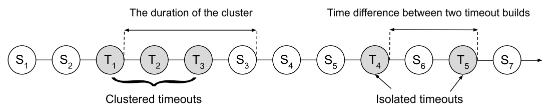


Fig. 5: An example of builds timeline.

Observation 6: Some observed features do not play a significant role in identifying CI builds that are likely to time out. The families of queued time, author experience, and size do not significantly contribute to the explanatory power of our model compared to the remaining two families. The family of features capturing size has the least importance to the model. Table 3 shows Wald χ^2 values for the aforementioned families; the ratio of the Wald χ^2 of the family to the entire model is $\frac{6.79}{4{,}350.03} = 0.0015$, $\frac{1.95}{4{,}350.03} = 0.0004$, and $\frac{0.15}{4{,}350.03} = 0.00003$ for queued time, author experience, and size-related features, respectively.

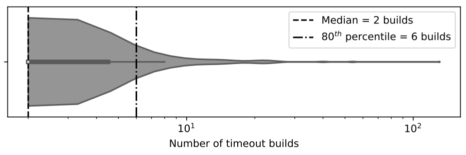
Timeout builds are strongly associated with the project build history and timeout tendency. In contrast, queued time, author experience, and size are weakly associated with timeout builds.

Model re-evaluation. Our *Observation 4* shows that the build history features have a strong explanatory power of our model. Hence, we build naïve baseline models for the three features of the build history family, and compare the $AUROC$ of these baseline models with our initial model discussed in Section 5.1. Firstly, regarding **the naïve baseline for the recent build status**, it posits that the current build status will mirror the previous one. This approach yields an $AUROC$ (Area Under the Receiver Operating Characteristic curve) of 0.7622. Secondly, **the naïve baseline for the timeout ratio** predicts a current build will timeout if the project's historical timeout ratio surpasses 0.5, achieving an $AUROC$ of 0.7212. Lastly, our **naïve baseline for recent build duration** assumes a current build will timeout if the duration of its predecessor exceeds CircleCI's standard time limit (ten minutes).[7] This method results in an $AUROC$ of 0.5. indicating a prediction no better than random chance. This baseline analysis highlights that our model's performance ($AUROC$ of 0.939) surpasses above baselines.

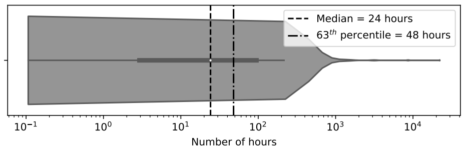
6 Longitudinal Analysis

The interpretation of our model (Section 5.2) indicates that the project build history is a strong indicator of whether a build will time out or not. To better understand this chronological relationship between timeouts, we perform a longitudinal analysis of the occurrences of timeouts.

Approach. To structure our longitudinal analysis, we take inspiration from recent studies of build breakages [31, 44], which achieve substantial improvements by treating consecutive breakages (i.e., those whose prior build was also broken) differently than novel breakages (i.e., those whose prior build was not broken). Similarly, we classify timeout builds as either isolated or clustered. Fig. 5 exemplifies each form. In the figure, circles denoted with an `S` represent signal-generating builds (either passing or failing), while

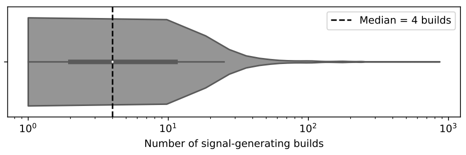


(a) Number of builds in timeout clusters.

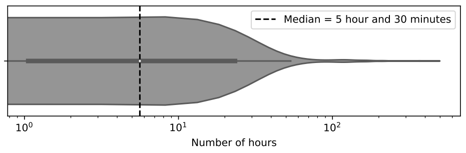


(b) Duration of builds in timeout clusters.

Fig. 6: The number and duration of a timeout cluster.



(a) Distance (number of builds) between an isolated timeout and the closest timeout.



(b) Time difference (hours) between an isolated timeout and the closest timeout.

Fig. 7: The distance and time difference between an isolated timeout and the closest timeout.

circles denoted with a `T` represent timeout builds. The first three timeout builds form *clustered timeout builds*. The last two timeout builds in the timeline are *isolated timeout builds*; they occur in between two signal-generating builds.

To analyze clustered timeout builds, we compute the number of timeout builds that compose a cluster as well as the time duration of the builds in the cluster. For example, the cluster annotated in Fig. 5 is composed of T_1, T_2, and T_3; hence, the number of timeout builds in the cluster is three. This cluster's time duration is the time between the moment that the first timeout in the cluster is observed and the moment the next signal-generating build is observed, i.e., the time between the end of T_1 and the end of S_3.

To analyze isolated timeout builds, we examine how close an isolated timeout build is to another timeout build (either a cluster or an isolated one), i.e., given an isolated timeout, we want to analyze the distance to the closest timeout to this isolated timeout. We measure the distance using the number of signal-generating builds between the isolated timeout and the closest timeout. Also, we analyze the time elapsed between an isolated timeout build and its closest timeout build. For example, consider the isolated timeout T_4 in Fig. 5. The closest timeout build to T_4 is T_5, which is also an isolated timeout. T_5 is one build away from T_4; hence, the number of builds between the two timeouts is one. The time duration until T_5 has occurred is measured by the time difference between the ends of T_4 and T_5.

Results. Our two main observations are as follows.

Observation 7: The majority (64.03%) of timeout builds in our dataset occur in clusters. Fig. 6a shows the distribution of the number of builds that compose a timeout cluster. The figure shows that although timeout clusters are composed of a median of two builds, 20% of the clusters are composed of at least six consecutive timeouts (see the 80^{th} percentile). Fig. 6b shows the distribution of the duration of timeout clusters. Accordingly, it takes a median of 24 hours before a signal-generating build occurs. More extremely, we find that it takes at least 48 hours in 37% of the timeout clusters (see the 63^{th} percentile). For example, the cptactionhank/docker-atlassian-confluence project[19] encountered a cluster of 14 consecutive timeout builds, and it took more than six days for the timeouts to subside.

[19]https://github.com/cptactionhank/docker-atlassian-confluence

Observation 8: Isolated timeout builds are often close to another timeout build. Fig. 7a shows the distribution of the number of builds between an isolated timeout and its closest timeout, and accordingly, the median is four builds. Furthermore, Fig. 7b shows the time difference between two timeouts, one of which is an isolated timeout. Indeed, the closest timeout to an isolated timeout build occurs in less than a day (i.e., five hours and 30 minutes) on the median. Examples of such isolated timeout builds can be found in the spotify/helios project.[20] The project encountered timeout builds in less than an hour (in the extreme case) after an isolated timeout build. Similar cases can be observed in other projects, such as cptactionhank/docker-atlassian-confluence,[19] palantir/atlasdb,[14] and onyx-platform/onyx.[21]

The majority (64.03%) of timeout builds occur in clusters. Moreover, after a timeout build, it takes a substantial amount of time (a median of 24 hours) until the occurrence of a signal-generating build.

7 Thematic Analysis

The discovery of timeout clusters and the key influential features led us to explore the root causes behind CI timeouts, and below, we detail our approach and results.

Approach. We start by gathering links to community discussions on timeout builds in the projects that we analyzed in our study by using GitHub search, with queries like "ci timeout" and "circleci time out" to find relevant issues and pull requests (PRs). By analyzing these, we aim to understand the reasons behind CI timeouts. Our search finds 79 issues and PRs with 406 comments related to CI timeouts in the projects examined. After collecting relevant documents, we perform a systematic inspection and a thematic analysis [47]. In the initial iteration, the first and second authors collaboratively review titles, descriptions, and discussion threads to create codes summarizing reasons

[20]https://github.com/spotify/helios
[21]https://github.com/onyx-platform/onyx

for timeout builds. If the reason remains unclear, we mark it as "Unknown." We then identify common themes that span across codes (which are not necessarily mutually exclusive), linking together similar topics or underlying issues.

In the second iteration of the analysis, the first and second authors independently assign themes to discussions, with any disagreements resolved through discussion or, if needed, a deciding vote by the last author. However, all disagreements were resolved without needing this vote. To assess the reliability of our themes and coding, we calculate Cohen's Kappa coefficient [8], which is 0.778, showing substantial agreement [33].

Results. Our thematic analysis yields six themes, as shown in Table 5. Each theme describes a reason for build timeouts and the solutions that developers implemented. The total frequency of the identified reasons is greater than 100% because multiple themes may apply to the same case.

(T1) Efficiency Issues in Testing. In this theme, testing issues cause timeouts for two reasons: First, extensive tests lead to timeouts, and prompt developers to exclude them from CI pipelines. For example, in the cptactionhank/dockeratlassian-jira project, developers have removed a set of long-duration tests to mitigate timeouts.[22] Second, timeouts may be caused by misconfigurations or an excessive number of tasks, which can prolong build times. For example, the CI pipeline in the tikv/tikv project is configured with a code coverage tool that prolongs the build time, and is to blame for timeouts.[23] The developers initiated executing this code coverage analysis in a separate build to reduce the prolonged build durations.

(T2) Project-Specific Issues. Timeouts arise from project-level parameters, such as the selection of programming languages, databases, and Android emulators. For example, in the BranchMetrics/android-branch-deep-linking-attribution project, the selected Android emulator becomes unresponsive, entering into infinite loops during the build process.[24] This unresponsiveness contributes to timeouts. To avoid such timeouts, the developers changed the version of the Android emulator they were using.

(T3) Network Issues. Another common reason for timeout builds is due to network issues, such as incorrect network settings, API network errors, and server timeouts. For instance, in the spacetelescope/notebooks project, slow network requests led to CI timeouts.[25] Attempts to address this included extending the timeout limit from 10 to 20 minutes, which, while reducing timeouts, is not ideal as it can increase costs due to longer billable service usage.

(T4) Resource Constraints. Timeout builds can be attributed to resource constraints, such as limitations in RAM and/or CPU, or parallelism constraints that manifest as race conditions. For example, in the tendermint/tendermint project, a statement to cause a thread to wait (`time.Sleep(time.Second)`) was added to mitigate race conditions that otherwise lead to timeouts.[26]

(T5) Efficiency Issues in the CI Provider. Builds may time out due to inefficiencies in the CI provider's infrastructure. For example, in the influxdata/kapacitor project, developers observed that certain builds running quickly on local machines faced delays and timeouts on CircleCI.[27] Similarly, in a PR within the influxdata/influxdb project, developers noted timeouts, which were suspected to be due to the limitations of the infrastructure on the CI provider's side.[28] In both cases, developers were in favour of restarting the build even though restarting without addressing the underlying issues can waste resources [37].

(T6) Containerized Environment Challenges. Timeouts within a containerized environment can be traced to container maintenance and configuration issues. For example, in the moby/libnetwork project, timeouts occurred because the network interfaces of Docker containers were not adequately cleaned up after the tests were completed.[29] On the other hand, in the Homebrew/brew project, Docker containers play a crucial role in the CI process by providing isolated environments for the building software packages.[30] This process times out when building large software packages, and the developers discussed the need to lift the timeout limit.

Emergent themes of root causes for CI timeouts range from technical challenges (T3, T4, and T6) and testing inefficiencies (T1) to project-specific (T2) issues and limitations with CI providers (T5).

TABLE 5: The extracted themes for CI timeout builds.

ID	Theme	Frequency (%)	Solution
(T1)	Efficiency Issues in Testing	28 (35.44%)	Remove long-running tests and/or execute long-running tasks in separate builds.
(T2)	Project-Specific Issues	10 (12.65%)	Project-specific patterns.
(T3)	Network Issues	8 (10.12%)	Increase the CI timeout setting.
(T4)	Resource Constraints	8 (10.12%)	Induce waiting in threads and/or processes.
(T5)	Efficiency Issues in the CI Provider	6 (07.59%)	Restart the build.
(T6)	Containerized Environment Issues	6 (07.59%)	Increase the CI timeout setting.
(TU)	Unknown	16 (20.25%)	Increase the CI timeout setting.

8 Threats to Validity

Construct Validity. We have not measured all potential characteristics that impact timeout builds. For instance, the

[22]https://github.com/cptactionhank/docker-atlassian-jira/commit/dbb3b143efe02351614e6f33be4b0239991f40f2
[23]https://github.com/tikv/tikv/issues/3012
[24]https://github.com/BranchMetrics/android-branch-deep-linking-attribution/pull/400
[25]https://github.com/spacetelescope/notebooks/issues/87
[26]https://github.com/tendermint/tendermint/issues/846
[27]https://github.com/influxdata/kapacitor/pull/1631
[28]https://github.com/influxdata/influxdb/pull/8961
[29]https://github.com/moby/libnetwork/pull/1325
[30]https://github.com/Homebrew/brew/issues/10597

performance characteristics of the CI server, such as parallelism and scalability, are not included in our models. Such features may better explain the likelihood of timeout builds than the features we use. However, such information is not available publicly. To mitigate this, we select a set of 19 features spanning five dimensions of CI builds by consulting the related literature on the build outcome prediction [7, 40, 44] and defect prediction [52].

Internal Validity. We may have missed confounding factors that could impact our interpretations. For example, we observe that the longer the duration of the previous build, the higher the likelihood of CI builds timing out, but this might reflect limited CI service resources, making the observed relationship coincidental. Additionally, there could be overlooked implicit aspects of builds, such as the CI provider's workload capacity (not available for us via public APIs/datasets) or configurable time limits, that might provide further context to our model.

Also, the time limit (the `no_output_timeout` setting), which developers can set, may have a relationship to the probability of timeout builds; when a project is assigned a larger time limit, the likelihood of encountering build timeouts naturally decreases. We collect `no_output_timeout` values for each build in our dataset (assuming the default time limit for the build that does not have the configuration explicitly set). Upon rerunning our models, we discover that the `no_output_timeout` settings do not significantly explain the model's outcomes. For a more detailed exploration of this aspect, we direct interested readers to our Online Appendix G,[10] which contains an in-depth overview of this updated model. Note that the goal of our study is to identify features that provide insights into the project's overall health in terms of timeouts, irrespective of whether those features share causal or correlational relationships with timeout builds. We encourage future research to explore causal links between our features and timeout builds.

Our decision to use statistical models was made because of their ability to elucidate the influences of the set of studied features on timeout builds. We recognize that this choice might introduce bias, especially when compared to visually intuitive and interpretable machine-learning models, such as decision trees. For further analysis, we construct a decision tree using the same dataset. This decision tree does not yield substantially new insights, suggesting that this experimental design choice is not a substantial threat to the validity of our results. For a detailed exploration of our decision tree analysis, we invite readers to consult our Online Appendix H.[10]

External Validity. Our models are built using data from projects that used CircleCI. As such, our results may not generalize to other CI services. However, there is nothing inherently service-specific about the phenomenon of timeout builds. Nonetheless, since we select statically-computable and CI service-agnostic features, our replication package[10] can be used to accelerate replication studies for other CI services (e.g., GitHub Actions, which is also known to be a popular CI service [19]). We select a sample of the 24 most timeout-prone projects for our analysis. As such, our results may not be generalized to all CircleCI users. We apply a set of conservative filters to exclude early-stage or immature projects where timeout builds are less relevant.

9 Related Work

A popular line of prior work focused on the difficulties of adopting CI services [16, 17, 18, 29, 57]. For example, Widder et al. [57] surveyed 132 developers and interviewed 12 developers, and reported several limitations in Travis CI, such as slow feedback, long wait times in the queue, and other resource constraints, including memory, disk space, and available build time. Also, Hilton et al. [29] surveyed 523 developers worldwide, and reported that 38% of them found overly long build durations as a barrier in CI. Ghaleb et al.'s [17] study is perhaps the most closely related work. Their study focused on the characteristics associated with long-duration CI builds by examining 104,442 Travis CI builds across 67 GitHub projects. Their investigation revealed that builds are more likely to take longer if they occur on weekdays or during daytime hours, and they discovered that 40% of these builds exceed 30 minutes. Moreover, they built a model to distinguish between long-duration and short-duration builds in Travis CI, identifying the `fast_finish` setting as a key feature. That said, our work differs from that of prior work by shifting the focus to examine a distinct type of long-running builds—those resulting in timeouts without providing a clear signal.

Previous research proposed methods to predict build failures using build history and change set features [7, 27, 31, 45]. For instance, Hassan and Wang [27] used past build characteristics for predictions, while Chen et al. [7] introduced an adaptive model that adjusts based on the outcome of the last build.

Other researchers proposed techniques to reduce the time-to-feedback by skipping builds that are unnecessary [1, 32]. For example, Abdalkareem et al. [1] inspected 1,813 commits that developers flagged to skip the build. The authors identified the reasons for skipping CI builds, and proposed a rule-based method (*CI-Skipper*) that automatically identifies the commits that could be CI skipped.

Other work investigated test case granularity solutions to reduce build time [13, 35, 36, 40]. Pan and Pradel [40] presented a test suite-level failure prediction approach, which predicts whether a particular code change would lead to a build failure. Other studies [2, 53] introduced test case prioritizing methods to rank test cases by their tendency to lead to build failures, and executed the test cases in the order of highest tendency to the least.

Our study complements these studies by specifically focusing on timeout builds. Recent studies [9, 14] have also raised concerns about timeouts. Gallaba et al. [14] identified 17,917 timeout builds across 936 GitHub projects; they mentioned that determining if a build will timeout is a form of the halting problem [54]. Our study adds to prior work by explicitly focusing on characterizing timeouts.

10 Implications

Project build history and timeout clusters can provide useful information to proactively allocate resources (for CI providers) and minimize CI waste (for CI consumers). *Observation 5* shows that the history of a project's builds is the strongest indicator of whether a build will time out. Additionally, we found that timeout builds occur in clusters (*Observation 7*), which generates considerable amounts of wasted build time. By leveraging these project tendencies and timeout patterns, CI providers can anticipate timeout builds and take appropriate action. For example, if additional resources are available, it may be more cost-effective to proactively allocate them to builds with a high likelihood of timing out in order to mitigate such issues. This is by no means a simple action since one cannot know in advance the quantities of additional resources required to allow the problematic build to terminate with a pass or fail signal [14]; however, it is likely that timeouts will be retried [9], which will likely cascade into a series of timeout builds (i.e., clusters), generating more waste than a proactive increment to the resources of the initial build would. After a timeout build, 24 hours are taken (on median) for a project to see a passing or failing build. Moreover, clusters of timeouts suggest a substantial problem, like a shared environmental condition or code change that introduces timeouts, rather than just flakiness. We recommend developers investigate the root causes (detailed in our Section 7)) to better understand and prevent future timeouts.

Prioritizing files that are prone to build timeouts can optimize resource allocation and may help to avoid incidences of timeout builds. *Observation 4* indicates that certain file characteristics can increase the likelihood of a CI build timing out. By inspecting examples of timeout builds, we find that some files are more often implicated in Timeouts. This may indicate that timeout builds are localized, and are often triggered by the changes to a small fraction of project files. Upon a closer inspection of our dataset, we find cases where developers make changes to certain files to fix the issue of timeouts, e.g., commenting out long-running test cases in test files,[31] adjusting test settings (such as changing the android emulator version[32]), and changing CI configurations.[33] We provide more such examples in Section 7 and our Online Appendix J.[10] Hence, a tool that ranks such timeout-prone files based on the strength of their association with builds that time out may have a potential impact. For example, a tool may flag a change to a file if it is likely to eventually lead to a timeout build, letting project maintainers direct their efforts to optimize components that pose the most elevated risk for timeout builds. This strategy could prioritize the files that are frequently associated with timeouts. Specifically, files that have a consistent track record of leading to timeouts across multiple builds ought to be placed at the top of the prioritization list for analysis. In addition to raising developer awareness when changing timeout-prone files, such tools could guide developers to make more informed decisions about allocating resources to mitigate timeouts.

Researchers should propose approaches for predicting timeout builds to assist developers in preventing timeouts. *Observations 1* and *2* show that our model achieves a high level of discriminatory power ($AUROC$=0.939), is well calibrated in terms of risk estimates (Brier score of 0.008), and is highly stable for explaining the incidences of timeouts. However, our primary goal in this paper is to use statistical models to characterize and understand timeout builds rather than to predict future timeouts. We encourage researchers to build upon our findings (the important features we identified in *Observations 4* and *5* and the temporal aspects in the analysis of build data in *Observations 7* and *8*), as well as those in the context of build failure prediction, to develop more powerful prediction models for timeout builds. For example, prior work (e.g., [7]) built machine-learning models to predict build failures, but our findings suggest that such approaches could be extended to predict timeout builds.

[31] https://github.com/cptactionhank/docker-atlassian-confluence/commit/e81db60ee6cbee71bb427aa015afb3b9762d029c

[32] https://github.com/BranchMetrics/android-branch-deep-linking-attribution/pull/400

[33] https://github.com/autoreject/autoreject/pull/194/commits/92b388594d3c1cb2678c1f189940b84cfc9b9f5c

11 Conclusion

This paper examines timeout builds in 24 highly prone open-source projects using CircleCI. Various project characteristics, including build history, queue time, and author experience, are analyzed using statistical models to understand their impact on timeout occurrences. Our findings highlight that the most influential factors are build history and timeout tendencies with 64.03% of timeouts occurring consecutively and a 24-hour delay on average before the next build. Our thematic analysis of GitHub discussions reveals root causes for timeouts ranging from testing inefficiencies and resource limitations to CI provider issues.

References

[1] R. Abdalkareem, S. Mujahid, E. Shihab, and J. Rilling, "Which commits can be ci skipped?" *Transactions on Software Engineering*, vol. 47, 2019.

[2] M. Bagherzadeh, N. Kahani, and L. Briand, "Reinforcement learning for test case prioritization," *IEEE Transactions on Software Engineering*, vol. 48, 2021.

[3] A. O. Balogun, S. Basri, J. A. Said, V. E. Adeyemo, A. A. Imam, and A. O. Bajeh, *Software defect prediction: analysis of class imbalance and performance stability*. School of Engineering, Taylor's University, 2019.

[4] G. W. Brier, *Verification of forecasts expressed in terms of probability*. American Meteorological Society, 1950, vol. 78.

[5] G. Cavalcanti, P. Borba, G. Seibt, and S. Apel, "The impact of structure on software merging: semistructured versus structured merge," in *Proceedings of the International Conference on Automated Software Engineering*, 2019.

[6] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "Smote: synthetic minority over-sampling technique," *Journal of artificial intelligence research*, vol. 16, 2002.

[7] B. Chen, L. Chen, C. Zhang, and X. Peng, "Buildfast: History-aware build outcome prediction for fast feedback and reduced cost in continuous integration," in *Proceedings of the 35th International Conference on Automated Software Engineering*, 2020.

[8] J. Cohen, "Weighted kappa: Nominal scale agreement provision for scaled disagreement or partial credit." *Psychological Bulletin*, vol. 70, 1968.

[9] T. Durieux, C. Le Goues, M. Hilton, and R. Abreu, "Empirical study of restarted and flaky builds on travis

ci," in *Proceedings of the 17th International Conference on Mining Software Repositories*, 2020.

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

[11] B. Efron, "How biased is the apparent error rate of a prediction rule?" *Journal of the American statistical Association*, vol. 81, 1986.

[12] J. G. Eisenhauer, *Degrees of Freedom in Statistical Inference.* Springer Berlin Heidelberg, 2011.

[13] S. Elbaum, G. Rothermel, and J. Penix, "Techniques for improving regression testing in continuous integration development environments," in *Proceedings of the 22nd SIGSOFT International Symposium on Foundations of Software Engineering*, 2014.

[14] K. Gallaba, M. Lamothe, and S. McIntosh, "Lessons from eight years of operational data from a continuous integration service: an exploratory case study of circleci," in *Proceedings of the 44th International Conference on Software Engineering*, 2022.

[15] K. Gallaba and S. McIntosh, "Use and misuse of continuous integration features: An empirical study of projects that (mis) use travis ci," *Transactions on Software Engineering*, vol. 46, 2018.

[16] T. A. Ghaleb, S. Hassan, and Y. Zou, "Studying the interplay between the durations and breakages of continuous integration builds," *Transactions on Software Engineering*, vol. 49, 2022.

[17] T. A. Ghaleb, D. A. Da Costa, and Y. Zou, "An empirical study of the long duration of continuous integration builds," *Empirical Software Engineering*, vol. 24, 2019.

[18] T. A. Ghaleb, D. A. da Costa, Y. Zou, and A. E. Hassan, "Studying the impact of noises in build breakage data," *Transactions on Software Engineering*, vol. 47, 2019.

[19] M. Golzadeh, A. Decan, and T. Mens, "On the rise and fall of ci services in github," in *International Conference on Software Analysis, Evolution and Reengineering*. IEEE, 2022.

[20] G. Gousios, M. Pinzger, and A. v. Deursen, "An exploratory study of the pull-based software development model," in *Proceedings of the 36th international conference on software engineering*, 2014.

[21] J. A. Hanley and B. J. McNeil, "The meaning and use of the area under a receiver operating characteristic (roc) curve." *Radiology*, vol. 143, 1982.

[22] N. R. Hansen and A. Sokol, "Degrees of freedom for nonlinear least squares estimation," *arXiv preprint arXiv:1402.2997*, 2014.

[23] F. E. Harrell *et al.*, *Regression modeling strategies: with applications to linear models, logistic regression, and survival analysis*, 2001, vol. 608.

[24] F. E. Harrell Jr, K. L. Lee, R. M. Califf, D. B. Pryor, and R. A. Rosati, "Regression modelling strategies for improved prognostic prediction," *Statistics in medicine*, vol. 3, 1984.

[25] F. E. Harrell Jr, K. L. Lee, D. B. Matchar, and T. A. Reichert, "Regression models for prognostic prediction: advantages, problems, and suggested solutions." *Cancer treatment reports*, vol. 69, 1985.

[26] T. Hasanin and T. Khoshgoftaar, "The effects of random undersampling with simulated class imbalance for big data," in *international conference on information reuse and integration (IRI)*, 2018.

[27] F. Hassan and X. Wang, "Change-aware build prediction model for stall avoidance in continuous integration," in *Proceedings of the International Symposium on Empirical Software Engineering and Measurement*, 2017.

[28] H. He and E. A. Garcia, "Learning from imbalanced data," *IEEE Transactions on knowledge and data engineering*, vol. 21, 2009.

[29] M. Hilton, N. Nelson, D. Dig, T. Tunnell, D. Marinov *et al.*, *Continuous integration (CI) needs and wishes for developers of proprietary code*. Corvallis, OR : Oregon State University, Dept. of Computer Science, 2016.

[30] M. Hilton, T. Tunnell, K. Huang, D. Marinov, and D. Dig, "Usage, costs, and benefits of continuous integration in open-source projects," in *Proceedings of the 31st International Conference on Automated Software Engineering*, 2016.

[31] X. Jin and F. Servant, "A cost-efficient approach to building in continuous integration," in *Proceedings of the 42nd International Conference on Software Engineering*, 2020.

[32] ——, "Which builds are really safe to skip? maximizing failure observation for build selection in continuous integration," *Journal of Systems and Software*, vol. 188, 2022.

[33] J. R. Landis and G. G. Koch, "The measurement of observer agreement for categorical data," *biometrics*, 1977.

[34] M. R. Lavery, P. Acharya, S. A. Sivo, and L. Xu, "Number of predictors and multicollinearity: What are their effects on error and bias in regression?" *Communications in Statistics-Simulation and Computation*, vol. 48, 2019.

[35] J. Liang, S. Elbaum, and G. Rothermel, "Redefining prioritization: continuous prioritization for continuous integration," in *Proceedings of the 40th International Conference on Software Engineering*, 2018.

[36] M. Machalica, A. Samylkin, M. Porth, and S. Chandra, "Predictive test selection," in *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice*. IEEE, 2019.

[37] R. Maipradit, D. Wang, P. Thongtanunam, R. G. Kula, Y. Kamei, and S. McIntosh, "Repeated Builds During Code Review: An Empirical Study of the OpenStack Community," in *Proc. of the International Conference on Automated Software Engineering*, 2023.

[38] S. McIntosh and Y. Kamei, "Are Fix-Inducing Changes a Moving Target? A Longitudinal Case Study of Just-In-Time Defect Prediction," *Transactions on Software Engineering*, vol. 44, 2018.

[39] S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan, "An empirical study of the impact of modern code review practices on software quality," *Empirical Software Engineering*, vol. 21, 2016.

[40] C. Pan and M. Pradel, "Continuous test suite failure prediction," in *Proceedings of the 30th SIGSOFT International Symposium on Software Testing and Analysis*, 2021.

[41] K. Pearson, "Contributions to the mathematical theory of evolution," *Philosophical Transactions of the Royal Society of London. A*, vol. 185, 1894.

[42] J. Prusa, T. M. Khoshgoftaar, D. J. Dittman, and

A. Napolitano, “Using random undersampling to alleviate class imbalance on tweet sentiment data,” in *2015 IEEE international conference on information reuse and integration*, 2015.

[43] J. N. Rao and A. J. Scott, “The analysis of categorical data from complex sample surveys: chi-squared tests for goodness of fit and independence in two-way tables,” *Journal of the American statistical association*, vol. 76, no. 374, pp. 221–230, 1981.

[44] I. Saidani, A. Ouni, M. Chouchen, and M. W. Mkaouer, “Predicting continuous integration build failures using evolutionary search,” *Information and Software Technology*, vol. 128, 2020.

[45] I. Saidani, A. Ouni, and M. W. Mkaouer, “Improving the prediction of continuous integration build failures using deep learning,” *Automated Software Engineering*, vol. 29, 2022.

[46] T. Saito and M. Rehmsmeier, “The precision-recall plot is more informative than the roc plot when evaluating binary classifiers on imbalanced datasets,” *PloS one*, vol. 10, 2015.

[47] C. B. Seaman, “Qualitative methods in empirical studies of software engineering,” *Transactions on software engineering*, vol. 25, 1999.

[48] C. Spearman, *The proof and measurement of association between two things.* Appleton-Century-Crofts, 1961.

[49] D. Ståhl and J. Bosch, “Experienced benefits of continuous integration in industry software product development: A case study,” in *Proceedings of the 12th IASTED International Conference on Software Engineering*, 2013.

[50] ——, “Industry application of continuous integration modeling: a multiple-case study,” in *Proceedings of the 38th International Conference on Software Engineering Companion*, 2016.

[51] X. Tan, M. Zhou, and Z. Sun, “A first look at good first issues on github,” in *Proceedings of the 28th Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020.

[52] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, A. Ihara, and K. Matsumoto, “The impact of mislabelling on the performance and interpretation of defect prediction models,” in *Proceedings of the 37th International Conference on Software Engineering*, 2015.

[53] S. W. Thomas, H. Hemmati, A. E. Hassan, and D. Blostein, “Static test case prioritization using topic models,” *Empirical Software Engineering*, vol. 19, 2014.

[54] A. Turing, “On computable numbers, with an application to the entscheidungsproblem. a correction,” *Proceedings of the London*, 1938.

[55] B. Vasilescu, Y. Yu, H. Wang, P. Devanbu, and V. Filkov, “Quality and productivity outcomes relating to continuous integration in github,” in *Proceedings of the 10th joint meeting on foundations of software engineering*, 2015.

[56] C. Vassallo, S. Proksch, H. C. Gall, and M. Di Penta, “Automated reporting of anti-patterns and decay in continuous integration,” in *Proceedings of the 41st International Conference on Software Engineering*, 2019.

[57] D. G. Widder, M. Hilton, C. Kästner, and B. Vasilescu, “A conceptual replication of continuous integration pain points in the context of travis ci,” in *Proceedings of the 27th Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019.

[58] F. Zampetti, G. Bavota, G. Canfora, and M. Di Penta, “A study on the interplay between pull request review and continuous integration builds,” in *Proceedings of the 26th International Conference on Software Analysis, Evolution and Reengineering*, 2019.

[59] F. Zampetti, C. Vassallo, S. Panichella, G. Canfora, H. Gall, and M. Di Penta, “An empirical characterization of bad practices in continuous integration,” *Empirical Software Engineering*, vol. 25, 2020.

Nimmi Weeraddana is a Ph.D. candidate in the Cheriton School of Computer Science at the University of Waterloo, Canada. Her research interests include build systems, mining software repositories, and diversity in software engineering. Find more about her at https://rebels.cs.uwaterloo.ca/member/nimmi.html.

Mahmoud Alfadel is a postdoctoral researcher in the Cheriton School of Computer Science at the University of Waterloo. His research interests include mining software repositories, empirical software engineering, software ecosystems, and release engineering. You can find more about him at https://rebels.cs.uwaterloo.ca/member/mahmoud.html.

Shane McIntosh is an associate professor in the Cheriton School of Computer Science at the University of Waterloo, where he leads the Software Repository Excavation and Build Engineering Labs (Software REBELs). In his research, Shane uses empirical methods to study software build systems, release engineering, and software quality: https://rebels.cs.uwaterloo.ca/.