What Factors Affect the Build Failures Correction Time? A Multi-Project Study

Gustavo Ivens
Federal University of Ceará (UFC)
Quixadá, CE, Brazil
gustavo_ivens@alu.ufc.br

Carla Bezerra
Federal University of Ceará (UFC)
Quixadá, CE, Brazil
carlailane@ufc.br

Anderson Uchôa
Federal University of Ceará (UFC)
Itapajé, CE, Brazil
andersonuchoa@ufc.br

Ivan Machado
Federal University of Bahia (UFBA)
Salvador, BA, Brazil
ivan.machado@ufba.br

ABSTRACT

Continuous Integration (CI) is a widely adopted practice in modern software engineering that involves integrating developers' local changes with the project baseline daily. Despite its popularity, recent studies have revealed that integrating changes can be time-consuming, requiring significant effort to correct errors that arise. This can lead to development activities being paused, including the addition of new features and fixing bugs, while developers focus on analyzing and correcting build failures. In this study, we investigate the factors that influence the time taken to correct build failures in CI. Specifically, we analyze the impact of developer activity, project characteristics, and build complexity on build failure correction time. To conduct our analysis, we collected data from 18 industrial projects of a software company, calculating 13 metrics for each project based on the literature on build failures analysis. We used association rules, a data mining technique, to examine the relationship between the defined factors and build failure correction time. Our findings reveal significant correlations between the factors studied and the duration of build failure correction time. Specifically, we found that more experienced developers require less time to correct build failures, while build failures that originate in the early stages of the project are resolved more quickly. Additionally, we observed that build failures with more lines and modified files tend to have longer correction times. Overall, this study sheds light on the factors that impact build failure correction time in CI. By identifying these factors, our findings can help software development teams optimize their CI processes and minimize the impact of build failures on development activities.

CCS CONCEPTS

• **Software and its engineering → Maintaining software**; • **General and reference → Empirical studies**.

KEYWORDS

empirical, continuous integration, build failures, association rules

ACM Reference Format:
Gustavo Ivens, Carla Bezerra, Anderson Uchôa, and Ivan Machado. 2023. What Factors Affect the Build Failures Correction Time? A Multi-Project Study . In *17th Brazilian Symposium on Software Components, Architectures, and Reuse (SBCARS 2023), September 25–29, 2023, Campo Grande, Brazil.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3622748.3622753

SBCARS 2023, September 25–29, 2023, Campo Grande, Brazil

1 INTRODUCTION

Continuous Integration (CI) is an adopted practice in modern software engineering that has received significant attention from both industry and academia. CI involves the continuous integration of small changes in the source code into the master branch of a shared repository, which enables the frequent delivery of high-quality software [4, 14, 24]. The primary goal of CI is to improve code quality by reducing the time between the implementation of a new software version and its deployment to the production environment through an automated CI pipeline [29]. This pipeline typically includes activities such as building, testing, and static code analysis, which are executed frequently to ensure that any errors or bugs are detected and corrected early in the development process [17, 26].

Although CI has significantly improved the software development process, CI errors can pose a significant threat to development efficiency due to the time required to resolve them [20]. These errors typically occur during the compilation and testing of changes submitted to the shared repository, resulting in failed builds [33]. Research has shown that failed builds can significantly hamper the development process, leading to project delays and increased costs [3, 20, 24]. For instance, while a build failure is being analyzed and fixed, the work of other team members may come to a halt, causing further delays and reducing productivity [3, 20, 24]. As such, it is essential to address CI errors promptly and efficiently to minimize their impact on the development process and ensure the timely delivery of high-quality software products.

Several studies have explored the root causes of failed builds by analyzing data from projects to identify measurable indicators that can predict the success or failure of a build [15, 20, 24, 31]. To this end, Hassan [14] and Vassallo et al. [30] have examined common errors that occur during failed builds to determine correction patterns and develop automated correction tools. In emphasizing the importance of prompt error resolution, Vassallo et al. [30] urge developers to identify the underlying causes of failed builds and rectify them as soon as possible to avoid project delays. Complementary, Silva and Bezerra [27] found that resolving failed builds in industrial projects can take months based on build history analysis.

Therefore, it is crucial to identify the factors that can affect the correction time of build failures, as this understanding can inform strategies to minimize the interval between build failure detection and resolution [7, 12]. Failing to do so may mislead developers or project managers on the possible actions that can be taken to reduce

the build failure correction time, ultimately hindering the benefits that CI can bring to the software development process.

This paper aims to fill these gaps by investigating how certain factors related to developer activity, project characteristics, and build complexity affect build failure correction time. We conducted a study using data from a large software company and defined a set of metrics to represent different properties of these factors. To analyze the relationship between these factors and build failure correction time, we used association rule mining [1]. Specifically, we mined associations from 413 failed builds across 18 industrial projects. To support our observations, we analyzed 13 well-established properties, quantified by metrics, that represent developer activity, project characteristics, and build complexity factors.

Our observations yielded several findings. Firstly, more experienced developers take less time to correct build failures. Secondly, build failures that originated in the early stages of the project are resolved in short time intervals. Lastly, complex builds can have a longer correction time interval. In addition to these findings, our contributions include a discussion of their implications and a novel dataset, available for researchers to further investigate the impact of these factors on build failure correction time.

2 BACKGROUND

2.1 Build Failures Correction Time

Continuous Integration (CI) entails regularly integrating code changes into a shared repository and building and testing them automatically. The complete process of executing all the steps in the CI pipeline is called a build. When a failure occurs during the execution of the CI pipeline, it is usually referred to as a build failure. A failed build indicates that there are problems with the modifications the developer pushed to the repository, and fixing it is a task that should be done as soon as possible [9].

Build failure correction time is the time interval between a failed build and the next successfully executed build and represents a central point of the CI process [27]. Recent studies have indirectly investigated the correction time of these errors and found it to be generally high [10, 27]. For instance, Silva et al. [27] analyzed the correction intervals of bad builds of two industrial projects through the Travis CI log records and observed that more than half of the failed builds took more than three hours to fix.

Build failures configure a problematic point within the software development process, mainly due to the delay they can cause to the project's progress. This is because feature development and release activities get stuck while the issue is being analyzed and fixed by the developers [20]. Generally, the longer it takes a developer to fix a bug, the less time she will spend on effectively productive activities, and the longer it will take to send new features or bug fixes to the client [21]. Therefore, reducing the build failure correction time is essential for software development projects to ensure faster delivery of new features and bug fixes.

2.2 Association Rules Mining

In this work, we employed association rule mining [1] to identify correlations between factors related to developer activity, project characteristics, build complexity, and the time required to correct build failures. Association rule mining is a data mining technique that aims to discover relationships among a set of data items [28]. To perform the associative analysis task, we implemented a mining model of multidimensional association rules. These rules represent patterns in a database and indicate a strong link between data items [22, 28]. Given a database D a multidimensional association rule $X \rightarrow Y$ is an implication of the form: $X_1 \wedge X_2 \wedge ... \wedge X_n \rightarrow Y_1 \wedge Y_2 \wedge ... \wedge Y_m$, where $1 \leq n$, $1 \leq m$ and $X_i (1 \leq i \leq n)$ and $Y_i (1 \leq i \leq m)$ are conditions defined in terms of the distinct attributes of D [22]. Thus, a multidimensional association rule $X \rightarrow Y$ suggests a relationship between the conditions that make up the antecedent X and the conditions that make up the consequent Y, indicating with a certain degree of assurance, that the occurrence of X implies the occurrence of Y.

There are three key metrics commonly used to filter the relevant association rules: *Support*, *Confidence*, and *Lift*. *Support* of a rule $X \rightarrow Y$ refers to the proportion of transactions in D that satisfy the conditions defined for both X and Y. It can be calculated using the expression $Sup(X \rightarrow Y) = \frac{|T_{X \cup Y}|}{|T|}$, where $T_{X \cup Y}$ is the set of transactions that satisfy the conditions of X and Y, and T is the set of all transactions. *Confidence*, on the other hand, analyzes the validity of a rule by calculating the proportion of transactions that satisfy both X and Y among the transactions that satisfy X. It indicates the probability of Y occurring once X has already occurred and can be calculated using the expression $Conf(X \rightarrow Y) = \frac{Sup(X \rightarrow Y)}{Sup(X)}$, where $Sup(X)$ corresponds to the number of transactions that meet the conditions for X [28].

Finally, the *Lift* indicates the extent to which the occurrence of X influences the occurrence of Y. Lift formula is $Lift(X \rightarrow Y) = \frac{Conf(X \rightarrow Y)}{Sup(Y)}$. Lift determines if the association between the antecedent X and the consequent Y is null, positive, or negative. If $Lift = 1$, there is no relationship between X and Y, meaning that the occurrence of X does not influence the occurrence of Y. If $Lift > 1$, there is a positive relationship between X and Y, which means that the occurrence of X increases the chances of Y happening. Conversely, if $Lift < 1$, there is a negative relationship between X and Y, and the occurrence of X decreases the likelihood of Y [22].

3 STUDY SETTINGS

3.1 Goal and Research Questions

We defined the goal of our study using the Goal-Question-Metric (GQM) template [6]: **analyze** the factors that affect the time taken to correct build failures; **for the purpose of** understanding what makes a build failure take more or less time to fix; **with respect to** developer activity, project characteristics, and build complexity; **from the viewpoint of** developers; **in the context of** 18 closed-source projects. Our research questions (RQs) are:

RQ$_1$: *Is developer activity related to longer or shorter build failure correction time?* – **RQ$_1$** aims to identify which metrics related to developer activities within a project influence the duration of build failure correction. By answering this question, we can determine which developer activities align with longer or shorter build failure correction time, thereby guiding developers on which practices or behaviors to follow or avoid.

RQ$_2$: *Do project characteristics affect build failure correction time?* – **RQ$_2$** aims to understand whether a project's size, age, and team

size have any relationship with the duration of build failure correction. This RQ is important to reflect on how the project's characteristics affect the duration of build failure correction. Thus, $\mathbf{RQ}_2$ might shed light on strategies to adopt when a project has characteristics that contribute to an increase in the duration of build failure correction.

$\mathbf{RQ}_3$: *Does build complexity correlate with build failure correction time?* – $\mathbf{RQ}_3$ seeks to determine whether there is a relationship between a more or less complex failed build and the time required to fix it. This RQ is crucial to verifying whether the good practice of keeping builds simple results in a reduction in the time taken to correct build failures [18].

3.2 Study Steps and Procedures

The **replication package** of our study, including all data, models, and scripts, is available at [16].

Step 1: Selection of target metrics. To measure the impact of developer activities, project characteristics, and build complexity on build failure correction time, we first identified the most commonly reported metrics in the literature [2, 15, 20, 24, 25]. Most of these studies use statistical models and machine learning techniques to predict build success or failure based on a project's build history [2, 24, 25]. We then selected a subset of the most relevant metrics, guided by an expert in CI from our industrial partner. Table 1 shows the 13 selected metrics, grouped into three factors: developer activity, project characteristics, and build complexity.

Step 2: Selection of closed-source projects. We selected 18 closed-source projects from a large software company with over 20 years of experience in the market. The company maintains over 1,500 software repositories related to open and closed projects on GitHub. Closed-source projects were selected due to the lack of existing studies investigating CI aspects in industrial settings, and to ensure that we adhered to confidentiality and security policies.

To select the projects, we established specific criteria. First, the project must have been using CI for at least one year to avoid those in their initial level of CI. Second, it must use the company's internal CI tool, which restricted our selection pool. Third, the project must have had at least 700 commits, as this would maximize the number of builds extracted. Last, to filter out inactive projects, we only considered those with at least one commit in October 2021. After applying these criteria, we compiled a list of 30 projects, which we then submitted to two professionals from our industrial partner for validation. After a thorough review, 12 projects were removed from the initial list due to confidentiality and security concerns. The remaining 18 projects were used in the study. Table 2 overviews the selected projects, listing the programming language, number of commits, code lines, age, and the number of successful and failed builds for each project.

Step 3: Automate the collection of metrics and builds. To collect the target metrics and builds, we developed two Python scripts: *Raw Data Extractor* and *Metrics Extractor*. The workflow of our scripts is illustrated in Figure 1.

The *Raw Data Extractor* script processes a software repository, capturing commits from the *master* branch, and stores them in MongoDB. For each captured commit, the script queries the GitHub Status API to check whether the commit triggered a build or not.

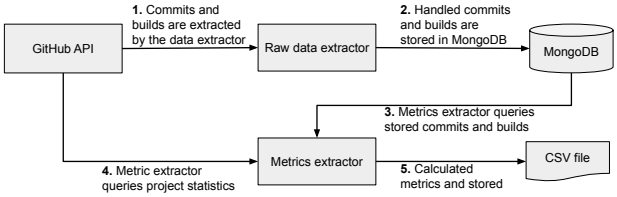

Figure 1: Workflow for collecting metrics and builds

If the commit triggered a build, the information about the build is extracted and saved in the database. After extracting the builds from the project, the *Raw Data Extractor* links the build history and commits using a strategy proposed by Rausch et al. [24], which creates a linear timeline of builds and associated commits.

The *Metrics Extractor* script retrieves failed builds from MongoDB for a given project and calculates the predefined metrics for each of them. The output is a CSV file containing the metric values for all failed builds in the project. In this study, we consider a build as a commit with a defined status, which can be either *success* or *failure*[1]. Thus, the author of a build is the same as the commit that triggered it, and the status of the build is the same as the status of the commit. The build date corresponds to the date on which the status was changed, and the commits of a build are the set of commits between the two commits that triggered it.

Overall, we extracted 10,128 builds from our sample repository, of which 9,715 were successful and 413 failed. We emphasize that we only considered builds with a *status* of *"failure"* as failed, unlike other studies [15, 20] that also include builds with a *status* of *"pending"* or different from *"success"*. Additionally, we acknowledge that the number of failed builds in our sample is relatively low. However, we believe that our data mining approach, which uses association rule mining to uncover patterns in the data, is still suitable for our study's goals, as it can reveal insights even with a small dataset.

Step 4: Labeling metrics to track correlation factors on build failures correction time. In this step, we aimed to better understand the factors influencing build failure correction time by discretizing the values of each metric and creating intervals with well-defined labels based on the quartiles strategy. By labeling the metrics, we can identify the specific factors that contribute to build failures. This labeling also allows us to establish an antecedent-consequent relationship spanning the project's history through association rules. Each transaction of our dataset is represented by a failed build consisting of the metric values from Table 1, and we considered the internal policy of our industry partner, where the developer responsible for breaking the build is also responsible for fixing it. The labels and limits of the intervals used to discretize the values by metric are described in the 3rd column of Table 1.

Step 5: Implementing the association rule mining model. In this step, we used an association rule mining model to find frequent relationships between the three factors – *developer activity, project characteristics,* and *build complexity* – and the duration of build failure correction intervals. We followed a process similar to previous studies [8, 22]. Firstly, we performed data preprocessing, which included treating missing values and discretizing

[1]The build status of the CI internal tool are: success, failure, and pending.

Table 1: Factors and Corresponding Metrics Selected for the Study

ID	Metric	Description	Source
		Developer Activity	
M1	dev_commit_freq	Average number of commits sent by the dev per week in the project repository. Where: *"few commits per week"* = up to 4 commits; *"average number of commits per week"* = between 4 and 7 commits; *"many commits per week"* = more than 7 commits.	[24]
M2	dev_avg_size_commits	Average size in lines of code of commits uploaded by the dev to the project repository. Where: *"short commits"* = up to 48 changes;*"mid commits"* = between 48 and 88 changes; *"long commits"* = more than 88 changes.	New metric
M3	dev_commit_rate	Percentage that the commits sent by the dev to the project repository represent across the entire repository. Where: *"low rate of commits"* = Up to 6%; *"mean rate of commits"* = Between 6 and 17%; *"high rate of commits"* = More than 17%.	[2]
M4	dev_project_time	Difference in days between the current commit and the dev's first commit in the repository. Where: *"short time in project"* = up to 54 days; *"mid time in project"* = between 54 and 174 days, *"long time in project"* = more than 174 days.	[24]
M5	dev_fail_history	Failure rate of builds by current dev in the past of the project. Where: *"few failures"* = up to 5% failure; "mean failures" = between 5 and 16% failure, "many failures" = more than 16% failure.	[25]
M6	dev_exp	Number of triggered builds by dev in the project's past. Where *"few builds"* = up to 24 builds; "average number of builds" = between 24 and 65 builds, *"many builds"* = more than 65 builds.	[25]
		Project Characteristics	
M7	project_size	Project size in lines of code up to the current build. Where: *"small"* = up to 3840 lines of code; *"middle"* = between 3840 and 8273 lines of code; *"large"* = more than 8273 lines of code.	[15]
M8	project_age	Difference in days between the project's first commit and the commit that triggered the current build. Where: *"new"* = up to 522 days; *"intermediate"* = between 522 and 923 days; *"old"* = more than 923 days.	New metric
M9	project_team_size	Number of contributors to the project in the last 60 days. Where: *"small team"* = up to 4 people; *"large team"* = between 4 and 12 people.	[15, 20]
		Build Complexity	
M10	build_qty_commits	Number of commits in the current build. Where: *"few commits"* = up to 18 *commits*, *"many commits"* = more than 18 commits.	[24]
M11	build_mod_files	Number of modified files in the current build. Where: *"few files"* = up to 2 files; *"average number of files"* = Between 2 and 7 files; *"many files"* = more than 7 files.	[24]
M12	build_mod_lines	Number of modified lines in the current build. Where: *"few files"* = up to 11 lines; *"average number of lines"* = Between 11 and 121 lines; *"many files"* = more than 121 lines.	[24]
M13	build_correction_interval	Time in minutes between one failed *build* and the next successful one. Where: *"small interval of correction"* = up to 32 minutes; *"mid interval of correction"* = Between 32 and 220 minutes; *"long interval of correction"* = Between 220 and 1430 minutes.	[20]

Table 2: Overview of the selected closed-source projects

ID	Prog. lang.	#Commits	#LOC	Age	Success	Failures
P1	TS, JS	3,397	15,630	3 years	1633	22
P2	JS	3,337	15,796	5 years	333	16
P3	TS, JS	2,262	11,251	5 years	1119	50
P4	TS	1,989	24,921	3 years	1064	30
P5	TS, JS	1,572	9,606	2 years	687	40
P6	TS, JS	1,378	11557	2 years	602	34
P7	TS	1,224	12,991	3 years	365	9
P8	JS	1,105	18,410	4 years	149	3
P9	TS, JS	1,088	18,892	3 years	169	4
P10	TS	1,054	5,484	3 years	474	5
P11	TS	1,045	5,174	2 years	706	38
P12	TS	1,947	6,777	3 years	424	79
P13	TS	890	1,946	4 years	391	34
P14	TS	825	6,923	4 years	401	2
P15	TS	783	11,324	3 years	121	6
P16	TS	777	5,913	4 years	257	19
P17	TS	758	10,137	2 years	489	12
P18	TS	727	15,174	3 years	331	10

TS = TypeScript, JS = JavaScript

numerical values into categorical ones. During manual analysis of the dataset, we observed some missing values for the metrics DEV_AVG_SIZE_COMMITS and BUILD_CORRECTION_INTERVAL. To address this issue, we replaced missing values with the average value of the metric, following the technique suggested in [13].

Secondly, we used the *Apriori* algorithm and *Association Rules* available in the mlxtend [23] to extract association rules from the discretized dataset. The *Apriori* algorithm identifies *frequent itemsets* in the dataset, while the *Association Rules* extracts association rules from the generated frequent itemsets. We defined minimum values for the support and confidence measures to ensure that the generated rules were not happening randomly. We set a minimum value of 10% for support, representing a total of 41 instances of failed builds, and 30% for confidence. We iteratively defined these minimum values, starting from the default values of the used implementation. The final values were based on the subjective representativeness of observations made on the results and ensured that the created rules were infrequent but present in multiple projects.

It is worth noting that the possibility of some relationships not being detected due to the chosen minimum values is an inherent threat to the validity of studies of this nature. Nonetheless, our observations and the resulting values ensured that our rules could uncover infrequent, yet impactful relationships between factors and build failures' duration correction time.

Step 6: Extract and analyze association rules based on Lift values. We used the support and confidence values previously defined to extract and filter the association rules. To facilitate the interpretation of the results, we limited the rules to contain up to three antecedents, resulting in a total of 2,633 rules. Next, we filtered the rules further to consider only those that have BUILD_CORRECTION_INTERVAL as the consequent, resulting in 151 rules. Finally, we analyzed each resulting rule based on their *Lift* values, which measures the strength of association between antecedents and consequents. We paid particular attention to rules with high lift values, indicating a strong correlation between the antecedents and the build failures' correction time. This analysis allowed us to identify the key factors influencing the build failures' correction time and insights into reducing it in future projects.

4 RESULTS AND DISCUSSION

4.1 Impact of Developer Activity on Fixing Build Failures (RQ_1)

To address $\mathbf{RQ_1}$, we examined the attributes derived from developers' activities in a project repository, which included: (i) the average number of commits submitted per week; (ii) the average size (in lines of code) of their commits; (iii) the commit rate in the repository; (iv) the developer's duration in days on the project; (v) developer experience expressed as the number of triggered builds in the project's past; and (vi) the build failure rate.

Figure 2 shows the *Lift* values for rules of type DEV_COMMIT_FREQ → BUILD_CORRECTION_INTERVAL. It indicates that the average number of commits submitted per week (DEV_COMMIT_FREQ) has a moderate impact on the build failures correction time, particularly when the developer submits a substantial number of commits per week. Moreover, when the author of the failed build is a developer who submitted many commits per week, the chances of obtaining an average correction interval increase by 31% (*Lift* = 1.31).

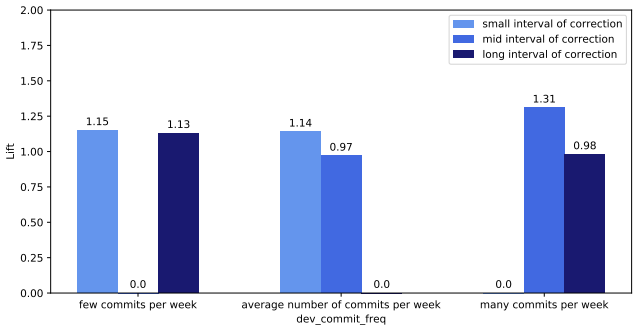

Figure 2: Relationship between commit frequency and build failures correction time

Similar to the DEV_COMMIT_FREQ, the average size of commits submitted by the developer also moderately influences the build failure correction interval. Figure 3 shows the *Lift* values for the association rules where DEV_AVG_COMMITS_SIZE is the antecedent and BUILD_CORRECTION_INTERVAL is the consequent.

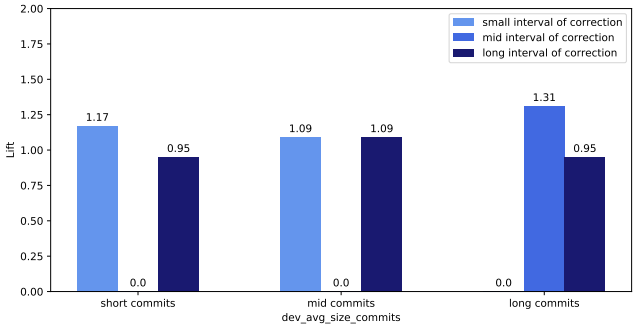

Figure 3: Relationship of average developer commit size to build failures fix time

Based on Figure 3, we can observe that the average size of commits submitted by the developer (DEV_AVG_COMMITS_SIZE) moderately influences the length of the fixed interval for failed builds. Specifically, when the average commit size is long, the chances of an average correction interval increase by 31% (*Lift* = 1.31). Conversely, when the average commit size is short, the chances of a small correction interval increase by 17% (*Lift* = 1.17).

Figure 4 illustrates a moderate influence of the commit rate on the duration of the correction interval, as indicated by the *Lift* values for rules where DEV_COMMIT_RATE is the antecedent and BUILD_CORRECTION_INTERVAL is the consequent.

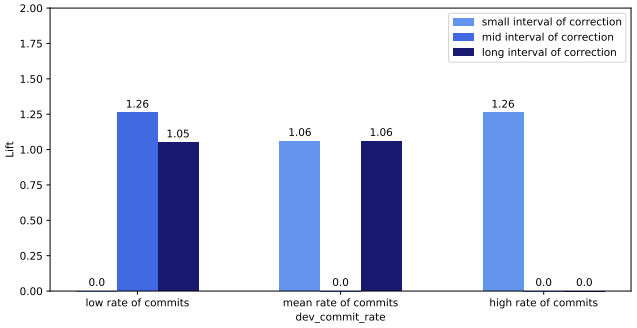

Figure 4: Relationship between developer commit rate and build failures correction time

As observed in Figure 4 also overviews that failed builds authored by developers with a high commit rate have a 26% higher chance of being fixed in a shorter time interval (*Lift* = 1.26). Conversely, when the author of the failed build is a developer with a low commit rate, the chances of the build being fixed in a mid-time interval also increase by 26%. To analyze the relationship between the length of the build failures correction interval and the developer's time on the project, we also considered the rules of type DEV_PROJECT_TIME → BUILD_CORRECTION_INTERVAL. Figure 5 displays the results of this analysis.

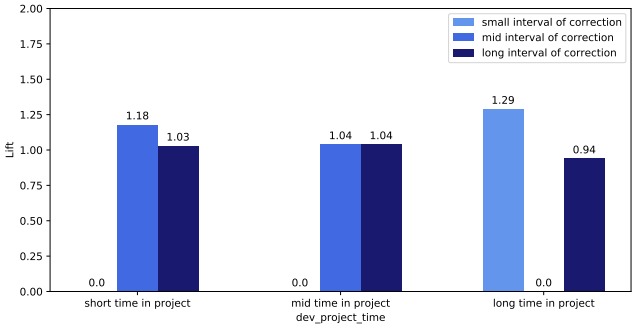

Figure 5: Relationship of developer project time to build failures correction time

As shown in Figure 5, when a developer has been working on the project for a long time, there is a 30% increase in the chances of obtaining a small correction interval. Conversely, when a developer

has been on the project for a short time, there is an 18% increase in the chances of having a mid-correction interval.

Figure 6 shows the relationship between the number of builds executed by the developer in the past (DEV_COMMITER_EXP) and the time to fix build failures. This figure shows the correlations and reveals a possible link between the two variables.

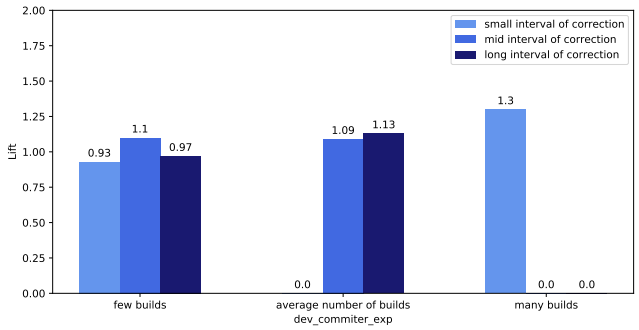

Figure 6: Relationship of developer experience to build failures correction time

Figure 7 presents the relationship between the developer's failure rate and the time to fix build failures. The rules represented in this figure indicate that when the developer has triggered a low number of failures in the past, there is a 21% increase in the chances of achieving a small correction interval. Conversely, when the developer has triggered many failures in the past, there is a 35% increase in the chances of obtaining a mid-correction interval.

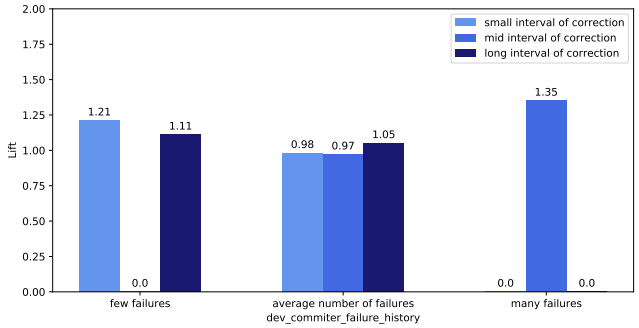

Figure 7: Relationship of developer failure rate to build failures correction time

Summary RQ$_1$: *Developers who submit an average number of commits tend to fix build failures faster than those who submit many commits per week. Keeping commits smaller may also contribute to shorter build failure correction times. Developers who contribute more to the project tend to generate failed builds that are fixed faster, while those who are more familiar with the project tend to fix build failures more efficiently.*

4.2 Impact of Project Characteristics on Fixing Build Failures (RQ$_2$)

A software project's properties, such as the number of code lines, the team's size, and the project's age in days, can affect build failure correction time. In this study, we examined the relationship between these three characteristics and build failure correction time. Figure 8 displays the *Lift* values for rules where *project_size* is the antecedent and *build_correction_interval* is the consequent.

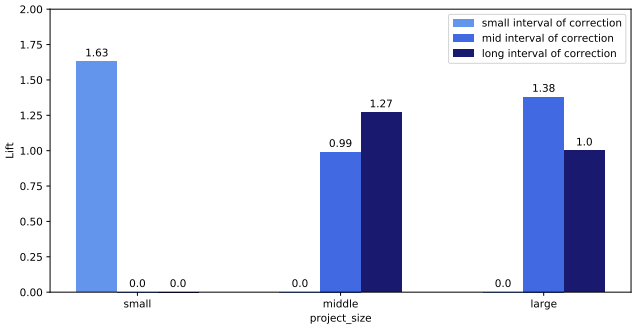

Figure 8: Relationship between project size and build failures correction time

Figure 8 shows the relationship between project size and build failures correction time. As expected, the analysis indicates that when a project has a small size, there is a significant increase in the chances of having a small correction interval, with a lift value of 1.63. This is clear from the rule PROJECT_SIZE = *small* → BUILD_CORRECTION_INTERVAL = *small interval of correction.*

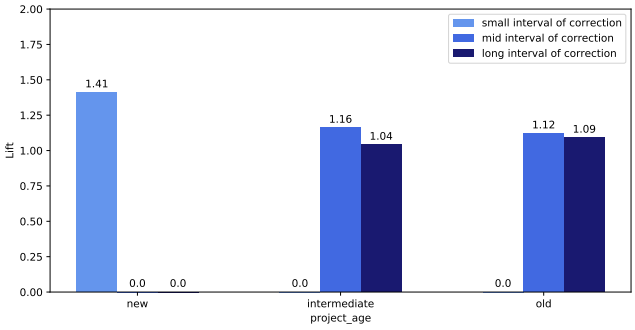

Figure 9: Relationship of project age to build failures correction time

Similarly, Figure 9 reveals that the project age has a similar effect on build failures correction time. Specifically, the analysis shows that when a project is new, there is a 41% increase in the chances of having a small correction interval, with a lift of 1.41. This is evident from the rule PROJECT_AGE = *new* → BUILD_CORRECTION_INTERVAL = *small interval of correction.*

Figure 10 shows the relationship between project team size and build failures correction time. We observed that builds that fail when the project has a large team are 23% more likely to be fixed

Table 3: Rules formed by the conjunction of project attributes

#	Antecedent	Consequent	Lift	Confidence	Support
1	*project_age = new* ∧*project_size = small*	*build_correction_interval = small interval of correction*	1.71	0.57	0.12
2	*project_team_size = small team* ∧ *project_size = large*	*build_correction_interval = mid interval of correction*	1.61	0.53	0.12
3	*project_team_size = large team* ∧ *project_size = small*	*build_correction_interval = small interval of correction*	1.53	0.51	0.1
4	*project_team_size = small team* ∧ *project_size = middle*	*build_correction_interval = mid interval of correction*	1.28	0.42	0.12

in a small interval (*Lift* = 1.23), suggesting a correlation between team size and the duration of the correction interval. However, we cannot conclude that only team size contributes to reducing the time to fix failed builds, as we observed two contradictory rules in terms of confidence.

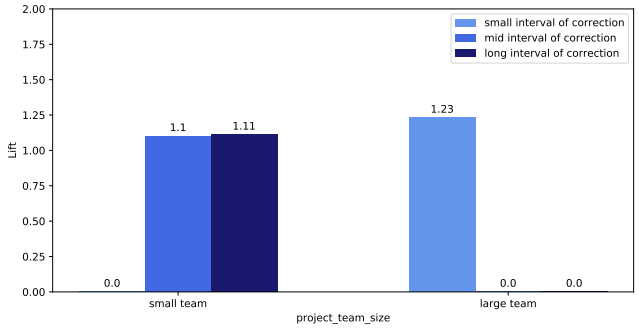

Figure 10: Relationship between team size and build failures correction time

The first rule, with confidence of 41%, is the PROJECT_TEAM_SIZE = *large team* ⟶ BUILD_CORRECTION_INTERVAL = *small interval of correction*, which suggests that a large team can contribute to faster correction times. The second rule, BUILD_CORRECTION_INTERVAL = *small interval of correction* ⟶ PROJECT_TEAM_SIZE = *large team* (confidence = 60%), suggests the opposite. To increase the chances of failed builds having smaller and medium correction intervals, it may be useful to combine multiple project characteristics.

Table 3 presents rules combining project characteristics as an antecedent and the correction interval as a consequent. For instance, the first rule suggests that failed builds occurring when the project is still in the early stages are 71% (*Lift* = 1.71) more likely to have a small correction interval. The third rule suggests that a larger team working on a small project increases by 53% (*Lift* = 1.53) the chance of failed builds taking less time to fix. Finally, the second and fourth rules suggest that a project with a small team size and a large or medium size tends to obtain a medium correction interval.

Summary RQ$_2$: *The analysis indicates that small project size and new project age significantly increase the chances of having a small correction interval. Furthermore, large team size is associated with shorter correction intervals, but contradictory rules suggest that team size may not be the only contributing factor. Combining multiple project characteristics may increase the likelihood of failed builds having smaller correction intervals.*

4.3 Impact of Build Complexity on Fixing Build Failures (RQ$_3$)

In answering **RQ$_3$**, we examined the relationship between the build correction interval and build complexity metrics such as the number of commits, modified files, and modified lines. Figure 11 shows the *Lift* values for rules of the form BUILD_QTY_COMMITS ⟶ BUILD_CORRECTION_INTERVAL. Surprisingly, all rules of this type have *Lift* values close to 1, indicating that there is no significant correlation between the number of commits and the time taken to fix build failures.

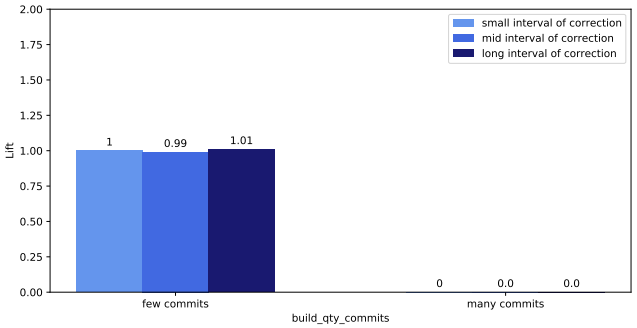

Figure 11: Relationship between the number of commits in the build and the time to fix build failures

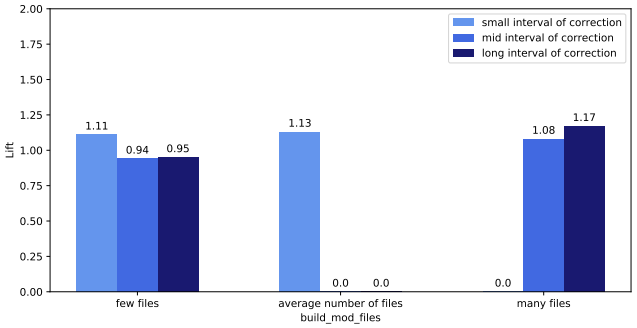

Figure 12: Relationship of the number of modified files in the build to the build failures correction time

Figure 12 presents the *Lift* values for rules linking the number of modified files to the duration of the build correction interval. We can observe that when BUILD_MOD_FILES is equal to *few files*, the chance of obtaining a small correction interval increases by 11% (*Lift* = 1.11), while the chance of obtaining a large correction interval decreases by 5% (*Lift* = 0.95). Conversely, the rule BUILD_MOD_FILES = *many*

files → BUILD_CORRECTION_INTERVAL = *long interval correction* suggests that failed builds with a higher number of modified files are 17% (*Lift* = 1.17) more likely to have a large correction interval.

Based on these observations, we can conclude that the number of modified files slightly contributes to the duration of the build correction interval, with fewer modified files being associated with a smaller correction interval.

Figure 13 presents the *Lift* values for the rules linking the number of modified lines to the build correction interval duration. We can observe that failed builds with BUILD_MOD_LINES = *few lines* are 10% (*Lift* = 1.1) more likely to have a small correction interval, while builds with an BUILD_MOD_LINES = *average number of lines* also contribute to a small correction interval. On the other hand, when many lines are modified, the chances of a mid-correction interval increase. Therefore, we can conclude that the number of modified lines slightly affects the build failure correction time.

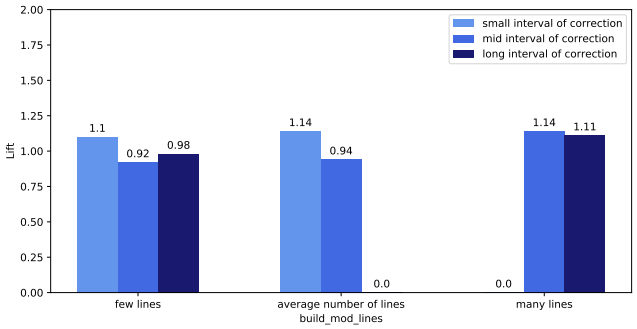

Figure 13: Relationship of the number of modified lines in the build to the build failures correction time

We conducted an analysis to investigate how the conjunction of build characteristics impacts build failure correction time. Table 4 summarizes the main rules we identified by combining values of the build metrics. Overall, the rules in Table 4 suggest that builds with a smaller set of changes tend to have a shorter correction interval, while builds with a larger change set are more likely to have a longer correction interval.

> **Summary RQ$_3$:** *The size of commits involved in failed builds does not significantly affect the time required to fix them. However, reducing the number of modified files and lines may help to decrease the correction interval for build failures.*

5 STUDY IMPLICATIONS

We summarize our main takeaways and discuss their implications for both research and practice as follows.

Developer Activity. Our study reveals that developer activity has a moderate influence on build failure correction time. Specifically, we found that developers with more experience, higher commit rates, and a history of triggering many builds are more likely to resolve failed builds in a shorter time interval. On the other hand, less experienced developers may struggle to resolve build failures on their own and should seek help from more experienced developers. These findings suggest several implications for practice. First, it is essential to provide opportunities for less experienced developers to learn from their more experienced peers. Code reviews and pair programming can help junior developers learn best practices and gain a better understanding of the project's codebase. Second, it is recommended to document common build failure problems and their solutions in a README section of the project repository. This documentation can help all team members quickly resolve common issues and reduce the time to correct build failures.

Project Characteristics. Our study highlights the strong influence of project characteristics on build failure correction time. Specifically, we found that builds triggered in the early stages of a project, with smaller size and age, are more likely to have a shorter correction interval. As the project grows and becomes more complex, teams may face difficulties locating and correcting build failures. These findings suggest that early detection and prevention of build failures are crucial to reduce correction time. Teams can adopt tools that help identify and address issues early in the development cycle, such as static code analysis, automated testing, and continuous integration. Moreover, teams can prioritize the resolution of build failures that occur in critical areas of the system, such as those that impact the user experience or system stability.

Build Complexity. Our analysis shows that build complexity has a moderate effect on the duration of build failure correction time. Specifically, builds with fewer modified lines and files tend to have a smaller correction interval, while those with a larger number of changes are more likely to require a longer time to fix. These findings suggest that reducing the complexity of changes can significantly improve the time to correct build failures. Teams can achieve this by integrating their code frequently and breaking down large changes into smaller, more manageable ones. Additionally, incorporating linting rules into pre-commit scripts can help flag commits that exceed a certain threshold for lines and files, reducing the likelihood of introducing complex changes.

In conclusion, our study provides insights into the factors that influence build failure correction time, highlighting the importance of developer activity, project characteristics, and build complexity. These findings can help inform best practices for teams to reduce build failure correction time and improve overall development efficiency. Future research can explore these factors and their interactions, as well as investigate additional factors that may influence build failure correction time in different development contexts.

6 THREATS TO VALIDITY

Construct and Internal Validity. Our analyses were performed on a set of 18 closed-source projects, which could be a threat. However, our selection was based on defined criteria to find systems (see Step 2 of Section 3.2) relevant to our research questions. We mitigated the occurrence of errors concerning the computation of each of the metrics inserted in the work. The two *scripts* developed to calculate their values were reviewed by a professional with greater experience in software development and an expert in CI.

About the computation of the build correction interval, as the first study that investigates this phenomenon, we have used the time difference between the introduction of the failure and its correction. However, this strategy is inherently noisy. Especially, in the case of developers could have fixed the build many minutes

Table 4: Rules formed by the conjunction of the build attributes

#	Antecedent	Consequent	Lift	Confidence	Support
1	*build_mod_lines = many lines ∧ build_qty_commits = few commits ∧ build_mod_files = many files*	*build_correction_interval = long interval of correction*	1.22	0.41	0.1
2	*build_qty_commits = few commits ∧ build_mod_files = many files*	*build_correction_interval = long interval of correction*	1.21	0.4	0.12
3	*build_mod_lines = many lines ∧ build_qty_commits = few commits*	*build_correction_interval = long interval of correction*	1.15	0.38	0.12
4	*build_mod_lines = average number of lines ∧ build_qty_commits = few commits*	*build_correction_interval = small interval of correction*	1.14	0.38	0.12
5	*build_qty_commits = few commits ∧ build_mod_files = average number of files*	*build_correction_interval = small interval of correction*	1.13	0.38	0.11
6	*build_mod_lines = few lines ∧ build_qty_commits = few commits ∧build_mod_files = few files*	*build_correction_interval = small interval of correction*	1.11	0.37	0.11
7	*build_mod_lines = many lines ∧ build_qty_commits = few commits*	*build_correction_interval = mid interval of correction*	1.11	0.37	0.12
9	*build_qty_commits = few commits ∧ build_mod_files = few files*	*build_correction_interval = small interval of correction*	1.11	0.37	0.14
8	*build_mod_lines = few lines ∧ build_qty_commits = few commits*	*build_correction_interval = small interval of correction*	1.1	0.37	0.13

before they committed the change. To mitigate this threat, we manually validated the computation of build correction interval in a subset of our dataset. Additionally, in the future, we plan to improve this strategy to avoid this type of threat better. Regarding the missing values of certain metrics (DEV_AVG_COMMITS_SIZE and BUILD_CORRECTION_INTERVAL) in our dataset, we have mitigated these cases by replacing the missing values identified for the average of their values. The same strategy was applied by a previous study [24]. However, this strategy may have contributed, even if minimally, to the occurrence of some association rules.

Conclusion and External Validity. All results were double-checked by two paper authors, to mitigate biases and the misapplication of procedures. A threat to the conclusion is the possible small number of projects used in the study due to the difficulty of selecting closed-source projects. Additionally, the small set of selected metrics for each analyzed factor could also be considered a threat to conclusion validity. However, the metrics were selected based on prior studies [2, 15, 20, 24, 25]. Our results are restricted to the projects of our industrial partner that has similar characteristics to those selected for this study. A threat to external validity is the limited generalizability of our findings to other projects outside the scope of this study. We only consider failure builds that had a status equal to failure, instead of considering as failed all builds that have a status other than success [15, 20].

7 RELATED WORK

CI and Pull Requests. Bernardo et al.[5] investigated whether adopting CI in a project improves pull request delivery time by analyzing 162,653 pull requests from 87 Github projects. The study found that only 51% of the projects showed a reduction in the pull request delivery time after CI adoption. Additionally, the study found an increase in the number of pull requests submitted after CI adoption, which is the main reason for projects to deliver pull requests more slowly after CI adoption. Similarly, Zampetti et al.[32] empirically analyzed the impact of CI build results on pull request discussions and found that the result exerts limited influence on the approval of a pull request. While a successful result increases the chances of a pull request being approved, other process-related factors have greater importance in the decision to accept or reject a pull request. Gallaba et al.[11] conduct an empirical study of 22.2 million builds spanning 7,795 open-source projects that used CircleCI from 2012 to 2020. The authors found is that availability issues, configuration errors, user cancellation, and exceeding time limits are key reasons that lead to premature build termination.

CI build failures. Studies have also investigated the impact of CI build failures on open and closed-source projects. Islam and Zibran [15] studied the factors that cause build failures and identified that the complexity of a task and the build strategies and development model strongly affect the failure of a build. Kerzazi et al.[20] found that the developer's role, the number of collaborators on the branch, and the nature of the work are factors that strongly predict the occurrence of building breakage. Rausch et al.[24] analyzed data from 14 open-source Java-based projects hosted on GitHub and identified that failed integration tests, sub-tolerable code quality metrics, and compilation failures are among the most common build failures. In addition, recent build stability, change complexity, and author experience were found to be influential factors in predicting the build outcome. To address the problem of build failures, Saidani et al.[25] developed and evaluated a build failure prediction model based on Multi-Objective Genetic Programming. They found that project statistics such as development team size, last build information, and changed file types are the most important factors in predicting build failure. In a subsequent study, Saidani et al.[26] proposed a new approach for predicting build failures using a deep learning model based on Long Short-Term Memory, which proved to be more efficient than traditional machine learning models.

Barrak et al. [3] performs a conceptual replication study on 27,675 Travis CI builds of 15 GitHub projects. The results identified that features from the build history, author, code complexity, and code/test smell dimensions are the most important predictors of build failures. Jin and Servant [19] conducted three empirical studies to enhance the observation of build failures and reduce computational costs in CI. Their approach outperformed the existing safest technique by saving more costs (5.5%) and virtually eliminating falsely skipped failures builds (reduced from 4.1% to 0% median value). Most of the related studies investigated CI problems in the context of open-source projects. Our study focused on the analysis of CI build failures in closed-source projects. Our study differs from other studies because we aim not to predict build failures but to understand the factors that influence the variation of the time interval needed to correct build failures. For this, we use association rule mining to find correlations between the characteristics of developers, projects and builds with the duration of the correction interval.

8 CONCLUSION AND FUTURE WORK

In this study, we have investigated key factors that influence the build failure correction time in 18 closed-source projects. Our findings shed light on the importance of developer activity, project characteristics, and build complexity in the build failure correction process. We quantified these factors through 13 well-known properties and found that the developer activity and project characteristics moderately to strongly influence the duration of the build failure correction time, respectively. Additionally, we observed that more complex builds, with increased lines of code and modified files, tend to have longer correction times.

Additionally, there is still room for improvement and future work in this area. We plan to expand our analysis by adding new metrics to measure other properties of the three factors analyzed in this study. We also aim to conduct a qualitative study to understand better the reasons underlying the influence relationships we have discovered. We also plan to replicate our study with open-source projects hosted on GitHub to verify our findings are generalizable to the open-source context. Finally, we intend to explore other factors like test coverage, team distribution, and CI tooling. As well as exploring whether the addition of bad CI practices contributes to longer build failure correction times.

ACKNOWLEDGMENTS

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001; FUNCAP (BP5-00197-00042.01.00/22); and FAPESB (PIE0002/2022).

REFERENCES

[1] Rakesh Agrawal, Ramakrishnan Srikant, et al. 1994. Fast algorithms for mining association rules. 1215 (1994), 487–499.

[2] Eman Abdullah AlOmar, Anthony Peruma, Christian D. Newman, Mohamed Wiem Mkaouer, and Ali Ouni. 2020. On the Relationship Between Developer Experience and Refactoring: An Exploratory Study and Preliminary Results. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops* (Seoul, Republic of Korea) *(ICSEW'20)*. Association for Computing Machinery, New York, NY, USA, 342–349.

[3] Amine Barrak, Ellis E Eghan, Bram Adams, and Foutse Khomh. 2021. Why do builds fail?—A conceptual replication study. *Journal of Systems and Software* 177 (2021), 110939.

[4] Moritz Beller, Georgios Gousios, and Andy Zaidman. 2017. Oops, My Tests Broke the Build: An Explorative Analysis of Travis CI with GitHub. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. 356–367.

[5] João Helis Bernardo, Daniel Alencar da Costa, and Uirá Kulesza. 2018. 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* (Gothenburg, Sweden) *(MSR '18)*. Association for Computing Machinery, New York, NY, USA, 131–141.

[6] Victor R Basili-Gianluigi Caldiera and H Dieter Rombach. 1994. Goal question metric paradigm. *Encyclopedia of software engineering* 1, 528-532 (1994), 6.

[7] Bihuan Chen, Linlin Chen, Chen Zhang, and Xin Peng. 2020. BuildFast: history-aware build outcome prediction for fast feedback and reduced cost in continuous integration. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 42–53.

[8] Daniel Coutinho, Anderson Uchôa, Caio Barbosa, Vinícius Soares, Alessandro Garcia, Marcelo Schots, Juliana Alves Pereira, and Wesley K. G. Assunção. 2022. On the Influential Interactive Factors on Degrees of Design Decay. In *Proceedings of the 29th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), 2022, Honolulu, Hawaii, March 15–18*. 1–12.

[9] Léuson Da Silva, Paulo Borba, and Arthur Pires. 2022. Build Conflicts in the Wild. *J. Softw. Evol. Process* 34, 4 (apr 2022), 28 pages.

[10] Wagner Felidré, Leonardo Furtado, Daniel A. da Costa, Bruno Cartaxo, and Gustavo Pinto. 2019. Continuous Integration Theater. In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 1–10.

[11] Keheliya Gallaba, Maxime Lamothe, and Shane McIntosh. 2022. 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* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 1330–1342.

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

[13] J. Han, J. Pei, and M. Kamber. 2011. *Data Mining*. Elsevier Science.

[14] Foyzul Hassan. 2019. Tackling Build Failures in Continuous Integration. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering* (San Diego, California) *(ASE '19)*. IEEE Press, 1242–1245.

[15] Md Rakibul Islam and Minhaz F. Zibran. 2017. Insights into Continuous Integration Build Failures. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. 467–470.

[16] Gustavo Ivens, Carla Bezerra, Anderson Uchôa, and Ivan Machado. 2023. Replication package for the paper: "What Factors Affect the Build Failures Correction Time? A Multi-Project Study". https://figshare.com/s/a6fc32d2dcb1194b6467 Accessed: 2023-07-23.

[17] Xianhao Jin and Francisco Servant. 2020. Evaluation of the Strategiesto Building in Continuous Integration. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 13–25.

[18] Xianhao Jin and Francisco Servant. 2021. What Helped, and what did not? An Evaluation of the Strategies to Improve Continuous Integration. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 213–225.

[19] Xianhao Jin and Francisco Servant. 2022. Which builds are really safe to skip? Maximizing failure observation for build selection in continuous integration. *Journal of Systems and Software* 188 (2022), 111292.

[20] Noureddine Kerzazi, Foutse Khomh, and Bram Adams. 2014. Why Do Automated Builds Break? An Empirical Study. In *2014 IEEE International Conference on Software Maintenance and Evolution*. 41–50.

[21] Mathias Meyer. 2014. Continuous Integration and Its Tools. *IEEE Software* 31, 3 (2014), 14–16.

[22] Daricélio Moreira Soares, Manoel Limeira de Lima Júnior, Leonardo Murta, and Alexandre Plastino. 2021. What factors influence the lifetime of pull requests? *Software: Practice and Experience* 51, 6 (2021), 1173–1193.

[23] Sebastian Raschka. 2018. MLxtend: Providing machine learning and data science utilities and extensions to Python's scientific computing stack. *The Journal of Open Source Software* 3, 24 (April 2018).

[24] Thomas Rausch, Waldemar Hummer, Philipp Leitner, and Stefan Schulte. 2017. An Empirical Analysis of Build Failures in the Continuous Integration Workflows of Java-Based Open-Source Software. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. 345–355.

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

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

[27] Ruben Blenicio Tavares Silva and Carla I. M. Bezerra. 2020. Analyzing Continuous Integration Bad Practices in Closed-Source Projects: An Initial Study. In *Proceedings of the 34th Brazilian Symposium on Software Engineering* (Natal, Brazil) *(SBES '20)*. Association for Computing Machinery, New York, NY, USA, 642–647.

[28] P.N. Tan, M. Steinbach, and V. Kumar. 2014. *Introduction to Data Mining*. Pearson.

[29] Bogdan Vasilescu, Yue Yu, Huaimin Wang, Premkumar Devanbu, and Vladimir Filkov. 2015. Quality and Productivity Outcomes Relating to Continuous Integration in GitHub. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (Bergamo, Italy) *(ESEC/FSE 2015)*. Association for Computing Machinery, New York, NY, USA, 805–816.

[30] Carmine Vassallo, Sebastian Proksch, Timothy Zemp, and Harald Gall. 2020. Every Build You Break: Developer-Oriented Assistance for Build Failure Resolution. In *Empirical Software Engineering*, Vol. 25. Springer, 2218–2257.

[31] Carmine Vassallo, Gerald Schermann, Fiorella Zampetti, Daniele Romano, Philipp Leitner, Andy Zaidman, Massimiliano Di Penta, and Sebastiano Panichella. 2017. A Tale of CI Build Failures: An Open Source and a Financial Organization Perspective. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 183–193.

[32] Fiorella Zampetti, Gabriele Bavota, Gerardo Canfora, and Massimiliano Di Penta. 2019. A Study on the Interplay between Pull Request Review and Continuous Integration Builds. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 38–48.

[33] Chen Zhang, Bihuan Chen, Linlin Chen, Xin Peng, and Wenyun Zhao. 2019. A Large-Scale Empirical Study of Compiler Errors in Continuous Integration. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. 176–187.