European Journal of Information Technologies and Computer Science
Vol 4 | Issue 2 | June 2024
ISSN 2736-5492

RESEARCH ARTICLE

On Association of Code Change Types and CI Build Failures in Software Repositories

Samiha Shimmi* and Mona Rahimi

ABSTRACT

The software development process heavily relies on building systems, which are prone to frequent failures, particularly in continuous integration (CI) environments. In this study, we investigated the impact of major change types, both individually and collectively, on CI build failure rates. Specifically, we compared the contribution of changes stemming from different underlying reasons, such as functional requirement additions, bug fixes, enhancements, and dependency removals. Preliminary results revealed that adding new functionalities had a lower impact on CI failures compared to maintenance changes. Furthermore, we analyzed the characteristics of the ultimate changes to identify common features among the change types that contributed to failures. Subsequently, utilizing these identified features, we developed a mathematical model to predict failures based on the characteristics of the triggering change type. The trained model demonstrated a commendable performance, accurately identifying potential failure-inducing changes in the dataset, with a recall of 78% and precision of 53%. This research sheds light on the relationship between change types and CI build failures, highlighting the significance of maintenance changes in driving failures. The identification of common features among failure- contributing change types aids in understanding failure patterns and supports the development of preventive measures. The predictive model offers a practical tool for early detection and mitigation of potential failures, contributing to improved software development processes and the adoption of effective CI practices.

Keywords: Build failure, Continuous integration, Mining software repository.

Submitted: April 17, 2024

Published: June 30, 2024

10.24018/ejcompute.2024.4.2.124

Department of Computer Science, Northern Illinois University, USA.

******Corresponding Author:*
e-mail: sshimmi@niu.edu

1. Introduction

Each type of software maintenance, including *corrective, preventive, perfective*, and *adaptive*, is performed throughout a software lifespan with a variety of motivations [1], [2]. Corrective maintenance is often due to bug reports submitted by the users of the product, while preventative activities refer to the future so that the software continues to work as desired for as long as possible. Examples include making necessary changes, upgrades, adaptations, and more. Preventive software maintenance addresses minor issues, which at a given time may lack significance yet may turn into a major problem in the near future. These are known as the software's latent faults, which require to be detected and corrected to prevent their transformation into effective faults. Perfective changes often occur due to users' need for new features or requirements, which are requested to be incorporated into the software to enhance the software into the potentially best tool available for the user's needs. Finally, Adaptive software maintenance addresses changing technologies as well as policies and rules regarding your software. For instance, changes in the operating system, cloud storage, or adjacent hardware components [2].

In an environment with continuous integration (CI) practices, software is maintained by several developers, who frequently commit their code changes into a shared repository to be automatically integrated into a single project, tested, and eventually built [3]. In a CI process, a significant number of heterogeneous and potentially inconsistent changes are continuously integrated and built, leading to *build break* being the most common cause of failures [4], [5]. Once a building procedure breaks, the software development process comes to a halt, especially in collaborative and agile environments, where fixing the build suddenly becomes a top priority [6]–[8]. Build failures slow down the product's release pipeline, decrease team productivity, and increase software production costs [9], [10]. A study reported a software project in which total build failures aggregated to a cost of more than 2,000 man-hours [11].

A commit build may break for several reasons, such as compilation errors or test failures. While less frequent changes to the repository reduce the process overhead caused by the build failures, delayed integration of the changes complicates troubleshooting and repair, yielding several conflicts in the system. Resolving complex conflicts eventually adds a larger overhead to the process than repairing the build failures of smaller changes. Hence, taking preemptive action seems necessary to minimize the frequency of build failures. The prediction of the issue types, which are more prone to generate build failures, helps to take preventive actions in regard to the build failures.

To address the problem of build failure, several research groups aimed to identify the most influential underlying roots of build breaks [4], [6], [9], [10], [12], [13]. Some studies developed tools and plugins to support fixing build failures and make the recovery process faster and more efficient [14]–[16]. For instance, among them, Beller *et al.* [16] proposed TravisTorrent to process and analyze build reports specific to Travis CI, from GitHub commits and extract more information about the failures that occurred.

While our ultimate objective aligns with minimizing CI failures, our approach revolves around predicting the likelihood of build failures. In contrast to the previously mentioned work, this research brings attention to the contributions of underlying change factors to these failures. Therefore, our work has a *dual purpose*. Firstly, we aim to investigate the specific issue types requested by end users (or developers) that primarily trigger CI build failures. Additionally, as each issue type necessitates distinct changes in the codebase, the second aspect of our research focuses on analyzing the characteristics of the resultant code changes, irrespective of their specific triggers. The goal is to identify *common features among the failure-inducing code changes that lead to failures across all user-requested issue types*.

The reason is to further exploit the features of the user-reported issues, as well as their potential subsequent change impacts in the code to flag the potential failure-inducing user requests as they are posted and before their occurrence. For this, we trained multiple binary classifiers, which assess the possibility of training a model for CI failure predictions according to the issue's characteristics.

The prediction of CI build failure reveals, in advance, the critical issues that yield changes that are more prone to failure once they are built in CI. Such warnings alert developers to prepare before the product is built and to take preventive measures. Therefore, the focus of this research will be on the issue types and issue-related metrics that contribute and, therefore, allow the detection of failure-inducing builds. For instance, are the perfective maintenance activities more prone to CI build failures than the corrective changes? In particular, this work answers the research questions below:

- *RQ_1*: What are the specific issue types requested by users (or developers) that more significantly contribute to CI build failures?
- *RQ_2*: What are the shared characteristics and features of code changes that lead to build failures across different user-requested issue types?
- *RQ_3*: How can a predictive model be developed to estimate the likelihood of CI build failures based on the identified issue types and their common features?

The first question, therefore, seeks to identify whether or not a change type is more likely to result in CI build failure. For instance, are fixing bugs in the code or fixing code dependencies more likely to trigger the CI build failures? How about code changes related to incorporating a new requirement or code improvement changes (e.g., refactoring)? Within each change type, once individually and once again jointly, the second question studies the (common) features of those changes which led to the failure. For instance, do changes whose related issue was of critical priority more often lead to the CI build failure, in comparison to the issues with *minor* priority? What about the issues which were *re-opened* or those which already had a *patch available*? The third research question explores the possibility of building predictive models to predict build failures according to the identified features in the previous research question.

The significant contributions of this paper are of great importance for the field, as they provide valuable insights and practical solutions for addressing CI build failures. Specifically, the paper focuses on three primary contributions:

- *Identification of issue types and their common features:* One of the main challenges in studying CI build failures is the vast amount of data generated from different types of issues. This paper successfully identifies the types of issues that contribute the most to CI build failures and extracts their common features. By doing so, researchers and practitioners can select a more efficient set of attributes for classification algorithms in CI failure studies. This not only improves the accuracy of classification models but also reduces the computational burden associated with analyzing large datasets, leading to more effective and targeted investigations.
- *Analysis of code changes and failure-inducing characteristics:* The study investigates the characteristics of code changes that result in build failures across different user-requested issue types. By examining the common features among these failure-inducing code changes, independent of their underlying reasons, the research provides insights into the shared characteristics that contribute to build failures. This analysis enhances the understanding of failure patterns and assists in developing more effective strategies for preventing similar failures in the future.
- *Prediction of CI build failures*: The ability to predict CI build failures is crucial for proactive problem-solving and minimizing the impact of failures.

By leveraging the identified issue types and their common features, this paper develops a predictive model that can anticipate potential build failures. This proactive approach enables teams to address the underlying issues before they manifest as actual build failures. For example, the model can assign problematic issues to more experienced developers or trigger a review process for critical components. By taking early action, developers can mitigate the risks associated with build failures, streamline the development process, and ultimately deliver higher-quality software.

The findings of this paper not only contribute to the advancement of research in CI failure analysis but also provide practical implications for industry practitioners. The identification of issue types and their common features helps optimize the selection of attributes in classification algorithms, leading to more accurate and efficient analyses. Furthermore, the prediction of CI build failures empowers teams to take proactive measures and prevent potential failures, thereby improving the overall software development process. The insights presented in this paper have the potential to enhance the reliability, efficiency, and effectiveness of CI systems, benefiting both researchers and practitioners in the field.

The rest of this paper is structured as follows. Section 2 contains the background required for the work, including information about our dataset. Section 3 provides a partial answer to RQ_1, while Section 4 tends to mathematically measure the significance of the observed patterns. Further, Sections 5 and 6 respond to RQ_2 and RQ_3, respectively. Sections 7 and 8 discuss the threats to the validity of our studies and the major related work to the research topic. Finally, the observations are concluded, and future directions are shared in Section 9.

2. Background

This section provides the background knowledge required to better read the rest of the paper.

2.1. Travis CI

Continuous integration is a software development practice where developers regularly merge their code changes into a central repository, after which automated builds and tests are run. A build pipeline can consist of several tasks, but at a minimum, three phases should be included:

1. Compiling code;
2. Executing tests, such as unit and integration tests;
3. Deployment, which contains packaging the complied code into artifacts (e.g., jar files) and deploying them.

Similarly, Travis CI provides a build environment for a repository, cloned into it, and then executes a set of build phases specified in a configuration file in .yml format. For instance, once linked to GitHub, Travis monitors the repository. Once a new pull request is opened, Travis receives a notification and executes the steps of the build pipeline as defined in its configuration file. If any of the build steps fails, the pipeline terminates and notifies users that the build is broken.

Travis includes multiple features that made the environment the developers' preferred option to start with build pipelines. For instance, Travis integrates with GitHub repositories, deploys to multiple cloud platforms, and supports different programming languages.

An overview of Travis build states is demonstrated in Fig. 1. As shown in Travis a build process may have one of the three final states:

- *Build Errored:* Once a build process breaks before the execution of build commands in *script* (e.g., due to dependency or dependency installation failures), the build is errored.
- *Build Failed:* Once build commands in Travis *script* fail or time out, the process still continues before the build is marked as failed.
- *Build Passed:* Once the execution of the entire build job is completed the status of the build is marked as successful.

In addition to the three statuses discussed above, canceled build status can occur in any phase and is triggered with an outside command.

2.2. SmartSHARK Dataset

For our analysis, we used a publicly available dataset, namely SmartSHARK, release 2.2 [17], integrating a collection of various software projects and their evolving artifacts from publicly available sources like GitHub, Travis CI, JIRA issue tracker [18] along with a number of analysis on the data. The dataset consists of 98 Apache project data, stored in MongoDb database, including 452,286 commits, 207,400 issues, 52,478 pull requests, and 89,160 Travis build records.

There are two versions of the dataset available, one being larger than the other version including code clone and multiple software metrics as well. We used the smaller dataset containing 36.3 GB of data, including the same

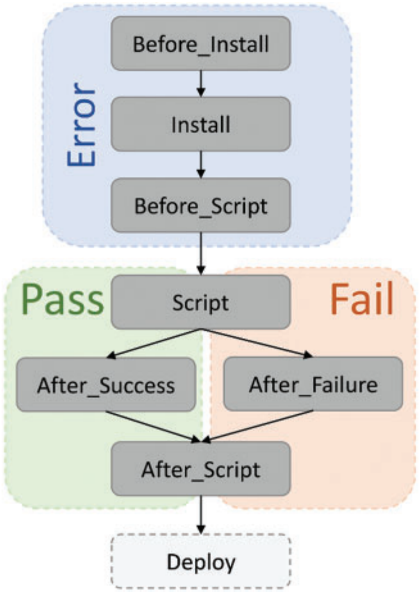

Fig. 1. Travis CI build process with final states of errored, passed, and failed.

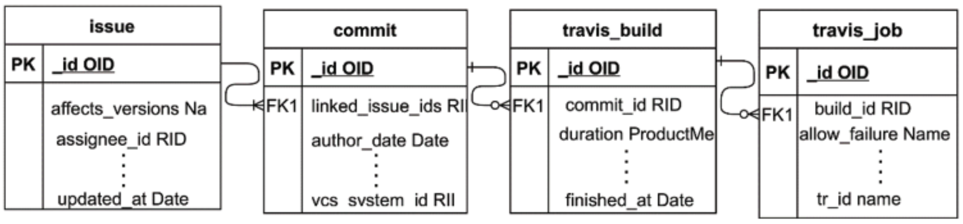

Fig. 2. Partial ER diagram of SmartSHARK database relevant to this work.

information of the entire projects in the larger dataset except for the code and metrics, not being required in this work.

The portion of the database schema, that we used here, is shown in Fig. 2 in the form of an E-R diagram. Each collection in the Figure contains several additional fields in the database, but due to space limitations, only a few of the fields and the relationship between the collections are demonstrated. The complete SmartSHARK schema definition, along with their documentation, is available in [19].

3. Whom to Blame in CI Build Failures?

This section provides a partial answer to RQ$_1$, while the following section provides statistical proof for this research question.

We initially analyzed the available issue types in the collection, identifying 31 unique types of issues among the 98 projects. Among the issue types, *new functionality*-, *bugs*-, and *improvement-related* issues contained the largest number of records, including 13,216, 108,137, and 49,512 issues, respectively. We additionally looked into three additional issue types to be included in the study, including task, sub-task, and issues related to dependencies consisting of 11,367, 11,755, and 2,178 records in the database, respectively. However, in our further manual analysis of several records for each issue type, we found significant inconsistencies among the types of issues labeled as "task" and "sub-task," leading us to exclude both these types from the study while including the dependency-related issues.

As such, the final types were selected from four major issue types, whose triggering reason is rooted in explicitly different classes of change. The first type includes the addition of new features to the software product (perfective maintenance), integrating new source code into the framework (requirements). The second is related to fixing product bugs (corrective maintenance) and integrating corrected source code (bug). The purpose of the third type is to improve the product (preventive maintenance), integrating both new and corrected source code (improvement). Finally, the last group is related to maintaining the present dependencies (dependency), such as upgrading the libraries that are in use by the software and addressing the issues that happened due to a change in frameworks (adaptive maintenance).

For the selected issues, the 'commit' collection, where all commit-related information is stored, was searched to identify the commits traced to each issue. As such, only a total of 5,286 issues were found to be traced to the commits, while the rest of the issue ids were not present in the collection. This could be due to the missing issue-commit trace links or due to not yet implemented new features. In either case, this study did not include the issues that were not linked to any commit in the database.

A total of 14,466 commits were retrieved through developing a query, searching the 'linked issue ids' field of the commit collection for the 'id' value of the 'issue' collection. Later, for the selected commits, the 'Travis build' collection of the schema was searched, resulting in the return of 901 builds linked to the commits, as shown in Fig. 2. The process of extracting the three artifact types, issues, commits, and Travis build, took about 3 hours on a machine with Intel core i7 1.80 GHz CPU with 16 GB RAM.

The entire process was then automatically repeated, taking 24 hours for issues with the type 'bug fix,' 12 hours for type 'improvement,' and 1 hour for type 'dependency' tags. Table I shows the counts of the artifacts issues, traced issues, commits, and builds for each issue class. In case of dependency-related issues, we merged the two issue types of 'Dependency' and 'Dependency upgrade' as both refer to the same type of issue.

As shown in Table I, the number of identified issues related to bug fixes was significantly higher than those relevant to the new features, with a total number of 108,137, among which 49,583 were traceable in the commit collection. Since several issues were linked to more than one commit, there were 94,324 commits in total related to the bug fix issues. Among them, only 7,553 commits relating to 4,693 issues were traced in the Travis build records. This happens since the relevant commits, addressing the same concern, are often merged into a larger group of changes to be built at once instead of re-building the entire system for each individual commit. The last three rows of the table show the number of failed, passed, and errored builds for each issue type. The total build number is larger than the value in the *build counts related to issues* in the sixth row since sometimes the unsuccessful builds were repeated multiple times before they succeeded. Fig. 3 represents a

TABLE I: The Counts of Commit and Build Artifacts for Each Issue Type

Statistics	Reqrm.	Bug	Imp.	Dpn.
Total issues	13,216	108,137	49,512	1,798
Distinct issues in commits	5,286	49,583	23,758	318
Commits related to issues	14,466	94,324	52,646	584
Build count related to commits	901	7,553	4,731	37
Build count related to issues	471	4,693	2,793	25
Failed issue-related builds	459	3,661	2,437	16
Passed issue-related builds	196	1,895	1,251	6
Errored issue-related builds	223	1,785	948	0

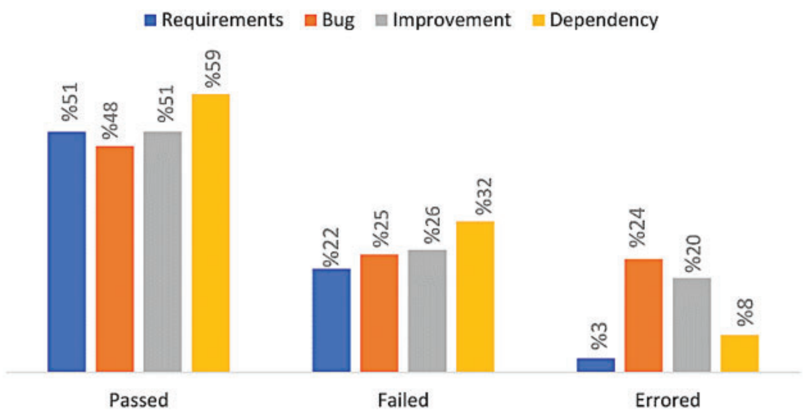

Fig. 3. Histogram showing % of failed, passed and errored verdicts for each issue type.

histogram of the failed, passed, and errored build rates per issue for each change type. Builds with other verdicts, such as canceled, are not shown here. As shown, a trivial observation represents that the bug-fixing changes lead to the majority of the CI Failures, yet further analysis is required to draw any conclusion.

4. RQ1 Statistical Analysis: Primary Change Type in CI Build Failures

To study the significance of our observations in the previous section, and to answer our first research question, we conducted a statistical analysis. As discussed earlier, RQ_1 attempts to study the significance and strength of the relation between the issue types and CI build final verdict.

4.1. Chi-Square Test

Since the dataset majorly contains nominal data, in order to use the same statistical equation for all the variables, we converted the commits and builds counts from the count scale to the four nominal scales. This type of conversion results in type consistency within the analysis and therefore allows us to measure and interpret the same metric for the entire variables in order to conduct a more fair comparison.

To assess the contingency between the nominal values, one commonly used test is *Pearson Chi-square statistical test*, intended to test how likely it is that an observed distribution is due to chance [20]. In this study, we adopted Chi-square test to assess the significance of relation, as well as *Cramer's V metric* to measure the strength of the significant relations [21]. We applied Pearson Chi-square test to assess the independence of each variable pair with each other. The *V value* is one measure to interpret the relation between nominal values for the Chi-squared test. To measure the correlation, the V coefficient divides the square root of the Chi-squared statistic by the sample and the feature size.

In addition, Chi-square distribution is accompanied by a parameter called *degree of freedom (df)*, representing the number of independent variables within which the Chi distribution is calculated. *df* estimates the number of the variables in the calculation that are free to vary, while the rest of the vectors are constrained to lie in the samples Chi subspace [22]. The *df* can be considered as the minimum number of the vectors which is required to define the sample data in Chi subspace and can be calculated by subtracting the number of Chi-estimated parameters from the total number of values in the sample. A small *df* between an issue feature and the dependent variable (build verdict) determines that a smaller dimension is required to define the samples with respect to this feature. Furthermore, Pearson Chi-squared test requires a minimum expected count of observations to be satisfied in all cross values of the two variables in the test. This means once a certain percentage of cross-value counts does not contain the Chi-expected number, then the result is not valid for the pair. This does not necessarily reject the correlation between the two variables, but rather states that more samples of a specific cross-value are required for the test to draw a conclusion. In the literature, once 50% or more of the cross-values counts are less than the Chi expected value, then the test is considered invalid [23], [24].

4.2. Results and Discussion

To assess the significance of the observations in Fig. 3, we need to show that the differences in the verdicts, which we observed between the issue types and their build final states (i.e., verdict), are statistically sufficient.

To indicate that the findings in Fig. 3 are not due to chance and are, therefore, statistically significant, we conducted a Chi-square test. The final verdict of the issues' build is considered as the dependent variable, which takes the values of *failed* and *passed*, and the nominal type categories as the independent variable. The results of the test will determine whether or not a statistically significant relationship is present between the failures and change types. As the results in Table II demonstrate, the significance value of 0.001, which is smaller than our commonly selected value of 0.05, verifies that an association between the issue types and CI build verdicts exists, which is mathematically significant.

RQ_1: Based on the empirical data analysis, changing the existing source code for the purpose of resolving issues relevant to bugs leads to the largest percentage of the CI builds being unsuccessful. The Chi-square statistically verified the significance of the association between the issue types and CI build verdicts with $p-value < 0.05$.

The findings from RQ1, which indicate that changing the existing source code to resolve bug-related issues contributes the most to unsuccessful CI builds, have several

TABLE II: RQ_1: Count and Chi-Square Test Results for Change Types

Statistics	Reqrm.	Bug	Imp.	Dpn.	Total
Verdict failed	228	2,216	1,453	15	3,912
Verdict pass	243	2,477	1,340	10	4,070
Total	471	4,693	2,793	25	7,982
	Value	df	Asymptotic significance		
Pearson chi-square	17.444	3	<0.001		
N of valid cases	7,982	–	–		

takeaways and implications for the software engineering or CI community, developers, and researchers. These include:

- *Prioritizing bug resolution:* The findings underscore the critical importance of effectively addressing bugs in the source code. Developers and software engineering teams should prioritize bug-fixing activities and allocate sufficient resources to ensure timely and accurate resolution. Emphasizing the importance of bug detection, prevention, and resolution can lead to a more stable and reliable CI process.
- *Quality assurance and testing:* The high percentage of unsuccessful CI builds associated with bug-related code changes highlights the need for rigorous quality assurance and comprehensive testing practices. Developers should focus on implementing robust testing strategies, including unit testing, integration testing, and regression testing, to identify and rectify bugs before they impact the CI builds. This emphasizes the significance of automated testing frameworks and continuous testing processes.
- *Code review and collaboration:* The findings emphasize the importance of code reviews and collaboration among developers. Thorough code reviews can help identify and address potential bugs and issues before the code changes are integrated into the CI pipeline. Collaboration and knowledge sharing among team members, particularly in bug resolution efforts, can enhance the overall quality of code changes and reduce the likelihood of build failures.
- *Continuous improvement:* The empirical evidence highlights the need for continuous improvement in software engineering practices. It is essential for developers and software engineering teams to continuously analyze and monitor the impact of code changes on CI builds. This enables them to identify patterns and trends related to specific issue types and take proactive measures to optimize the development process and minimize build failures.
- *Research implications:* The statistically significant association between issue types and CI build verdicts, as verified by the chi-square test, provides valuable insights for researchers. This finding encourages further investigation into the underlying factors that contribute to unsuccessful builds, exploring additional dimensions beyond code changes, such as build configurations, testing environments, and team dynamics. Researchers can develop more sophisticated models and approaches to predict and prevent build failures based on these insights.

In summary, the findings from RQ_1 highlight the impact of bug-related code changes on CI build failures. These findings emphasize the importance of bug resolution, quality assurance practices, code reviews, collaboration, and continuous improvement in the software engineering and CI community. They also provide valuable implications for researchers to delve deeper into the causes and prevention of build failures in the context of issue types and other relevant factors.

5. RQ_2 Statistical Analysis: Primary Features of Change in CI Build Failures

This research question attempts to study whether a common feature potentially exists between the features of changes that lead to the CI Failures.

5.1. *Issues' Profiles*

To conduct an analysis on the issues, we initially built a dataset by creating a profile for each issue. We investigated the relations among a larger number of issues' characteristics than only the issue types, such as their priority and final status.

Thus, we identified a set of issue-related features that we hypothesized primarily contribute to the final state of the build process. For this, we first searched through issue collection in the database to scan and select the potentially correlated variables among the recorded features for issues present in the database. Among the 25 present features in the dataset, we selected issue type, priority, status, resolution, and counts of commits and builds, associated with each feature. These features and their existing values in the dataset are demonstrated in Table III, Accordingly, we then built a profile for each 7,982 issues in the four selected types.

Table III lists the selected independent variables and the dependent variable (i.e., CI final state) in the study. The possible values of each feature are presented in the row below. The first column contains a unique ID for the issue. The second column shows the initially selected feature and issue types, with the values of requirements, bugs, improvements, and dependency issues. The next three columns contain the possible values for the issues' features of priority, status, and resolution. The status of an issue is limited to the five categories of blocker, critical, major, minor, and trivial, while the resolution feature provides a more detailed description of the problem's nature and takes 13 values, including cannot reproduce, done, duplicate, fixed, implemented, information provided, invalid, none, not a bug, not a problem, resolved, won't fix, and works for me. Columns six and seven demonstrate the number of commits and builds associated with each issue. Finally, for the dependent variable, the build verdict, we initially searched the database for the values of *failed, passed, errored, canceled, created and started*. However, the builds associated with the selected issue types did not have a final state of created or started.

TABLE III: The Profile (ID, Independent, and Dependent Variables) Constructed for Each Issue. The Assigned Values are Listed Underneath Each Variable

	Independent variables			
ID	Type	Priority	Status	Resolution
	Requirement, bug, im-provement, dependency	Blocker, critical, major, mi-nor, trivial	Closed, in-progress, open, patch available, reopened, resolved	Cannot-reproduce, done, duplicate, fixed, implemented, information provided, invalid, none, not-a-bug, not-a-problem, won't-fix, workaround, works-for-me
	Commits	**Builds**		Verdict
	$n = 1, 2 \leq n \leq 3, 3 < n \leq 10, n > 10$	$n = 1, 2 \leq n \leq 3, 3 < n \leq 10, n > 10$		Passed, failed

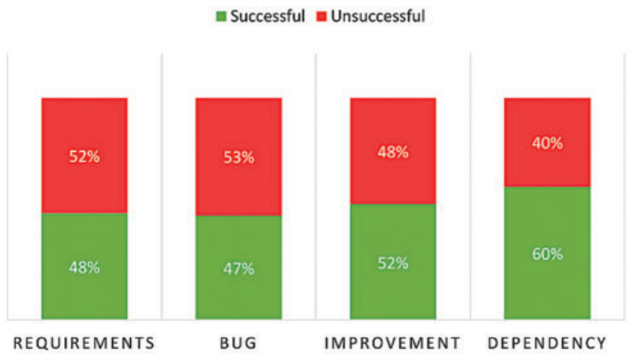

Fig. 4. Histogram showing the percentage of *failed* and *passed* verdicts for each issue type.

As discussed earlier, several issues in the database were associated with more than one build entity and, therefore, contained more than one build verdict. For obvious reasons, in multiple cases, the verdicts were not necessarily consistent. For instance, the issues with passed and errored builds, in the majority of the cases, were eventually followed by a successful build after the problem was resolved. Since the objective of the study is to study difficulties that occur during the software building, we only considered the two verdict values of non-problematic or problematic issues as failed and passed values, respectively. As such, the issues with only failed builds were considered problematic (failed), while all other issues associated with any passed, errored, or canceled builds were considered as non-problematic (passed). As such, each category of change consisted of relatively balanced data to train a binary classifier accordingly. As illustrated in Fig. 4, the requirements contained 48% of data from class failed and 52% from class passed. Similarly, bug, improvement, and dependency types, respectively contained 47% failed and 53% passed records, 52% and 48%, and finally, 60% and 40%.

5.2. *Cramer's Value*

To identify the significant correlations between the issues and their build verdict, we measured a correlation metric for each feature, showing the amount of their interference in the build's success or Failure. For this, we conducted a Chi-square test for the most contributing features of the change in the failure of the CI builds. In addition, in order to measure the extent to which the features contributed, we conducted a Chi test with *Cramer's V.*

Cramer's V value measures the strength of a significant association between the two categories. The higher the value, the stronger the correlation between the pairs. For instance, within the new requirements, commits counts have the highest contingency with the build verdict.

Moreover, *Cramer's V value* measures the strength of a significant association between the two categorical variables. It ranges between 0 and 1. Whenever the value is closer to 0, no association exists, and whenever it is larger than 0.25, a strong relationship exists [25]. For instance, within the new requirements, commit counts have the highest contingency with the build verdict. *Cramer's V* can be defined using the following formula [26], where ϕ_c denotes *Cramer's V*, χ^2 denotes the Pearson chi-square statistic, N denotes the sample size, κ is the lesser number of categories of either variable.

$$\phi c = \sqrt{\frac{\chi^2}{N\,(\kappa - 1)}}$$

5.3. *Results and Discussion*

The study results identify which feature of the posted issues, regardless of the following change type in the source code, commonly contributed to the final verdict of the CI build. As shown in Table IV, the significance value of 0.001, which is a $p-value$ smaller than our chosen significance level ($\alpha = 0.005$) represents a significant correlation among the number of commits, number of builds, priority, and status type of the posted issues, with the CI build failure. However, the *Cramer's Value* of 0.06 prevents us to scientifically draw a conclusion for the extent to which the issues' priority type contributed to the CI failures, since this value demonstrates a specially small and negligible association regardless of a significantly high confidence (significance value < 0.001). This association is stronger between build counts and the CI failure rate with the *Cramer's V values* of 0.29 and 0.27, respectively, showing a large contribution, and the value of 0.12 is considered to be representative of a medium contribution between status type and failure.

The first column of Table IV represents the minimum count percentage of each test. The statistics that passed to pertain to this assumption, and therefore were ignored in the study, are denoted with a * sign in the Table. This conveys that there are not enough samples of relation to draw statistically significant relation among the variables. However, the rest of the cells, with minimum count values of below and equal to 50%, are the features for which the Chi value is valid because the minimum count assumption was met.

TABLE IV: *RQ_2*: The Profile (ID, Independent, and Dependent Variables) Constructed for Each Issue Where Cells Denoted with * are Pairs with Significant-Enough Results

	Feature pairs	Assumption	Pearson chi-square tests				Cramer's V	
		min#	Value	df	Sig.	Critical	Value	Sig.
All	Commits-verdict	0.0%	597.46	3	<0.001	12.838	0.27	<0.001
	Builds-verdict	0.0%	688.81	3	<0.001	12.838	0.29	<0.001
	Priority-verdict	0.0%	28.86	4	<0.001	14.860	0.06	<0.001
	Status-verdict	16.7%	125.51	5	<0.001	16.750	0.12	<0.001
	Resolution-verdict*	41.7%	16.05	11	0.139	5.578	0.04	0.139

For the valid statistics then, the Chi value, (in column three Table V) is compared to a calculated value from the Chi distribution table with the corresponding *df* and *$p-value$*. This value is called the *critical value* and is recorded in the sixth column. Once the Chi value is greater than the critical value, then the null hypothesis is rejected, and an association can be inferred. For the values denoted with *, the number of visited samples was not enough to draw a conclusion. The cells denoted with ^ are pairs with significant enough results.

The test significance value finally determines whether the test was able to determine a statistically significant correlation between the two features or not.

To further investigate the occurrence of the same pattern individually in each category of change, we extended the study to each class of issue related changes. The results are shown in Table V, representing the same pattern in changes relevant to the addition of new functionalities, as well as changes due to the improvement of the product's quality. In the change class relevant to removing dependencies, such a pattern is not statistically shown due to the large p-values. Yet, the alternative hypothesis is not rejected, and instead, it is solely concluded that there is not sufficient evidence to suggest an association between the variable and verdict in this category. The addition of samples in this category of change may improve the p-values so that a statistical conclusion can be drawn. For obvious reasons, the study shows that the number of commits and builds of an issue has a large and positive association with the final build status.

Note that it is not necessarily obvious that more builds will lead to more failures. The relationship between the number of builds and the occurrence of failures can vary depending on various factors. Here are a few scenarios that illustrate different possibilities:

- *Improved stability with more builds:* In some cases, increasing the number of builds can actually improve stability. By running builds more frequently, developers can catch and address issues earlier in the development cycle. This proactive approach helps identify and resolve problems before they accumulate and cause failures. Consequently, a higher number of builds may result in fewer failures overall.
- *Diminishing returns:* In certain situations, there may be diminishing returns associated with increasing the number of builds. Initially, as more builds are executed, the chances of detecting and fixing issues

TABLE V: *RQ_2*: Chi-Square Test Results for the Contribution of Change Attributes to CI Build Failure

Issue types	Feature pairs	Assumption	Pearson chi-square tests				Cramer's V	
		min#	Value	df	Sig.	Critical	Value	Sig.
Requirements	Commits-verdict	0.0%	25.27	3	<0.001	12.838	0.23	<0.001
	Builds-verdict	12.5%	20.40	3	<0.001	12.838	0.20	<0.001
	Priority-verdict^	40.0%	2.02	4	<0.001	14.860	0.06	<0.001
	Status-verdict^	40.0%	17.39	4	<0.002	14.860	0.19	0.002
	Resolution-verdict*	71.4%	10.59	6	0.102	18.548	0.15	0.10
Bugs	Commits-verdict	0.0%	387.28	3	<0.001	12.838	0.28	<0.001
	Builds-verdict	0.0%	464.91	3	<0.001	12.838	0.31	<0.001
	Priority-verdict^	0.0%	22.60	4	<0.001	14.860	0.06	<0.001
	Status-verdict^	50.0%	56.06	5	<0.001	16.750	0.10	<0.001
	Resolution-verdict*	65%	8.10	9	0.523	14.684	0.042	0.523
Improvement	Commits-verdict	0.0%	204.74	3	<0.001	12.838	0.27	<0.001
	Builds-verdict	0.0%	201.109	3	<0.001	12.838	0.26	<0.001
	Priority-verdict^	0.0%	7.153	4	0.128	1.064	0.051	0.12
	Status-verdict^	33.3%	66.071	4	<0.001	14.860	0.15	<.001
	Resolution-verdict*	60%	5.614	9	0.778	4.168	0.04	0.778
Dependency	Commits-verdict*	66.7%	1.681	2	0.432	0.211	0.25	0.432
	Builds-verdict*	66.7%	2.680	2	0.262	0.211	0.32	0.261
	Priority-verdict*	66.7%	0.104	2	0.949	0.211	0.06	0.949
	Status-verdict^	50%	3.175	1	0.075	2.706	0.35	0.075
	Resolution-verdict*	66.7%	0.446	1	0.504	0.016	0.13	0.504

may increase, resulting in a decrease in failures. However, beyond a certain point, additional builds may yield diminishing benefits. The rate of failure reduction may slow down, indicating that other factors, such as code quality or environmental issues, need to be addressed to further improve stability.

- *Unstable or volatile codebase:* If the codebase being built and tested is inherently unstable or frequently changing, it is possible that increasing the number of builds could lead to a higher probability of failures. Rapid code changes, frequent updates, or complex integrations can introduce more opportunities for issues to arise. In such cases, a higher number of builds may uncover more failures, highlighting the need for more rigorous testing and better code management practices.
- *Infrastructure limitations:* The capacity and scalability of the infrastructure supporting the CI system can also influence the relationship between the number of builds and failures. If the infrastructure is insufficient to handle a large number of concurrent builds or lacks necessary resources, it may lead to increased failure rates. Addressing infrastructure constraints, such as upgrading hardware or optimizing resource allocation, can help mitigate the impact of higher build volumes on failure rates.

The relationship between the number of builds and failures is instead context-dependent and can vary based on project characteristics, development practices, and infrastructure. Understanding the specific dynamics at play in a given environment is crucial to effectively managing and optimizing the CI process. Continuous monitoring, analysis, and adaptation are key to ensuring a stable and reliable CI system, regardless of the relationship between build numbers and failures. This being said, understanding the relationship between the number of builds and the final outcome of CI builds can provide valuable insights and help improve the CI process. Here are some specific points that highlight the importance of this relationship:

- *Resource allocation and optimization:* By analyzing the relationship between the number of builds and the final outcome, organizations can gain insights into the resources required for successful CI builds. It helps them determine the optimal number of builds needed to achieve the desired outcome and allocate resources accordingly. For example, if a certain number of builds consistently lead to failure, it may indicate the need for additional testing environments, hardware resources, or infrastructure improvements.
- *Build stability and reliability:* The relationship between the number of builds and the final outcome can shed light on the stability and reliability of the CI system. If there is a high rate of build failures early in the process, it suggests potential issues with the build environment, code quality, or integration problems. Identifying such patterns can help teams diagnose and address the underlying problems, leading to more stable and reliable CI builds.
- *Optimization of build frequency:* CI systems often involve running builds at regular intervals, such as hourly, daily, or on every commit. Understanding the relationship between the number of builds and the final outcome can help optimize the frequency of builds. If the analysis reveals that a certain number of consecutive successful builds significantly reduces the chances of failure, organizations can consider adjusting the build frequency to strike a balance between detecting issues promptly and minimizing unnecessary resource consumption.
- *Performance monitoring and trend analysis:* Monitoring the relationship between the number of builds and the final outcome over time enables organizations to track performance trends. By observing changes in the success rate, they can identify improvements or deteriorations in the CI process. For instance, if the success rate gradually decreases over a period, it may indicate the accumulation of technical debt or increasing complexity in the codebase. Early detection of such trends allows teams to take corrective actions and maintain the efficiency and effectiveness of CI builds.

RQ_2: The study shows that the status of the issue also has a medium positive association with the final build verdicts. This means that not only a large number of commits and builds often led to CI build failure, but the final status of the issue moderately contributed to identifying the verdict of the CI builds. Additionally, a small positive association is observed among the priority flags and the success of the build.

The findings from RQ_2, which indicate a medium positive association between the status of the issue and final build verdicts, as well as a small positive association between priority flags and build success, have several takeaways and implications for the software engineering or CI community, developers, and researchers. These include:

- *Issue status and build verdicts:* The study highlights that the status of an issue, such as open, in progress, or resolved, moderately contributes to identifying the verdict of CI builds. This suggests that the progress and resolution of issues play a significant role in the overall success or failure of CI builds. Developers and software engineering teams should prioritize timely issue resolution and ensure that issues are appropriately tracked and managed throughout the development process.
- *Proactive issue management:* The findings emphasize the importance of actively monitoring and managing the status of issues. Software engineering teams should implement effective issue tracking systems and workflows to ensure that issues are promptly addressed and properly communicated among team members. Regularly updating the status of issues and monitoring their impact on build outcomes can help in identifying potential issues that may lead to build failures.

- *Importance of priority flags:* The small positive association observed between priority flags and build success suggests that assigning appropriate priority levels to issues can contribute to the overall success of CI builds. Prioritizing critical issues and addressing them with higher urgency and attention can reduce the likelihood of build failures. Developers and project managers should consider the priority flags as a useful mechanism for prioritizing and managing issues effectively.
- *Optimization of issue management processes:* The findings highlight the need for optimizing issue management processes within the CI workflow. This includes streamlining issue tracking, ensuring clear communication among team members, and establishing effective protocols for issue resolution. Integrating issue management with the CI pipeline can facilitate better coordination and decision-making, ultimately improving the success rate of CI builds.
- *Further research opportunities:* The findings provide insights into the associations between issue status, priority flags, and build outcomes.

Researchers can explore additional factors that influence build verdicts, such as issue severity, team collaboration, or the impact of issue resolution time on build success. Further investigations can help in developing more comprehensive models and approaches for predicting and preventing build failures based on a broader range of factors.

In summary, the findings from RQ$_2$ emphasize the importance of issue status and priority in relation to CI build outcomes. These findings have implications for software engineering and CI communities, highlighting the need for proactive issue management, optimized processes, and appropriate prioritization of issues. They also suggest opportunities for further research to explore additional factors and enhance the understanding of build failure prediction and prevention.

6. RQ$_3$ Statistical Analysis: Predicting CI Failure-Inducing Changes

The answer to RQ$_2$ statistically demonstrated that the current status of user-requested issues has an acceptable association with the final verdicts of CI builds. Issues with the current status of *in-progress*, *open*, *patch available*, and *reopened* are likely to be identified while the code changes are happening and before the time that a build potentially failed. Yet, the values of *closed* and *resolved* status are most likely not available until the build is complete. As such, this feature (issue's status) is shown useful for the prediction analysis of the CI build failures, only when the implementation of the issue is not yet fully resolved.

For this reason, we further extended the study to measure the extent to which the selected features of the Pass builds were associated with each other (rather than with the build Fails). This allows us to identify the dependent variables whose values increase and decrease relative to each other.

6.1. Feature Interrelations of Failure-Inducing Changes

Contingency coefficient provides a basic picture of the interrelation between two variables, statistically measuring the strength of the relationship between the relative changes of two nominal (categorical) variables.

To identify the presence of a latent relation among pairs of change features, we measured the dependency of each independent variable on the other variables, with the purpose of possibly replacing the non-predictable features with the more predictable ones. The removal of such relations, additionally, removes the conditions' effects from our further analysis.

Table VI displays the contingency coefficient matrix, where each cell demonstrates the value of contingency between each variable pair. The contingency matrix displays the (multivariate) frequency distribution of the variables, and the coefficient is calculated between 0 and 1, where a larger number is representative of a tighter association.

As shown, the *build-commit* and *resolution-status* pairs are highly correlated with a coefficient of 0.7. This size dependency is a relatively high correlation (denoted with *), representing that change in one variable would cause change to another, so the model results fluctuate significantly. While the high correlation among the counts of build and commit is understandable, both values are only identifiable after the changes and failure occurred. However, an issue's status indicates its current place in the project's workflow, but resolutions are the ways in which an issue can be closed, completed, or resolved in many ways [27]. As such, the resolution values are known before their current status is known. The high and positive correlation is understandable since an issue's resolution is usually set when the status is changed.

Fig. 5 shows the total number of builds counts individually for each possible value of the independent variables, *Priority*, *Status*, and *Resolution*. As displayed, on average, the builds failed more frequently for the issues with the "*Trivial*" priority, the status of "*Reopened*," and the resolution of *"Won't Fix"*.

TABLE VI: RQ$_3$: Contingency Matrix

Feature	#Commit	Priority	Status	Resolution	#Build
#Commit	1.00	0.11	0.119	0.08	0.70
Priority	0.01	1.00	0.122	0.08	0.08
Status	0.11	0.12	1.00	0.70	0.14
Resolution	0.08	0.08	0.70*	1.00	0.11
#Build	0.70*	0.08	0.14	0.12	1.00

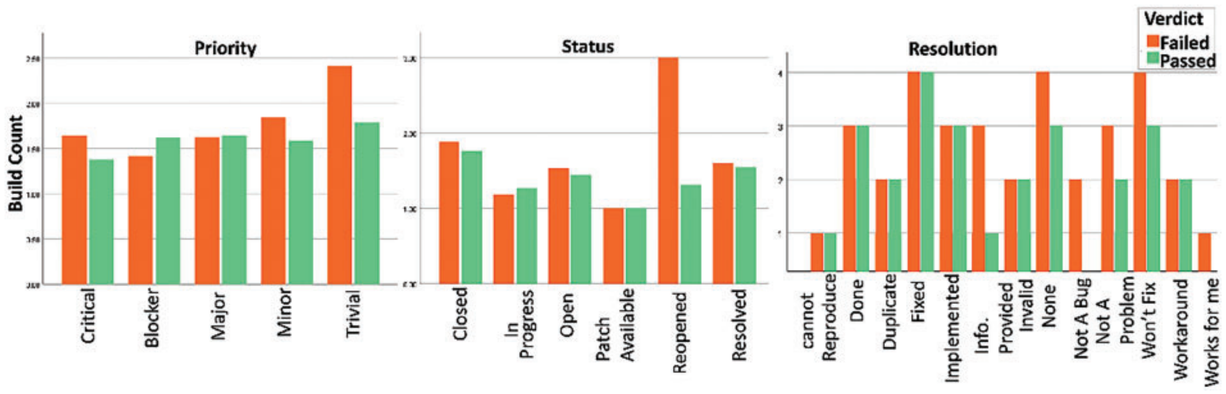

Fig. 5. Build Count for the independent variables, Priority, Status, and Resolution, respectively.

Please note that as the focus of paper is largely on finding and comparing the discriminating ability of code change features, rather than the classification ability of different algorithms, we did not attempt to improve the classification accuracy by adopting other classifiers or to a base model. The study tends to instead identify predictable features of the change to enable software engineers to predict and prevent the build failures.

The study reveals a robust association between the status and resolution of changes, as well as the counts of builds and commits. These significant relationships between independent variables and a specific dependent variable necessitate their mitigation prior to training a classifier on the data. This step is crucial to avoid introducing bias into the predictive model.

6.2. *Model Training*

We investigated whether training a binary classifier allows us to predict the failure-inducing issues ahead of time. For this purpose, we selected a probabilistic classification algorithm, known as discriminant function analysis (DFA). DFA is a statistical procedure that classifies unknown individuals and the probability of their classification into a certain group, assuming that the two different classes generate data based on different Gaussian distributions.

Since the objective is to predict the CI build failures before they occur, we attempted to select the independent variables from the attributes of user-requested issues since they are identifiable before the changes are applied and failure occurs. Yet, to provide a baseline for comparison, we assessed the trained model with attributes that were only predictable for a subset of values.

6.3. *Results and Discussion*

The accuracy of this classification is reported in Table VII, using 10-fold cross-validation. For the comparison purposes, we report the results once using the identifiable attributes and once again with the attributes that are identifiable only for some values before the actual failure occurs but not the entire range of possible values. As shown, the top part of the table represents the selected dependent variable(s) and whether or not the variable(s) is(are) predictable ahead of time. The bottom part displays the count of the samples for each class of failed and passed. Considering the *Failed* samples as negative and *passed* instances as positive instances, then the first square denoted with * from left represents the number of true positives as 3,166 (77.8%) records. These are the CI builds which failed in reality and were correctly detected by the classifier as failure-inducing builds by only using the priority tag of the posted issues as the discriminating variable. The second the first square denoted with *, displays the number of true positives. As shown, 26.6% of the passed builds were correctly marked by the model. However, 2,870 successful builds were mistakenly marked by the classifier as potential to fail by the model while they pass in real life. While this represents 73.4% false positives, it leads to leaving out, marking only 22.2% of the actual failure-inducing builds.

Please note that in this problem, we are more interested in identifying the potentially failing builds, which will consequently delay the development process. While false positives require developers' effort to be filtered out, improve the possibility to identify the changes which are the most likely to pass. This said the model yields a higher chance of detecting CI build failures (recall of 77.78%) and a precision of about 52.45%. Including resolution (partially predictable), the recall and precision are increased to 78.15% and 52.51%, respectively.

An effort to improve precision is simply obtainable through setting a higher probability threshold to tag failing builds, decreasing the number of false positives. However, this will also negatively impact the recall. The balance between the two metrics, precision and recall, largely relies on the context of the project, as well as the resources available in a software project. For instance, safety-critical software may expect a more pessimistic approach (a higher threshold), but instead require a more expensive process for filtering the instances which were mistakenly marked as potentially-failing builds.

RQ$_3$: By utilizing a sufficient sample size, we achieved statistical prediction of failure-inducing changes based on the priority of user-posted issues, yielding a recall of approximately 78% and precision of approximately 53%. Additionally, comparable performance values were obtained when considering both the priority and resolution of the issues.

Note that while the importance of recall or precision depends on the specific context and requirements of the software engineering or CI application, if the primary goal is to identify as many failure-inducing changes as possible,

TABLE VII: RQ_3: DFA's Prediction Results with Possible-to-Know, Not-Possible-to-Know, Both, and All Dependent Variables

	Independent variables							
	Predictable				Partially predictable			
	Priority		Priority & resolution		Priority & status		All	
Ground truth	Predicted independent value							
	Failed	Passed	Failed	Passed	Failed	Passed	Failed	Passed
Fail	3,166* (77.8%)	904 (22.2%)	3,180∗ (78.2%)	889 (21.8%)	2,643* (64.9%)	1,427 (35.1%)	2,076* (51.0%)	1,993 (49.0%)
Pass	2,870 (73.4%)	1,042* (26.6%)	2,875 (73.5%)	1,037* (26.5%)	2,107 (53.9%)	1,805* (46.1%)	1,015 (25.9%)	2,897* (74.1%)

the cost of some false positives, then maximizing recall could be more important in this case than maximizing precision. However, maintaining an acceptable level of precision ensures that developers can effectively utilize the approach without becoming overwhelmed by excessive false negatives.

The findings of RQ_3, which demonstrate the ability to statistically predict failure-inducing changes based on the priority and resolution of user-posted issues, have several takeaways and implications for the software engineering or CI community, developers, and researchers. These include:

- *Enhanced issue prioritization:* The findings suggest that considering the priority of user-posted issues can be valuable in predicting failure inducing changes. Software engineering teams and developers can prioritize their efforts based on issue priority, focusing on high-priority issues that are more likely to result in build failures. This can lead to more efficient resource allocation and better issue management practices.
- *Proactive issue resolution:* By leveraging the predictive model, developers can take a proactive approach to resolving issues with a higher likelihood of causing build failures. This allows them to address critical issues in a timely manner, potentially reducing the overall number of build failures and improving the stability of the CI process.
- *Refining classification algorithms:* The findings highlight the potential of incorporating issue priority and resolution as important features in classification algorithms used for failure prediction. Researchers can further explore and refine these algorithms to improve the accuracy and performance of predictive models. This involves investigating additional factors or features that may contribute to failure prediction, such as issue severity or the impact of other contextual information.
- *Continuous improvement of CI practices:* The findings provide insights into the relationship between user-posted issue characteristics and failure-inducing changes. This encourages software engineering teams to continuously evaluate and optimize their CI practices. By monitoring and analyzing the performance of the predictive model, teams can identify areas for improvement, refine their processes, and make informed decisions to minimize build failures.
- *Generalizability and transferability:* The findings highlight the potential generalizability and transferability of the predictive model to other software engineering or CI contexts. Developers and researchers can explore the applicability of the model in different projects, development environments, or domains to enhance their understanding of failure prediction and develop best practices tailored to specific contexts.

In summary, the findings of RQ_3 provide insights into the prediction of failure-inducing changes based on issue priority and resolution. These findings have implications for software engineering or CI communities, emphasizing the importance of issue prioritization, proactive issue resolution, refining classification algorithms, continuous improvement of CI practices, and exploring the generalizability and transferability of the predictive model.

7. Threats to Validity

A threat to construct validity in our study is the potential bias caused by 98 software projects we used. We minimized this threat by selecting a dataset which contained projects relevant to a wide range of applications and functionalities. Yet, further cross-dataset validations are necessary to draw certain conclusions.

A threat to internal validity contains categorizing the issues based on the issue types. The issues could be assigned to the wrong category, or mistakenly a wrong issue tag could be selected by the developers [28]. We only selected issue types with those tags which clearly mentioned the context of the change, to minimize this threat while this may lead to removing some issues that could fall under our selected categories. Another bias was raised because of the uneven sample size among different types of issues in the database. This could influence the statistics we provided for the group containing all the issue types. We tried to apply a set of pre-processing techniques to minimize other statistical threats as much as possible, as discussed in the paper. In addition, while some artifact counts were uneven among the four-issue types, each type was individually balanced in terms of successful and failed build counts. Another bias may occurred due to the dataset incomplete Travis builds and jobs tables, which excluded information for multiple of the projects. While our statistical studies consider the amount of observed samples to measure evidence, yet the reported results are only on a subset of the build jobs.

The *external validity* includes the selection of open-source projects which might not be representative of or generalizable to closed-access commercial projects. In our future work, we will conduct experiments with additional projects.

8. Related Works

This section shares a brief summary of the research related to finding the root causes of, predicting and preventing the software build failures. Further, we share the related works which specifically studied CI builds. We provide literature on using profiles for data mining tasks, in particular in the software domain. Finally we summarized the works in the impact of human factors in CI builds, as well as the studies which particularly analyzed Travis CI.

8.1. *Finding the Root Causes of Build Failures*

Several prior studies have investigated the underlying causes of build breaks. Research on open source projects indicated project build failures happen mainly due to unit testing failures [29]; a group of researchers found that builds generally fail because of failed test cases [12], and similarly, a research study specified testing failures, compilation errors, and poor code quality as the most recurrent causes of build failures in Microsoft projects [4]. Another study identified poor code quality, identified with static analysis techniques, as the primary reason of failures [30].

This research instead associates build failure with code change type and identifies common features contributing to the failed builds.

8.2. *Predicting/Preventing Build Failures*

Saidani *et al*. [10] proposed a search-based approach, multi-objective genetic programming (MOGP) technique, to predict CI build failures. The model aims to identify the failures through finding the best combination of CI build features.

Although the approach attempts to predict CI build failures, the primary focus is on generating a set of rules for the establishment of good practices by software developers.

Xia and Li [13] developed multiple classifiers to predict build failures for TravisTorrent projects, identifying the majority of the metrics to be specific to TravisTorrent database. The results indicated that the predictive models performed well for cross-validation scenarios but failed to perform well in on-line scenarios. Hassan and Wang [9] built a prediction model which performed with an average F-measure of 78% in cross-project prediction scenarios. Similarly, the majority of the metrics selected to build the models were features of the TravisTorrent environment. The rest of the metrics were particularly defined at the method-level changes, such as method body change count and method signature change count. Rausch *et al.* [31] conducted an empirical study of CI builds only for Java-based applications. Their analysis had two folded directions. The first one intends to identify the types of errors in CI builds, while the second part aims to develop a general process and specific CI metrics for CI build failures.

Ni and Li [32] proposed a cost-effective approach to predict build outcomes using cascaded classifiers based on the commit information from Version Control systems. They extracted several features, such as last build result, time elapsed, etc., to train the cascaded classifier. Chen *et al.* [33] proposed a history-aware approach to predict CI build outcomes that can help to get fast integration feedback and also reduce integration costs by analyzing build logs and changed files closely related to historical builds. Saidani *et al*. [34] proposed a CI build failure prediction as a time series problem using LSTMRMM based model by considering various features such as number of commits, number of files added, deleted, and so on.

Compared to the above-mentioned works, this work searches for a set of generic features of the code changes and their initiating user requests, which more frequently lead to build failures. This research builds a predictive model whose features are not specific to a specific framework and, therefore, are generalizable to any environment.

8.3. *Study/Analysis/Impact of CI Builds*

Zhao *et al*. [35] performed a qualitative study to identify how the rise of CI practices changed software development practices in general. Several research studied and analyzed the impacts of CI practices on different software development approaches from multiple perspectives, such as code review [36], [37] and delivery time of pull requests [38]. The aforementioned research significantly contributed to better analyzing the software development environments which practice CI and in providing insight about the impacts of CI applications. Compared to these works, this research mainly focuses on predicting the final verdict of CI builds according to the characteristics of the precedent changes, resulting in taking preemptive actions to prevent the halt of the development process, if necessary. In addition to the above-mentioned works, Zolfagharinia *et al.* [39] studied the CI build inflation by analyzing the relationship between Runtime Environments (RE) and Operating Systems (OS) and build failures on 30 million build records of CI environments. Their result indicates, the builds on Perl packages act differently on different Res and OSes. Paixao *et al*. [40] conducted a study to investigate the relationship between Non-functional requirements (NFR) and Travis-CI build statuses. However, they focused on NFR related build failures mainly where they categorized their types and duration to fix, which is a different area of research.

8.4. *Creating Developer Profile*

Constructing and using profiles for software artifacts is an active research area. One common application is to extract developer profiles from publicly available sources. For instance, multiple works created a profile for developers active on GitHub [41], [42]. Mining developer profile data is also used to match job advertisements [43], find experts [44], [45], measure developers' contributions [46], personalizing recommendations [47], identify the gender and nationwise diversity of the team [48], recommending relevant project [49], exploring the patterns of social behavior [50] and so on. Bao *et al.* [51] conducted a study to find long-time contributors for open-source applications based on their previous activities on Github. Tools like CVExplorer [52] and CPDScorer [53] are also developed to mine technical skills by the developers. Xiong *et al.* [54] explored cross-site developer behavior on StackOverflow and GitHub T-graph analysis, LDA-based topics clustering, and cross-site tagging. Souza and Silva [55] analyzed Travis builds and their comments to see if negative sentiments have some impact on Travis builds.

To compare our work with those mentioned, this research focused on the issues' profile in order to find the potential relations between the issue types, their consequent changes, and their features which were more likely to contribute to the CI build failures.

8.5. *Human Factors in CI Builds*

Studies on the impacts of human factors on the successful build process are less common, but there has been recent progress. For instance, Wolf *et al*. [56] studied the association of build fails with the social factors of developers. Using social network analysis, they trained a model to predict whether an integration will fail based on its developers' communications. Other researchers studied features related to relationships between developers of build, such as communication, trust, and conflict. For instance, Phillips *et al.* found that social challenges of build team engineers impact their effectiveness and therefore, proposed to address social impediments in build teams in order to improve the build process [57].

Although mining data from software repositories, including Travis CI and Pull requests, is not new research, creating issue profiles from the build reports has not yet been explored to the best of our knowledge. Since SmartSHARK [17] database integrates data from multiple sources, such as GitHub, Travis CI, Issue trackers, we used the data to study whether we can build issue profiles to later foresee the verdict of a CI build according to the nature of the change. In contrast, this work tends to emphasize the potential relations between the issue types, consequent changes, and their features, which contribute more to the build failures so that potential failures can be predicted and prevented in advance.

9. Conclusion and Future Work

Continuous Integration (CI) is frequently practiced to provide the developers with the functionality to merge their code changes into a central repository to be automatically built and incorporated into the development process. During the building process, as an important part of the software development process, identifying the recurrent causes of the build failure allows to prevent their occurrence.

To this end, we analyzed the impact of the major maintenance change types on the build verdicts of 98 Apache projects. The study statistically showed that addressing the issues which required a bug fix led to the largest percentage of the CI build failures.

Further, we observed that the issues' *priority* tags have a positive association with the success of the build, while the issues' *status* and *resolution* were additionally associated. This led to building multiple mathematical models that detected the builds that were highly likely to fail, only according to the priority of their associated issues, as well as their priority and resolution. These independent variables are more feasible to be known before the CI build fails in the real world.

In the future, we plan to collect additional software projects to include a larger set of logs related to building job records. Extending our dataset, we intend to study dependencies among a larger set of attributes and change types. Additionally, we want to parse Travis logs to find the root causes of CI build failures.

Acknowledgment

I want to add acknowledgement The work in this paper was funded by the Office of Naval Research (ONR) (Grant#:G2A62826).

Data Availability

The artifacts of this research are publicly accessible on GitHub[1].

Conflict of Interest

The authors declare that they do not have any conflict of interest.

References

[1] Chapin N, Hale JE, Khan KM, Ramil JF, Tan WG. Types of software evolution and software maintenance. *J Softw Maint Evol: Res Pract*. 2001;13(1):3–30.
[2] What is a software maintenance process? 4 types of software maintenance. 2024. Accessed: 2024-01-01. Available from: https://cpl.thalesgroup.com/.
[3] What is continuous integration (CI)? 2024. Accessed: 2024-01-01. Available from: https://circleci.com/continuousintegration/.
[4] Miller A. A hundred days of continuous integration. *Agile 2008 Conference*, pp. 289–93, IEEE, 2008.
[5] Hilton M, Nelson N, Tunnell T, Marinov D, Dig D. Trade-offs in continuous integration: assurance, security, and flexibility. *Proceedings of the 2017 11th Joint Meeting on Foundations Software Engineering*, pp. 197–207, 2017.
[6] Vassallo C, Proksch S, Zemp T, Gall HC. Every build you break: developer-oriented assistance for build failure resolution. *Empir Softw Eng*. 2020;25(3):2218–57.
[7] Sulír M, Bačíková M, Madeja M, Chodarev S, Juhár J. Large-scale dataset of local java software build results. *Data*. 2020;5(3):86.
[8] Lou Y, Chen J, Zhang L, Hao D, Zhang L. History-driven build failure fixing: how far are we? *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 43–54, 2019.
[9] Hassan F, Wang X. Change-aware build prediction model for stall avoidance in continuous integration. *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pp. 157–62, IEEE, 2017.
[10] Saidani I, Ouni A, Chouchen M, Mkaouer MW. Predicting continuous integration build failures using evolutionary search. *Inf Softw Tech*. 2020;128:106392.
[11] Kerzazi N, Khomh F, Adams B. Why do automated builds break? an empirical study. *2014 IEEE International Conference on Software Maintenance and Evolution*, pp. 41–50, IEEE, 2014.
[12] Beller M, Gousios G, Zaidman A. Oops, my tests broke the build: An explorative analysis of travis CI with GitHub. *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pp. 356–67, IEEE, 2017.
[13] Xia J, Li Y. Could we predict the result of a continuous integration build? an empirical study. *2017 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, pp. 311–5, IEEE, 2017.

[1] https://github.com/SamihaShimmi/CIBuildFailure.

[14] Log parser plugin. 2024. Accessed: 2024-01-01. Available from: https://wiki.jenkins.io/display/JENKINS/Log+Parser+Plugin.

[15] Macho C, McIntosh S, Pinzger M. Automatically repairing dependencyrelated build breakage. *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 106–17. IEEE; 2018.

[16] Beller M, Gousios G, Zaidman A. Travistorrent: synthesizing travis CI and GitHub for full-stack research on continuous integration. *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pp. 447–50. IEEE; 2017.

[17] Trautsch A, Trautsch F, Herbold S. MSR mining challenge: the SmartSHARK repository data. *Proceedings of the International Conference on Mining Software Repositories (MSR 2022)*, 2021.

[18] Dashboard System. 2024. Accessed: 2024-01-01. Available from: https://issues.apache.org/jira/secure/Dashboard.jspa.

[19] Documentation of the SmartSHARK database. 2024. Accessed: 2024-01-01. Available from: https://smartshark2.informatik.uni-goettingen.de/.

[20] Karl Pearson FRS. X. On the criterion that a given system of deviations from the probable in the case of a correlated system of variables is such that it can be reasonably supposed to have arisen from random sampling. *London, Edinburgh, Dublin Philos Mag J Sci*. 1900;50(302):157–75. doi: 10.1080/14786440009463897.

[21] Cramer H. Mathematical methods of statistics. In *Cráer Mathematical Methods of Statistics 1946*. Princeton: Princeton University Press, 1946.

[22] McHugh ML. The chi-square test of independence. *Biochemia Medica*. 2013;23(2):143–9.

[23] Onchiri S. Conceptual model on application of chi-square test in education and social sciences. *Educ Res Rev*. 2013;8(15):1231–41.

[24] Sharpe D. Chi-square test is statistically significant: now what? *Pract Assessment, Res, Eval*. 2015;20(1):8.

[25] Akoglu H. User's guide to correlation coefficients. *Turkish J Emer Medi*. 2018;18(3):91–3.

[26] Tutorial SPSS. 2014. Accessed: 2024-01-01. Available from: https://www.spss-tutorials.com/cramers-v-what-and-why/.

[27] Support ATLASSIAN. 2024. Accessed: 2024-01-01. Available from: https://support.atlassian.com/.

[28] Herbold S, Trautsch A, Trautsch F. Issues with SZZ: an empirical assessment of the state of practice of defect prediction data collection. CoRR. 2019; abs/1911.08938. Available from: http://arxiv.org/abs/1911.08938.

[29] Vassallo C, Schermann G, Zampetti F, Romano D, Leitner P, Zaidman A, *et al.* A tale of CI build failures: an open source and a financial organization perspective. *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 183–93. IEEE; 2017.

[30] Zampetti F, Scalabrino S, Oliveto R, Canfora G, Di Penta M. How open source projects use static code analysis tools in continuous integration pipelines. *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pp. 334–44. IEEE; 2017.

[31] Rausch T, Hummer W, Leitner P, Schulte S. An empirical analysis of build failures in the continuous integration workflows of java-based opensource software. *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pp. 345–55. IEEE; 2017.

[32] Ni A, Li M. Cost-effective build outcome prediction using cascaded classifiers. *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pp. 455–8, 2017.

[33] Chen B, Chen L, Zhang C, Peng X. BuildFast: history-aware build outcome prediction for fast feedback and reduced cost in continuous integration. *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering. ASE '20*, pp. 42–53, New York, NY, USA: Association for Computing Machinery, 2021. doi: 10.1145/3324884.3416616.

[34] Saidani I, Ouni A, Mkaouer MW. Improving the prediction of continuous integration build failures using deep learning. *Autom Softw Eng*. 2022;29(1):21.

[35] Zhao Y, Serebrenik A, Zhou Y, Filkov V, Vasilescu B. The impact of continuous integration on other software development practices: a large-scale empirical study. *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 60–71. IEEE; 2017.

[36] Rahman MM, Roy CK. Impact of continuous integration on code reviews. *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pp. 499–502. IEEE; 2017.

[37] Cassee N, Vasilescu B, Serebrenik A. The silent helper: the impact of continuous integration on code reviews. *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 423–34. IEEE; 2020.

[38] Bernardo JH, da Costa DA, Kulesza U. Studying the impact of adopting continuous integration on the delivery time of pull requests. *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, pp. 131–41. IEEE; 2018.

[39] Zolfagharinia M, Adams B, Guéhéneuc YG. A study of build inflation in 30 million CPAN builds on 13 Perl versions and 10 operating systems. *Empir Softw Eng*. 2019;24(6):3933–71.

[40] Paixão KV, Felício CZ, Delfim FM, Maia MdA. On the interplay between non-functional requirements and builds on continuous integration. *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pp. 479–82. IEEE; 2017.

[41] Montandon JE, Silva LL, Valente MT. Identifying experts in software libraries and frameworks among github users. *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pp. 276–87. IEEE; 2019.

[42] Santos A, Souza M, Oliveira J, Figueiredo E. Mining software repositories to identify library experts. *Proceedings of the VII Brazilian Symposium on Software Components, Architectures, and Reuse*, pp. 83–91, 2018.

[43] Hauff C, Gousios G. Matching GitHub developer profiles to job advertisements. *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pp. 362–6. IEEE; 2015.

[44] Mockus A, Herbsleb JD. Expertise browser: a quantitative approach to identifying expertise. *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*, pp. 503–12. IEEE; 2002.

[45] Schuler D, Zimmermann T. Mining usage expertise from version archives. *Proceedings of the 2008 International Working Conference on Mining Software Repositories*, pp. 121–4, 2008.

[46] Gousios G, Kalliamvakou E, Spinellis D. Measuring developer contribution from software repository data. *Proceedings of the 2008 International Working Conference on Mining Software Repositories*, pp. 129–32, 2008.

[47] Ying AT, Robillard MP. Developer profiles for recommendation systems. In *Recommendation Systems in Software Engineering*. Springer, 2014, pp. 199–222.

[48] Ortu M, Destefanis G, Counsell S, Swift S, Tonelli R, Marchesi M. How diverse is your team? Investigating gender and nationality diversity in GitHub teams. *J Softw Eng Res Dev*. 2017;5(1):1–18.

[49] Guendouz M, Amine A, Hamou RM. Recommending relevant open source projects on GitHub using a collaborative-filtering technique. *Int J Open Source Softw Processes (IJOSSP)*. 2015;6(1):1–16.

[50] Yu Y, Yin G, Wang H, Wang T. Exploring the patterns of social behavior in GitHub. *Proceedings of the 1st International Workshop on Crowdbased Software Development Methods and Technologies*, pp. 31–6, 2014.

[51] Bao L, Xia X, Lo D, Murphy GC. A large scale study of long-time con- tributor prediction for github projects. *IEEE Trans Soft Eng*. 2019;47(6):1277–98.

[52] Greene GJ, Fischer B. Cvexplorer: identifying candidate developers by mining and exploring their open source contributions. *Proc 31st IEEE/ACM International Conference on Automated Software Engineering*, pp. 804–9, 2016.

[53] Huang W, Mo W, Shen B, Yang Y, Li N. *CPDScorer: Modeling and Evaluating Developer Programming Ability Across Software Communities*. In *SEKE*, 2016, pp. 87–92.

[54] Xiong Y, Meng Z, Shen B, Yin W. *Mining Developer Behavior Across GitHub and StackOverflow*. In *SEKE*, 2017, pp. 578–83.

[55] Souza R, Silva B. Sentiment analysis of travis ci builds. *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pp. 459–62. IEEE; 2017.

[56] Wolf T, Schroter A, Damian D, Nguyen T. Predicting build failures using social network analysis on developer communication. *2009 IEEE 31st International Conference on Software Engineering*, pp. 1–11. IEEE; 2009.

[57] Phillips S, Zimmermann T, Bird C. Understanding and improving software build teams. *Proceedings of the 36th International Conference on Software Engineering*, pp. 735–44, 2014.