# Why do Builds Fail? – A Conceptual Replication Study

Amine Barrak[a], Ellis E. Eghan[a], Bram Adams[b], Foutse Khomh[a]

[a]*Polytechnique Montréal*
*Montréal, Canada*
[b]*School of Computing Queen's University*
*Kingston, Canada*

## Abstract

Previous studies have investigated a wide range of factors potentially explaining software build breakages, focusing primarily on build-triggering code changes or previous CI outcomes. However, code quality factors such as the presence of code/test smells have not been yet evaluated in the context of CI, even though such factors have been linked to problems of comprehension and technical debt, and hence might introduce bugs and build breakages. This paper performs a conceptual replication study on 27,675 Travis CI builds of 15 GitHub projects, considering the features reported by Rausch et al. and Zolfagharinia et al., as well as those related to code/test smells. Using a multivariate model constructed from nine dimensions of features, results indicate a precision (recall) ranging between 58.3% and 79.0% (52.4% and 69.6%) in balanced project datasets, and between 2.5% and 37.5% (2.5% and 12.4%) in imbalanced project datasets. Models trained on our balanced project datasets were later used to perform cross-project prediction on the imbalanced projects, achieving an average improvement of 9.3% (16.2%) in precision (recall). Statistically, the results confirm that features from the build history, author, code complexity, and code/test smell dimensions are the most important predictors of build failures.

*Keywords:* Continuous integration, Build failure, Test smells, Code smells, Quantitative analysis, Cross-project prediction

## 1. Introduction

The last 20 years have seen continuous integration (CI) turn into a best practice for software organizations to manage the regular integration of developers' code into shared repositories such as GitHub or Bitbucket [1]. CI tools automatically run all (scheduled) builds and tests to ensure that submitted code changes are free of compilation and test failures. However, this process can be time-consuming and costly. It is estimated that large (legacy) projects may take hours or even days to build [2, 3, 4].

This affects the overall efficiency of developers since major decisions (on code changes) depend on the outcome of the ongoing build. The situation becomes worse when build inflation is considered. Build inflation occurs when a single commit results in several automated builds due to the different specified configurations (runtime environments, operating systems and hardware architectures) [5]. Therefore, the ability to automatically predict build outcomes could avoid waiting for the (many) build results to come in, saving the organization substantial time and money. For example, organizations can choose not to run predicted successful builds, freeing up time and resources to focus on the predicted failing builds.

It should be noted that a build, in the context of this work, is a set of configured jobs that differ from each other by the execution environment. The status of a build is determined by the state of its jobs — a build is successful only if all jobs are successful. CI jobs are often comprised of three phases: (i) the traditional build and compile phase, (ii) a phase in which automated static analysis tools (e.g., FindBugs, PMD and JSHint) are executed, (iii) a testing phase for running unit, integration, and system tests. If one of these phases fails, the entire build fails [3].

Many explanatory and prediction approaches have been proposed by the research community to identify the causes of software build failures and ways to reduce the cost and effort of fixing such failures. These studies characterise build failures, in general, using properties of social networks [6], socio-technical congruence [7], without taking into account the specific context of CI builds. Recently, other studies have focused on the use of source code features in predicting build outcomes. For example, Finaly *et al.* [8], using Hoeffding Tree classification method, identified source code features as useful features to predict build success and/or failure in software products. Also, Hassan [9] proposes a model based on the complexity of code changes. However, the vast majority of these works use univariate models that only assess the effect of individual features, and also pay little attention to code quality.

Therefore, in this paper, we are interested in investi-

*Email addresses:* amine.barrak@polymtl.ca (Amine Barrak), ellis.eghan@polymtl.ca (Ellis E. Eghan), bram.adams@queensu.ca (Bram Adams), foutse.khomh@polymtl.ca (Foutse Khomh)

gating whether multivariate models, rather than univariate models of project features, can more accurately explain and predict build failures. To achieve this, we perform a *conceptual replication* of the studies by Rausch *et al.* [10] and Zolfagharinia *et al.* [5]. Rausch *et al.* [10] found code complexity, author experience and build history to be statistically influential factors in explaining build failure, while Zolfagharinia *et al.* [5] explored the impact of build environments and platforms on build failures.

Unlike exact replications that follow the procedures (and data) of the original study as closely as possible, conceptual replications use different experimental procedures to study the same (or similar) research questions and allow the research community to gain additional confidence in the results of the original study [11].

Therefore, in this paper, we build multivariate models based on a combination of the features used by Rausch *et al.* [10] and Zolfagharinia *et al.* [5]. In addition, we include a number of quality features, related to code and test smells, in our multivariate models. Previous studies have shown that the presence of code and test smells can negatively affect the quality of test suites and production code [12], potentially exposing the codebase to incremental changes and fault-proneness [13, 14, 15, 16].

To the best of our knowledge, this is the first study that is combining code/test quality and development process features; we consider 16 development process features (e.g., the complexity of code changes, developer experience, date and time of commits, build history) [10], one build environment feature (runtime environments) [5], 12 code smells and 5 test smells.

In addition, our study separately analyzes the effect of the aforementioned features on 27,675 Travis CI builds of 15 Java GitHub projects. By building explanatory (RQ1/2) and predictive (RQ3) random forest models, we address the following research questions:

- **RQ1:** ***How well do models based on the considered factors explain build failures?***
  Our exploratory models, combining nine different dimensions of features, achieve a median precision of 79.0%, a median recall of 69.6% and a median AUC of 75.3% in balanced projects. However, models in imbalanced projects achieve a median precision of 37.5%, a median recall of 12.4% and a median AUC of 55.3%.

- **RQ2:** ***Which features are the best indicators of build failures?***
  We found that in both balanced and imbalanced datasets, individual features of the code complexity (*Number of commits, Number of Author, Number of Lines Removed, Number of Lines Added*), author (*Author Experience*), build history (*Build Climate, Previous Build, Days since Last Fail*), code smell (*Long Parameter List, Many Field Attributes But Not Complex, AntiSingleton, Large Class, Speculative Generality*), test smell (*Sensitive Equality*) dimensions are the most important features that could be used to explain build breakages.

- **RQ3:** ***Can we predict the outcome of future builds?***
  Our on-line prediction model was able to predict future build failures with a precision (recall) ranging between 25% and 80% (5% to 70%) based on the proportion of failing builds in the dataset. Our results confirm earlier on-line prediction results by Xia and Li [17].

We believe that both practitioners and researchers would benefit from the results of our replication study. Our results confirm most of the key findings of the original studies. More specifically, we confirm that the complexity of code changes, build type, results of previous build, and author experience have high correlation with build failures. On the other hand, the impact of file types and build environment on the outcome of a build were not (fully) confirmed in our study. Additional analysis conducted in this work shows that the performance of build prediction models on imbalanced datasets can be improved using cross-project models trained on balanced datasets. We recorded an average improvement of 9.3% (16.2%) in precision (recall) for imbalanced project datasets. These results provide opportunities for future research; our replication package is made publicly available to aid in such future research endeavours [18].

The remainder of this paper is organized as follows. First, we discuss the related literature on build failure in Section 2. Next, we describe the experimental setup of our study in Section 3 and report our findings in Section 4. In Section 5, we provide in-depth discussion of our findings and compare our results with those of the replicated papers. Section 6 discusses threats to the validity of our work. Section 7 concludes our work and outlines avenues for future work.

## 2. Related Work

In this section, we go through related studies and highlight our contributions. We group our related work into two different areas: (i) predicting builds and (ii) code quality and failure proneness.

### *2.1. Predicting failing builds*

Existing studies have identified compilation errors, failing tests, dependency errors and build configuration/environments as the most common causes of build failures [5, 10, 19, 20]. In addition, other studies have used explanatory models to identify correlations between build failures and historical project information such as code changes and socio-technical factors [6, 10, 21]. For example, Seo *et al.* [21] observed that the dependencies among developers and software components increase when the sizes of the team and the project are large. Such dependencies, particularly in code, are susceptible to broken builds. Wolf *et al.* [6] also studied the relation between the communication structure of development teams and the result of their code integration build processes. They found that developer communication has an important role in the quality of software integration. Their exploratory model yielded recall values between 55% and 75%, and precision values between 50% to 76%.

Several other researchers have proposed approaches that use code features, social network analysis and socio-technical factors to predict the outcome (failure) of future builds. The first study in the area of predicting failing builds, by Hassan and Zhang [4], introduced a decision tree model that was able to predict 69% of failures based on historical information. Xia and Li [17] compare the performance of nine classifiers in predicting build failures across 126 OSS projects. Using cross-validation models, they observed that the Random Forest classifier attains the best performance (across 48 projects) with median AUC and F1 scores of 76% and 56%, respectively. They also performed on-line prediction on chronologically ordered builds and found the Random Forest among the best performing classifiers with median AUC and F1 scores of 54% and 10%.

Other researchers have used build prediction as a way to reduce the total number of builds in a CI pipeline. Jin and Servant [22] proposed a tool called *SmartBuildSkip*, which reduces the number of builds by skipping predicted build successes until a failure is predicted; predicted failed builds are then executed until they succeed. Their approach uses features such as the complexity of changes, historical builds (ratio of passing build, failure distance), date and time, and project characteristics (e.g., age, test density, project, and team size). They proposed two variants of their approach, one to learn from previous builds within a project and another to learn from cross-project data (at the start of a new software project). Similarly, Ni and Li [9] use cascaded classifiers to set high-confidence threshold for successful builds and only keep builds that are likely to fail, thus reducing the number of executed builds and providing more refined analysis. Their model was able to identify 85.2% of the defective builds. They used a combination of historical statistics (project and committer), last push, and current build dimensions as features to the prediction classifiers. Their approach outperforms classifiers like C4.5 decision tree and NaiveBayes with a precision (recall) of 73.7% (80%).

The closest works to ours, for which this replication study is conducted, are by Rausch *et al.* [10] and Zolfagharinia *et al.* [5]. Rausch *et al.* [10] performed qualitative and quantitative studies on CI build failures and identified several features that have strong correlations with build failures. These features were categorized into two groups: (1) process features collected from the characteristics of the commits that trigger automated builds, *i.e.,* complexity of code changes, file type, date and time, developer-specific criteria; and (2) CI features, corresponding to build type, pull request scenario and characteristics of build history. Using univariate models, the authors identified code changes (of high complexity), build history climate, and developer experience to have significant effect on a build's outcome. In contrast to their strategy, we use both individual and multivariate analysis to better explain the relative impact of a feature on build failure. Multivariate models take all the analyzed features into account when explaining or predicting build failures.

Zolfagharinia *et al.* [5], on the other hand, explored other kinds of features such as the operating systems, runtime environments and hardware architectures on which the build was executed in the continuous integration systems. Based on an analysis of the impact of 7 OSes and 22 environments on build failures, the authors observe that the most common build failure categories are caused by the targeted OSes and build environments. In addition, their results show that the build results across all OS are not necessarily uniform. They recommend careful filtering and selection of the results of CI in order to identify reliable build failure data. They found, across the studied versions, that 77% of the builds were successful on all build environments, and 12% of the previously failing builds are fixed in later versions of the build environment. However, the remaining builds (11%) always fail irrespective of the used build environment version. In our replication, we chose a different set of Java projects (instead of the Perl projects used by Zolfagharinia *et al.* [5]) due to the requirements of our code/test smell analysis tools. We applied random forest exploratory models to analyze the impact of four Java environments (identified from the studied projects) on the builds failure. While, we did not find a statistical improvement of environments as a dimension, we found the effect sizes of the individual environments.

### *2.2. Code quality and failure proneness*

Code and test smells are poor solutions to design and implementation problems, and their presence can lead to fault proneness. Khomh *et al.* [14] studied the relationship between code smells, code changes and fault-proneness in 4 object-oriented systems. They found that classes affected by code smells are frequently changed and have a higher tendency to be fault-prone compared to safe classes. Sabane *et al.* [23] also studied the impact of anti-patterns on the cost of testing in a given system. They showed that the presence of anti-patterns, on average, require a higher number of test cases compared to safe classes *i.e.,* without anti-patterns. They conclude that anti-pattern classes should be carefully tested due to their fault proneness compared to other safe classes.

Researchers have shown that test smells are widespread in the software. Bavota *et al.* [16] introduced test smells as code smells related to test files *i.e.,* unit test. They proposed a test smell detection approach which they used to analyse the occurrence of test smells in both open source and industrial software systems. Their results indicate that test smells are highly diffused and impact 86% of JUnit test files. Moreover, they found evidence that test smells impact program comprehension and maintainability – program comprehension is improved by 30% in the absence of test smells.

Peters *et al.* [24] investigate the lifespan of code smells and how developers refactor source code. They analyzed several Java open source systems using a tool called SACSEA. They concluded that despite the importance of treating code smells, engineers often neglect code smells and do not pay great attention to their existence. Besides, Tufano *et al.* [25] explored the presence of test smells in projects, how long test smells could occur in a project over its lifespan, and if their presence is correlated with some code smells. They found that usually test smells are introduced at the beginning of the testing process of the project and they tend to remain in the system for a long time.

Other researchers used machine learning approaches based on the code design to predict bugs. Taba *et al.* [26] showed that the occurrence of a single anti-pattern in files tends to increase the bug density of such files, and they recommend developers to pay more attention on testing activities. Also, they found that traditional code features such as LOC, PRE (Pre-Released bugs), and Churn do not improve bug prediction, but features related to anti-pattern ANA (Average Number of Antipatterns), ACM (Antipattern Complexity feature), and ARL (Antipattern Recurrence Length) can provide additional information related to build failure. They recommend using ARL features based on their improvement to bug prediction models.

In this paper, we built multivariate hierarchically models using multiple dimensions that combine the build failure-related features used by Rausch *et al.* [10] and Zolfagharinia *et al.* [5], with software quality features and test smells by Taba *et al.* [26] and Bavota *et al.* [16]. From these features, we were able to identify the features that have the most impact on build failure.

## 3. Case Study Design

This section presents the overall design of our conceptual replication study, including the selection of projects from the open source TravisTorrent repository, the collection and rationale of features, the identification of code and test smells from source code, and the construction of build failure models. The overall setup of our study is illustrated in Figure 1.

### *3.1. Selection of Case Study Systems*

As previously mentioned in Section 1, the goal of this conceptual replication study is to apply a different experimental procedure (multivariate models) on different subject systems to identify the individual and combined effect of previously identified features on explaining and predicting build failures.

Since we require projects with substantial size such that the number of analyzed builds, features, and build failures are non-trivial (and hence build failure models become beneficial), we purposively sample projects from the TravisTorrent dataset [27]. TravisTorrent automatically applies several criteria to ensure that its dataset contains only non-fork, non-toy, and popular projects (> 10 watchers on GitHub) with a history of more than 50 builds. Purposive sampling allows us to select projects with specific characteristics that are relevant to this study [28].

In order to filter out projects with trivial build activity, we consider the build duration and the number of builds in projects. We extract the build duration from the build logs for each 2-week period, then rank the median build durations of all the projects. We select the top 20 projects with the highest median build duration. Since such projects take a long time to complete their builds, they could benefit the most of build failure prediction models.

Next, we filter out two non-Java projects and three projects with extremely low number of failing builds (11 failing builds in total across the three projects) from our study. We only consider Java projects in our analysis due to the use of the Ptidej tool [29] to detect code smells and Bavota's tools [16] to detect test smells in JUnit test cases.

The remaining 15 projects are further split into two sets based on the number of builds. We require a minimum of 510 builds for each project, which is 34 (the number of features detailed in Section 3.2) multiplied by 15. This sample size requirement ensures adequate discrimination by our models, as recommended by Harrell [30]. The first set contains the 10 projects that meet the minimum requirement for the number of builds. This set of projects is used to answer RQ1, RQ2 and RQ3, and contains 3 of the 14 projects used by Rausch *et al.* [10], but none of the (Perl) projects used by Zolfagharinia *et al.* [5]. It should be noted that we include the *storm* project (having only 477 builds), since it has balanced classes of builds (69% failure classes) and is the project closest to the filter limit. The other 5 projects are later used in our discussion section (Section 5.1) to evaluate our cross-project prediction models for projects with insufficient builds.

We present the chosen 15 projects with a short description, the main characteristics for selection, and the percentages of failed builds in Table 1. The 10 projects used in our research questions are indicated in bold, while the 5 projects used for cross-project prediction are not. Also, as shown in the table, the 15 projects are categorized into balanced or imbalanced projects based on their proportion of failed builds (see column *# Failures in (D)*). Projects with a 30% or less proportion of failed builds are considered imbalanced, *i.e.,* our dataset consists of nine balanced and six imbalanced projects. It should be noted that, While the TravisTorrent dataset has four types of build status (*passed*, *failed*, *errored* and *canceled*), we ignored canceled builds (0.2% of total builds) because they denote an interrupted build process. We consider the passed build status as a successful build, while the failed and errored build

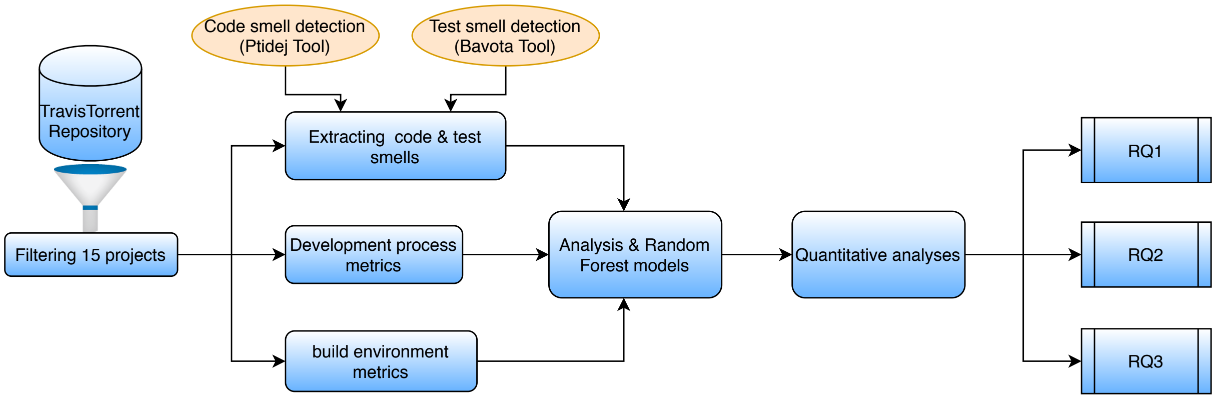
Code smell detection (Ptidej Tool)
Test smell detection (Bavota Tool)
TravisTorrent Repository
Filtering 15 projects
Extracting code & test smells
Development process metrics
build environment metrics
Analysis & Random Forest models
Quantitative analyses
RQ1
RQ2
RQ3

Figure 1: Overview of case study setup.

statuses are considered as failed builds [31].

Finally, due to the requirements of the code and test smell detection tools, we filter out builds that have no Java sources and run no tests. These builds, representing 36.6% of all builds, are kept separately and later analyzed in more detail (see Section 5.2).

### *3.2. Features Selection*

As previously mentioned, this replication study reuses features studied by Rausch *et al.* [10] and Zolfagharinia *et al.* [5], and also adds other kinds of code quality features. We compute code smells using the Ptidej tool [29], and identify test smells using Bavota *et al.*'s approach [16]. Tables 2, 3 and 4 summarize the development process features, code smells and test smells used in this work, respectively. The replicated features of Rausch *et al.* [10] are indicated in bold (in Table 2) while Zolfagharinia *et al.*'s feature(s) [5] are underlined (in Table 2). In what follows, we provide the rationale for selection and the methodology of collection for the non-trivial features.

In the rest of the paper, we use the term "commit" to refer to units of work stored in a version control system like Git, while a "changeset" (sometimes called a "push") is a sequence of commits that, together, are sent to the CI system to be built. Ideally, a changeset should consist of only one commit. However, due to build volume, modern organizations group multiple commits into a changeset to reduce the number of resources needed for the CI infrastructure [21]. The most recent build in a changeset is said to "trigger" the build for the changeset.

**Complexity of changes:** This dimension of data contains information related to size and complexity of code changes; the larger part of a system one needs to modify, the higher the chance the build will fail. Entropy for code changes (CX) measures how spread out each changeset is across the changed files, and is computed using an existing open source tool[1] that we adapt to report entropy across all changed files of a build (containing multiple commits) instead of across a single commit.

**File types:** Changes in certain kinds of project files might be more frequent and failure-prone than others. In our approach, we distinguish between sources, documents and other files based on the file type feature from Travis torrent [27]. We manually added information about changed test files by extracting commits that changed files in test repositories using the approach of Macho *et al.* [32], which specifies that test files should be in a specific *"/src/test"* folder.

**Date and time:** This feature dimension contains the time and weekday that developers make code changes in the build-triggering commit.

**Author:** Several studies in literature show that certain characteristics of developers directly affect the quality of source code and have an impact on build failure. Suzuki *et al.* [33] observed that the experience of committing authors and the number of developers involved with the modified files in a commit are causal factors of build breakage. Eyolfson *et al.* [34] found that daily-committing developers produce less buggy commits, and in that case, experienced developers could be reviewers for other developers' commits. Apart from more traditional experience features like AX and ACF, the CTM feature identifies whether a build-triggering commit was issued by core team member. A core team member is someone who has committed code at least once within the three months before this commit, either directly or by merging commits [27]. We consider the author of the build-triggering commit as the main author of the changeset. In the context of this feature, authors and committers are considered the same — the core team member who pushed a source code changeset.

**Build type:** Similar to Rausch *et al.* [10], we classify different Travis CI build types (BT) based on the type of build-triggering commit. A build triggered by a commit made directly to the master branch is classified as a *PUSH*. Pull request related builds are classified as either a *PR*, if the build is triggered upon the creation of a pull request, or as a *PR-MERGE* when the build is triggered upon merging a pull request into the codebase. For builds triggered by manual merge commits (branches merged locally and then pushed), we classify the build as a *MERGE* if the trig-

[1] https://github.com/GripQA/commit-entropy

Table 1: Characteristics of the selected TravisTorrent projects. Projects in bold meet the minimum requirement of 510 builds (see column # *Useful Builds*).

| Project Name | Description | Median build duration$^*$ | # Builds **(A)** | # Canceled Builds **(B)** | # Builds without Java sources&Test **(C)** | # Useful Builds **(D)=(A)-(B)-(C)** | # Failures in **(C)** | # Failures in **(D)** | Date of 1st build | Date of last build |
|---|---|---|---|---|---|---|---|---|---|---|
| **Geoserver** | Server to share and edit geospatial data | 22.28 | 2996 | 8 | 1058 | 1930 | 508 (31.69%) | 1095 (57%) | 2013-09-28 | 2016-08-31 |
| **Grails-core** | Framework used to build web applications | 39.72 | 2207 | 32 | 1319 | 856 | 399 (57.99%) | 289 (34%) | 2014-05-12 | 2016-08-31 |
| **Apache jackrabbit-oak** | Open source content repository for Java platform | 25.57 | 8205 | 0 | 1405 | 6800 | 589 (17.08%) | 2859 (42%) | 2012-06-18 | 2016-08-31 |
| **Facebook presto** | Distributed SQL query engine for big data | 30.97 | 2153 | 0 | 511 | 1642 | 259 (23.42%) | 847 (52%) | 2013-11-06 | 2015-08-07 |
| **Hubspot Singularity** | Scheduler for running Mesos tasks | 27.93 | 3871 | 1 | 2809 | 1061 | 1451 (76.89%) | 436 (41%) | 2016-02-13 | 2016-08-31 |
| **Apache storm** | Distributed computation framework | 73.93 | 935 | 0 | 458 | 477 | 332 (50.15%) | 330 (69%) | 2016-03-03 | 2016-08-31 |
| aws-sdk-java | Ease the work with Amazon Web Services | 20.05 | 366 | 2 | 132 | 232 | 83 (62%) | 153 (66%) | 05-07-2015 | 08-30-2016 |
| helios | Docker orchestration platform | 46.5 | 389 | 1 | 155 | 233 | 61 (39%) | 64 (37%) | 06-06-2014 | 10-23-2014 |
| jcabi-github | Object-oriented wrapper of Github API | 65.2 | 820 | 0 | 384 | 436 | 188 (48%) | 138 (64%) | 04-11-2014 | 08-31-2016 |
| **Grpc-java** | RPC library and framework for JDK | 39.88 | 1204 | 2 | 272 | 930 | 40 (30.53%) | 91 (10%) | 2015-03-20 | 2016-08-31 |
| **Lenskit** | Tools for benchmarking filtering algorithms | 29.82 | 906 | 3 | 381 | 522 | 112 (61.2%) | 71 (14%) | 2013-09-19 | 2016-07-22 |
| **Dropwizard** | Framework for developing RESTful web services | 44.73 | 1537 | 5 | 596 | 936 | 49 (40.5%) | 72 (8%) | 2013-04-05 | 2016-08-27 |
| **Cloufoundary uaa** | Multi_tenant identity management service | 73.93 | 1208 | 20 | 427 | 761 | 71 (51.45%) | 67 (9%) | 2013-02-04 | 2013-09-19 |
| hivemall | Hive scalable machine learning library | 75.6 | 310 | 5 | 111 | 194 | 37 (33%) | 30 (18%) | 10-30-2014 | 07-15-2016 |
| optiq | Dynamic data management framework | 27.41 | 568 | 0 | 162 | 406 | 31 (19%) | 50 (14%) | 07-16-2013 | 10-19-2014 |

$^*$ In minutes

gering commit is the sole commit in the build's changeset, and as an *INTEGRATION* if the build's changeset contains multiple commits. All other cases are classified as *UNKNOWN* builds.

**Build history:** The build history dimension takes into account the historical state of the builds of projects. Hassan *et al.* [4] show that using previous build results helps to fit prediction models with high accuracy. Within the Travis torrent dataset [27], we used the parent of a given build as the previous build (PB), and recursively iterated over further ancestors to identify the most recent failure for the calculation of days since last failure (DLF). Furthermore, we used $k = 10$ to compute build climate (BC). We were unable to compute build history features for 0.4% of the studied builds due to the lack of parent's build *i.e.,* at the beginning of a project. To deal with such cases of data imputation, we replace the null values by '-1' [35] [36].

**Build environment:** In Travis CI, builds are composed of several jobs depending on the number of environments requested by developers in the build configuration. Therefore, we create our dataset using each environment as an attribute (column) — if there are 5 jobs with different environments, we will have one row (representing the build) with 5 "true" values in different columns. Since our studied Java projects are built on only the Ubuntu OS (by TravisTorrent), we limit our study on build environments to the different versions of Java runtime environments used (e.g., openJDK7 and oracleJDK8).

**Code smells:** Table 3 lists the set of smell features (obtained from [29]) used in this study. To identify code smells in the source code, we followed these steps for each project:

1. Starting from the first commit, we take snapshots of the project's repository using a window of 200 commits. Although related work by Canfora *et al.* [37] considers a period of 500 commits to increase the probability that a source code file is changed within the period, we recommend that a period of 200 commits is large enough and finer-grained to better detect the evolution of code smells in the source code. This decision is based on the high effort to compute code smells and the low likelihood of their refactoring [38].

2. Compute and gather code smells for each selected snapshot using the Ptidej tool [29].

3. Map every single build changeset to its closest snapshot, then assign the identified smells of that snapshot to the changed files of the changeset and its build.

**Test Smells:** We used the approach by Bavota *et al.* [16] to extract five major test smells (described in Table 4) from JUnit test suites and also analyze their impact on program comprehension and maintenance. The steps taken to link the identified test smells to builds are as follows:

1. Retrieve build logs and identify all executed test classes in each build. In some cases when log files were not helpful, due to the used build automation tool (*i.e.,* Gradle, Maven) and how developers configure the log level, we determine test files from the file system and build configuration.

2. Find the source files that were tested by the identified test classes of the build. We search inside test files, and we determine the tested source classes using specific java keywords *i.e.,* import, package, new.

3. Use the approach proposed by Bavota *et al.* [16] to detect the test smells in Table 4.

Please note that we could not retrieve test files for 6.9% of the builds due to missing information in the log files caused by multiple reasons (*i.e.,* cancelled builds, skipped tests, missing logs, and errors in parsing travis.yml files). We

Table 2: Overview of development process features of Rausch *et al.* [10] and Zolfagharinia *et al.* [5]

| Dimension | Feature | Name | Definition |
|---|---|---|---|
| Complexity of changes | **Number of commits** | **NC** | The number of commits that were involved in the actual build |
| | **Number of authors** | **NA** | The number of distinct authors involved in the changeset. |
| | **Number of modified files** | **NMF** | The total number of distinct files changed in all commits of the changeset. |
| | **Number of lines added** | **NLA** | The number of lines added in all commits of the changeset. |
| | **Number of lines removed** | **NLR** | The number of lines removed in all commits of the changeset. |
| | **Change scattering** | **CX** | Shannon entropy measures how concentrated each changeset was across the changed files. |
| File types | **File types** | **FT** | The types of files that were modified in the changeset. For example, if a source and a test file were changed, the value of FT will be "src+test". Rausch *et al.* [10] characterize four categories: *source*, *documentation*, *test*, and *others*. |
| Date and time | **Time of day** | **TD** | The time-zone adjusted time of day [0, 23] of the commit that is triggered the build |
| | **Weekday** | **WD** | The time-zone adjusted weekday [0, 6] (0 refer to Monday) of the commit that triggered the build. |
| Author | **Author experience** | **AX** | The time difference in days between the first commit and the current build of the author of the commit that triggered the build of the changeset. |
| | **Author commit frequency** | **ACF** | The author's most occurring time difference between consecutive commits. We categorize the commit frequency into: daily, weekly, monthly, other (<20 commits), and single (1 commit). |
| | Core team member | CTM | Whether this commit was authored by a developer who has committed code at least once within the last three months before this commit. |
| Build type | **Build type** | **BT** | The build type BT $\in$ {PUSH, PR, MERGE, INTEGRATION, PR MERGE, UNKNOWN}. First, we determine the event type (pull request or push) from the Travis Torrent metadata. Then, we determine from the changeset whether the build is a push, merge, integration commit or pull request merge in case we have sole merging trigger commit in the changeset. |
| Build history | **Previous build** | **PB** | The result of the previous build(s). In most cases we have one or two predecessors. We consider a previous build as a fail if one or both predecessors are fail. There are some builds without previous (first build in actual or different branches) |
| | **Build climate** | **BC** | The build climate, or build stability is the percentage of the last *k* builds that failed. |
| | **Days since last fail** | **DLF** | The time taken since the last reported build failure. We discretize the variable using four intervals. 1) Less than a day ago, 2) A day ago, 3) Less than a week ago, 4) More than a week ago. |
| Build environment | Build environment | BE | The build environment where the build was launched, in our case, it varies between different versions of the Java JDK: openjdk6, openjdk7, oraclejdk7, oraclejdk8. |

Table 3: Overview of code smells features [29]

| Dimension | Feature | Name | Definition |
|---|---|---|---|
| Code Smells | Anti Singleton | AS | A class that provides mutable class variables that could be used as global variables. |
| | Base Class Should Be Abstract | BCSBA | A class that has many subclasses without being abstract. |
| | Blob | Bl | A large controller class that declares many fields and methods with a low cohesion. It monopolises most of the processing, and takes most of the decisions. |
| | Complex Class | CompC | A class that has (at least) one large and complex method, in terms of cyclomatic complexity and LOCs. |
| | Large Class | LC | A class that has grown too large in terms of LOCs. |
| | Lazy Class | LazyC | A class that has few fields and methods. |
| | Long Method | LM | A class that has (at least) a method that is very long based on LOCs. |
| | Long Parameter List | LPL | A class that has (at least) one method with a too long list of parameters. |
| | Many Field Attributes But Not Complex | MFABNC | A class that declares many attributes but which is not complex and, hence, more likely to be some kind of data class holding values without providing behaviour. |
| | Refused Parent Bequest | RPB | A class that redefines inherited method using empty bodies, thus breaking polymorphism. |
| | Spaghetti Code | SC | A class with no structure, declaring long methods with no parameters, and utilizing global variables. It does not use object orientation mechanisms. |
| | Speculative Generality | SG | A class that is defined as abstract but that has very few children, which do not make use of its methods. |

Table 4: Overview of test smell features [16]

| Dimension | Features | Name | Definition |
|---|---|---|---|
| TestSmells | Assertion Roulette | AR | JUnit classes containing at least one method having more than one assertion statement, and having at least one assertion statement without explanation. |
| | General Fixture | GF | JUnit classes having at least one method not using the entire test fixture defined in the *setUp()* method |
| | Indirect Testing | IT | JUnit classes invoking, besides methods of the tested class, methods of other classes in the production code |
| | Sensitive Equality | SE | JUnit classes having at least one assert statement invoking a toString method. |
| | Mystery Guest | MG | JUnit classes that use an external resource (e.g., a file or database). |

were, however, able to repair almost 71% of these builds using the list of executed tests in the closest previous build. We were incapable of repairing the remaining 29% due to missing previous builds.

### *3.3. Building Models*

In this section, we present the process followed to build our build failure explanatory and prediction models, based on the extracted 34 features (along 9 dimensions, as cited in Table 2, 3 and 4) from 15 studied projects. In the following, we first present the different models used to understand the extent to which build failures can be explained.

### *3.3.1. RQ1. Fitting Explanatory Models*

We use a standard, hierarchical model building approach, in which we build nine explanatory models for each project, each time adding an additional dimension of features in order to know its impact on improving models. Knowledge of the major feature dimensions helps practitioners prioritize their effort in calculating features for their own models. The order in which the dimensions are added to our hierarchical model is obtained from a preliminary stepwise backward elimination approach on the entire dataset. Stepwise backward elimination [39] starts with a model containing the full set of features, and in a number of iterations, removes the worst feature from the model after each iteration (based on mean decrease in impurity [40]).

The outcome of the stepwise backward elimination process is an ordered list of features. Since our approach is primarily interested in the order of dimensions instead of the order of features, we group the ordered features by dimension and then record their median values. These median values are used to obtain the order of dimensions. Thus, our baseline model uses only complexity dimension features — the best dimension based on our backward elimination procedure. Then, step by step, we add a new dimension and identify any improvements in explaining build failures.

To build the models, we applied Random Forest classification [40] that uses a majority voting of decision trees to create a classification. This algorithm generates an ensemble that can configure Random Forest to bias and low variance [41]. Random Forest classifiers have been shown to achieve the best performance in predicting build failure [17].

In this study, we build 1000 trees that contain 5 randomly selected features each. To measure the fitness of the models, we apply ten-fold cross-validation [42], which randomly splits the subject build into ten disjoint sets. Nine of them are used as training data and the remaining one as testing data. We repeat this process ten times and report the overall results for precision, recall, and the area under the curve (AUC).

We used McNemar's statistical test, recommended by Thomas Dietterich [43], to compare between the hierarchical models to evaluate whether there is a significant improvement from a model to another, *i.e.,* to determine whether a dimension of features is enhancing the model. In particular, we use a 99% confidence level (*i.e.,* $\alpha = 0.01$) to interpret if there is a significant improvement between two binary classifier. Since we perform a comparison of a classifier with his predecessor and successor, we applied the Bonferroni correction [44] to control the familywise error rate. Concretely, we divide our $\alpha$ by the number of comparisons, *i.e.,* $\alpha = 0.01/2 = 0.005$.

In the dataset, there are some projects that have an imbalanced percentage of build failures, as indicated in Table 1, which can lead to biases and inaccuracy in the results [45]. Kotipalli *et al.* [46] show that performance degrades as the majority-minority ratio increases. As a result, the SMOTE multi-class re-sampling method [47] has been proposed as a technique to increase the number of minority cases by generating synthetic examples in the neighborhood of observed build failure minority classes. However, applying the same SMOTE ratio irrespective of the majority-minority ratio results in un-uniform performance improvements [46]. In an attempt to make the overall performance more uniform, we apply different SMOTE re-sampling level ratios as follows:

- Percentage of failure from 8% to 10% → ratio of 1/4 from majority classes
- Percentage of failure from 10% to 20% → ratio of 3/7 from majority classes
- Percentage of failure from 20% to 30% → ratio of 2/3 from majority classes
- Percentage of failure more than 30% → no SMOTE

### *3.3.2. RQ2. Most Important Features*

After determining the dimensions with the highest impact in our models, we built an additional model that includes all features of a given project to analyze which individual features have the highest impact in the model. We followed the **effect size analysis** used by Shihab *et al.* [48] and Mockus *et al.* [49].

The effect size outcome helps to know the impact of each feature, either a positive or negative effect, on the failure proneness of a project. The effect size approach first creates a baseline random forest model using the median value of each feature as input. The mode (most appearing value) is used for categorical variables like File Type. To evaluate which features have the most significant impact on the models, we create a new model for each feature. Each model replaces the input value for the studied variable with its median plus one standard deviation (for categorical values, the second most current value replaces the mode), but keeps the median value for the rest of the variables. The effect size of each feature on build breakage can then be calculated, based on a comparison with the baseline model, as $\frac{newvalue-basevalue}{basevalue}$. We repeat this process for all 10 projects.

Finally, the effect sizes of the features can be compared to each other to find the features with the most significant impact on build failure. A positive effect size indicates that an increase in the feature has a high correlation with an increase in build failures. On the other hand, the more of a negative effect size feature in a project, the lesser the likelihood of a build failure. For example an effect size of 0.5 indicates 50% increase in failure-proneness, and vice-versa for a negative effect size.

### *3.3.3. RQ3. Predicting Future Builds*

We used the most impactful features identified in RQ1 and RQ2 to build models predicting future failure. To predict future builds, we ordered our data chronologically. We split up the dataset in sets of 200 builds, then build and evaluate models exploiting the sequential ordering of the sets. First, we use a model trained on the first set to predict the next set. Then, the one which was predicted becomes the training set for a new model that will predict the subsequent set, and so on.

## 4. Study Results

In this section, we provide answers to the research questions posed in Section 1. For each question, we describe the motivation behind the question, the approach to address the question, and present our findings.

*RQ1: How well do models based on the considered factors explain build failures?*

***Motivation:*** Previous works have studied the correlation between features and failures within software systems. Rausch *et al.* [10] and Zolfagharina *et al.* [5] found that different kinds of features (*e.g.,* development process, environment) have a direct correlation with build failures. Even though these results were promising, they analyze the correlation between build failures and one single feature at a time. We replicate these earlier works on additional features, all while using multivariate models instead of the existing independent statistical tests/correlations. Our study on both the individual and cumulative effect of these features is needed to better understand the relative impact of a feature on build failure, since individual features might correlate with each other, which becomes clear while building multivariate models.

***Approach:*** To understand how much we could utilize the information from the different feature dimensions in Tables 2, 3 and 4, we hierarchically build nine models using the random forest algorithm as described in section 3.3. Additionally, we use the SMOTE statistical technique to correct any imbalance within our training set. Table 5 summarizes the results for each project. The models that were significantly improved, according to results of McNemar's statistical test when compared to previous models (with fewer features), are in bold in Table 5.

***Results:***

**For *balanced projects*, we found that the median precision (recall) across projects increased from 58.3% (52.4%) in the baseline model to 79.0% (69.6%) in the final model (adding the file type dimension).** We note the largest improvement in model 2 (build history) in terms of median precision (recall) of almost 13.6% (14.7%) compared to the previous model (complexity). We found that the median AUC kept increasing after each additional model. The median AUC range starts from 56.2% in the complexity model to 75.3% in the file type model. The better median AUC was noticed in the model 5 (author), which attains 76.3%. Based on McNemar's statistical test, we note the highest number of improvements in the Build History model (4 out of 6 projects), Date&Time (2 out of 6 projects) and Author model (1 out of 6 projects).

However, **for *imbalanced projects*, we observed low and fluctuating prediction results each time a dimension is added; starting from a median precision (recall) from 2.5% (2.6%) in the baseline complexity model to 37.5% (12.4%) in the file type model.** We found a median AUC of 49.0% in the complexity model, which later improved to 55.3% in the file type model. Yet, this still hardly outperforms a random model. While these projects contain a large amount of data, they have a very low percentage of build failures, varying between 8% and 14%. Although the SMOTE technique was applied in these cases, our models still have difficulties analyzing these imbalanced datasets. The low performance of models on imbalanced datasets is still an ongoing challenge. As such, our findings confirm those of a recent study by Jin and Servant [22], which also showed a relatively low prediction performance in 10-fold models, with an interquartile precision ranging between 15% and 46%, and recall between 0% and 30%.

With the exception of the models that introduce the *Build History* (model2) and *Author* (model5) dimensions, all other models do not show any noticeable statistical improvements. We perform further analysis (in RQ2) to better understand these findings by studying the effect of the individual features within the models.

> *The precision, recall and AUC of our models improve substantially as we incrementally add feature dimensions. Our explanatory models achieve a (median)* ***precision of 79.0%****, a* ***recall of 69.6%****, and* ***AUC of 75.3%*** *in balanced projects. However, in imbalanced projects, we attain (median)* ***precision of 35.5%, recall of 12.4%*** *and* ***AUC of 55.3%****.*

*RQ2: Which features are the best indicators of build failures?*

***Motivation:*** After building our explanatory models based on the different dimensions, we would like to perform finer-grained analysis to further understand why some dimensions have more considerable significant improvements in performance than others, and also identify the individual effects of each feature. Determining the dimensions and features that have the highest correlation with build breakage would allow practitioners to choose the essential features in future predictive models, as well as identify features that increase/decrease the possibility of build failure. Consequently, knowing the effect of individual features will help communities apply best practices (e.g., avoid inflation of specific code smells) before launching a build.

***Approach:*** We performed an analysis to measure the effect sizes of our features (as described in Section 3.3.2). We first create a baseline random forest model using the median values of each feature (obtained from *model9* in RQ1) as input. Then, for one feature at a time, we create a new 10-fold cross-validation model that increases the value of the (one) feature by its standard deviation but keeps the same value for all other features.

Using the outcome of these models, we are able to find out the sensitivity of each feature (*i.e.,* a positive or negative effect on the build failure). Figure 2 shows the effect sizes for each feature across the studied projects. A positive effect of a feature means that the increase of that feature correlates with more build failure.

We present, in the following, the results of feature effect size (in Figure 2) by dimension of features.

***Results:*** **5 code smells (LPL, MFABNC, AS, LC, SG) and 1 test smell (SE) have a positive effect size, while only a single smell from each category (SC and MG) has a negative impact.** Although SC and MG do not have a direct impact on build failure, these smells were present in 70% and 39% of the failure changeset (of the studied projects), respectively because they are more correlated with the use of external resources and the misuse of OOP principles, and not directly related to the outcome of a project's build. On the other hand, we identified AS in 9.3%, LPL in 44.3%, MFABNC in 1.9%, LC in 2.4%, SG in 1.5% and SE in 20% of the failure changeset. Other test/code smells features such as long method,

Table 5: Overview of Precision, Recall, Area Under the Curve (AUC) of Random Forest Models for the balanced and unbalanced projects (percentages in bold represent significant improvement between two binary classifiers with 99% of confidence level and Bonferroni correction *i.e.,* $\alpha = 0.01/2 = 0.005$).

| | | Balanced projects | | | | | | Imbalanced projects | | | | Median of model performance | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | geoserver | grails-core | jackrabbit-oak | presto | Singularity | storm | grpc-java | lenskit | dropwizard | uaa | **Median Balanced** | **Median Imbalanced** |
| **Model1:**<br>Complexity | Precision | **63.5** | 46.5 | 46 | 58.3 | 58.3 | 69.5 | 5 | 0 | 0 | 50 | **58.3** | **2.5** |
| | Recall | 70.7 | 24.6 | 32.3 | 57.7 | 47.2 | 82.4 | 5 | 0 | 0 | 26.8 | **52.4** | **2.5** |
| | Auc | 58.7 | 55.4 | 52.7 | 57.1 | 59.4 | 49.5 | 49.2 | 47.2 | 48.8 | 61.6 | **56.2** | **49.0** |
| **Model2:**<br>Build History | Precision | **80.1** | **73.1** | **70.7** | **76.3** | 55.8 | 70.4 | **40** | 0 | **50** | 45 | **71.9** | **42.5** |
| | Recall | **76.4** | **55.2** | **63.7** | **70.6** | 49.1 | 85.3 | **21.1** | 0 | **26.8** | 14.3 | **67.2** | **17.7** |
| | Auc | **76.3** | **72.4** | **72.6** | **75.3** | 58.1 | 51 | **59.1** | 48.9 | **62.5** | 56 | **72.5** | **57.5** |
| **Model3:**<br>Date&Time | Precision | 80.8 | 79.1 | **73.7** | **80.4** | 57.6 | 72.3 | 41.7 | 12.5 | 50 | 50 | **76.4** | **45.8** |
| | Recall | 78.2 | 59.7 | **63.1** | **73.5** | 53.8 | 88.2 | 16.7 | 6.3 | 19.6 | 14.3 | **68.3** | **15.5** |
| | Auc | 77.2 | 75.7 | **73.8** | **77.8** | 60.2 | 55.1 | 57.4 | 51.5 | 59.2 | 56.3 | **74.7** | **56.9** |
| **Model4:**<br>Build Env | Precision | 81.9 | 80.1 | 73.6 | 81.3 | 59 | 71.4 | 36.7 | 10 | 41.7 | 50 | **76.8** | **39.2** |
| | Recall | 79.0 | 62.1 | 62.6 | 74.7 | 55.1 | 88.2 | 16.7 | 6.3 | 19.6 | 13.4 | **68.7** | **15** |
| | Auc | 78.4 | 77 | 73 | 77.7 | 61 | 51.6 | 56.9 | 51 | 59.2 | 55.5 | **75** | **56.2** |
| **Model5:**<br>Author | Precision | 82.2 | 80.5 | **76.7** | 80.7 | 60.9 | 72.3 | 36.5 | 0 | **87.5** | 41.7 | **78.6** | **39.1** |
| | Recall | 80.7 | 59.7 | **65.6** | 76.5 | 57 | 89.7 | 20 | 0 | **26.8** | 12.5 | **71** | **16.2** |
| | Auc | 78.4 | 77.2 | **75.3** | 78 | 62.9 | 53.5 | 58.5 | 48.9 | **63.4** | 56 | **76.3** | **57.2** |
| **Model6:**<br>Test Smells | Precision | 81.9 | 75 | 76.8 | 81.1 | 65 | 72.1 | 40 | 0 | 63.3 | 61.9 | **75.9** | **51** |
| | Recall | 82 | 58.6 | 65.9 | 74.7 | 59.4 | 88.2 | 21.1 | 0 | 25 | 26.8 | **70.3** | **23.1** |
| | Auc | 78.5 | 74.1 | 75.8 | 77.1 | 66.4 | 54.3 | 58.8 | 50 | 62.2 | 62 | **75** | **60.4** |
| **Model7:**<br>Code Smells | Precision | 82.2 | 80 | 75.8 | 80.8 | 65.3 | 72.6 | 26.7 | 0 | 66.7 | 66.7 | **77.9** | **46.7** |
| | Recall | 79.0 | 58.6 | 65.9 | 74.7 | 56.6 | 88.2 | 10.6 | 0 | 19.6 | 19.6 | **70.3** | **15.1** |
| | Auc | 78.2 | 75.7 | 75.4 | 78.7 | 65.3 | 54.9 | 53.5 | 50 | 59.5 | 59.8 | **75.6** | **56.5** |
| **Model8:**<br>Build Type | Precision | 82.1 | 78.6 | 77 | 80 | 70.2 | 71.5 | 41.7 | 0 | 58.3 | 58.3 | **77.8** | **50** |
| | Recall | 82.9 | 58.6 | 66.6 | 74.1 | 53.8 | 88.2 | 15.6 | 0 | 19.6 | 14.3 | **70.4** | **14.9** |
| | Auc | 78.7 | 74.5 | 77 | 77.1 | 66.7 | 53.5 | 56 | 48.9 | 59.2 | 56.8 | **75.7** | **56.4** |
| **Model9:**<br>File Type | Precision | 82.5 | 83 | 77 | 81 | 70 | 70.9 | 25 | 0 | 50 | 63.3 | **79.0** | **37.5** |
| | Recall | 82.5 | 56.9 | 66.3 | 72.9 | 59.4 | 88.2 | 10.6 | 0 | 25 | 14.3 | **69.6** | **12.4** |
| | Auc | 78.7 | 75 | 75.7 | 77.3 | 68 | 51.9 | 53.5 | 50 | 62.5 | 57.1 | **75.3** | **55.3** |

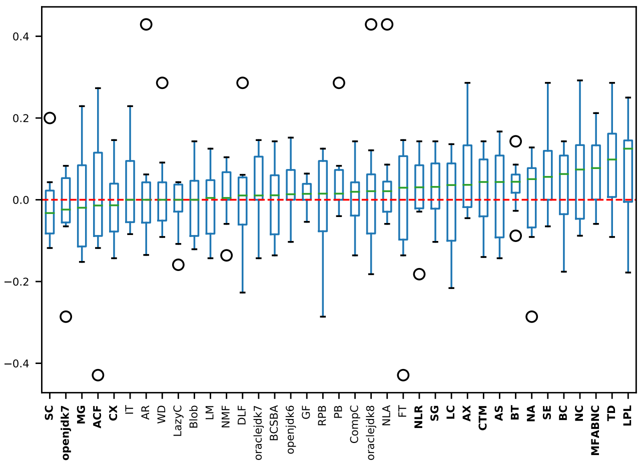

Figure 2: Distribution of Effect Size of all features across the 10 projects

blob, assertion roulette show neither positive nor negative effects on build failure. Although code and test smells are mostly about readability, understandability and maintainability of the codebase, smells manifest into defects that can lead to build failure over time [50]. Therefore, developers should be aware of the potential impact of the presence of smells on the overall quality of the codebase.

**Openjdk7 from Build Environment (BE) has a negative effect size, while Openjdk6, Oraclejdk7 and Oraclejdk8 have positive effect sizees**. Although, build environment has no remarkable improvement in the performance of RQ1, we identify openjdk7 as the safest environment for builds — the chances of build failure decreases when more builds use the openjdk7 environment.

**Complexity features, *i.e.,* Number of commits (NC), Number of Author (NA), Number of Lines removed (NLR) and Number of Lines Added (NLA) have positive effect sizes.** Although in RQ1, we were not able to identify the effect of the complexity dimension because it was chosen as a baseline of the explanatory models, these results indicate that projects are more prone to build failure as the complexity of the commits increase. This result supports the findings by Rausch *et al.* [10] that high change complexity leads to an increase in build failure.

**Three features of the build history dimension — Build Climate (BC), Previous Build (PB), and Days since Last Fail (DLF) — have positive effect sizes**. An increase in the recency and number of previous build failures has a high correlation with an increase in build failures. This result supports the findings by Rausch *et al.* [10], as well as, our findings in RQ1 on the importance of the build history dimension in explaining build failures.

**Author Experience (AX) has a positive effect size**. This result shows that developers who build more frequently (*i.e.,* have high AX and ACF) may have a higher failure ratio, as also observed by Seo *et al.* [21]. Our result explains the significant statistical improvement of the author dimension model (model 5) in 1 out of 10 projects (see RQ1). We also confirm the results shown in the replicated study [10].

**Time of Day (TD) has a positive effect size**. Similar to previous studies (e.g., [51]), our results show that commits after midday have a high correlation with build failure.

*Several dimensions in RQ1, such as code/test smells, did not show a statistical improvement. However, individual features* ***(5 code and 1 test smells) have a positive effect size*** *and correlate with build failure. Additionally, individual features of the* ***complexity of changes, build history,*** *and* ***author*** *dimensions have the highest correlation with build failures.*

*RQ3: Can we predict the outcome of future builds?*

***Motivation:*** In RQ1 and RQ2, we trained and tested models that explored the importance of dimensions and features (see Tables 2, 3, and 4) to explain build failures. In this research question, we aim to use the identified important features to predict the outcome of automated CI builds. This will give developers the ability to decide if they should review the changeset (*i.e.,* commits included in the build) or launch the build, depending on their need to optimize time.

***Approach:*** To predict future builds, the dataset for each of the 10 studied projects was ordered chronologically. Then, each successive 200 builds was grouped into one set. We train our model on the first group of 200 builds, and test on the next group of builds. This process is repeated for the subsequent groups of builds (see section 3.3.3 for details). The model used in this prediction was built with a subset of the features. To avoid building a model biased towards the features with positive effect sizes, only the top 50% of such features are included, together with all features with negative effect sizes. In total, the model uses only 19 features (indicated in bold in Figure 2).

Figures 3 and 4 present a comparison of the performance (*i.e.,* precision, recall) between the 10-fold cross-validation explanatory model in RQ1 (model9) and the on-line prediction model. The purpose of this comparison is to identify any performance gap that may exist between the explanatory and on-line models. Please note that within these figures, the first six boxplots (from the left) represent the balanced projects, and the last four represent the imbalanced ones.

***Results:***

**Prediction models achieve a median precision (recall) of 32.7% (31.9%), which is 2.4 (1.2) times lower than the 10-fold cross-validation models.** The median precision of our prediction model for future builds ranges between 24.2% and 80.2% in balanced projects, but less than 20% for imbalanced projects. The median recall is between 31.1% and 72.7% for balanced projects, and between 3.8% and 13.6% for imbalanced projects. In comparison, using 10-fold cross-validation explanatory models, we recorded median precision of 79.0% and 37.5% in balanced and imbalanced projects, respectively. Also, the recall of our 10-fold cross-validation model has a median recall of 69.6% in balanced projects and 12.4% for the imbalanced ones. The results of this RQ shows that the model containing all features obtained high performance — with less effort to identify and calculate the best features, one can obtain a strong model. We noticed that our on-line prediction results, although sometimes low, conform to reported results by Xia *et al.* [17] and Jin and Servant [22]. Xia *et al.* [17] used different classifiers to predict on-line build failures on the same

TravisTorrent dataset used in this paper [27]. Their random forest classifier achieved a median AUC around 0.54 and an F1 score of 10% across their studied projects. Similarly, we computed an F1 score of 29.3% in predicting on-line build failure across studied projects. This outcome could be attributed to imbalanced classes in the projects. In the subsequent section, we discuss alternative approaches that can improve the prediction results for imbalanced projects.

*We are able to predict future builds with a precision-recall ranging between 25% and 80% for balanced projects; for imbalanced projects, our performance fluctuates with the highest precision (recall) of 20% (15%).*

## 5. Discussion and Implications

In this section, we provide in-depth discussions (and further analysis) on how our models can be used to still predict build failures in projects with insufficient data: few failing builds, fewer number of total builds, or builds without source code. In addition, we discuss and compare the results of our replicated studies and their implications, as well as the observed correlation between build failures and the new code/test smell features.

### *5.1. Predicting build failures in projects with insufficient data*

As our results in RQ1 and RQ3 have shown, despite the efforts to oversample failing builds within our dataset through the SMOTE approach (see Section 3.3.1), imbalanced datasets present a big concern to the performance of our prediction models. Thus, in this section we first investigate the number of failed builds required to deliver

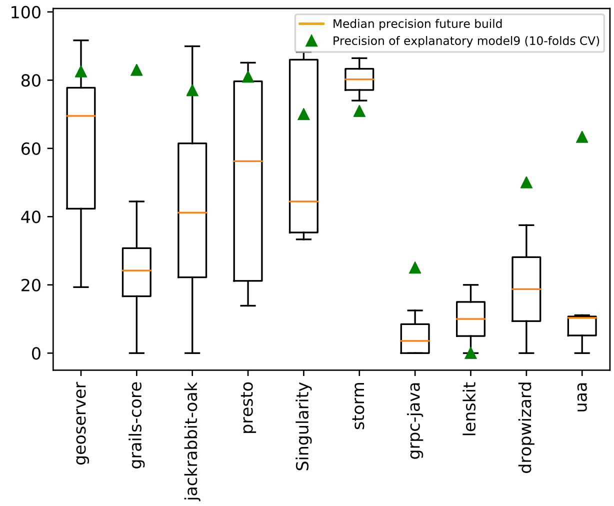

Figure 3: Distribution of precision of predicting Future Builds across the 200-commit folds compared to the precision of the 10-fold CV models (green triangle)

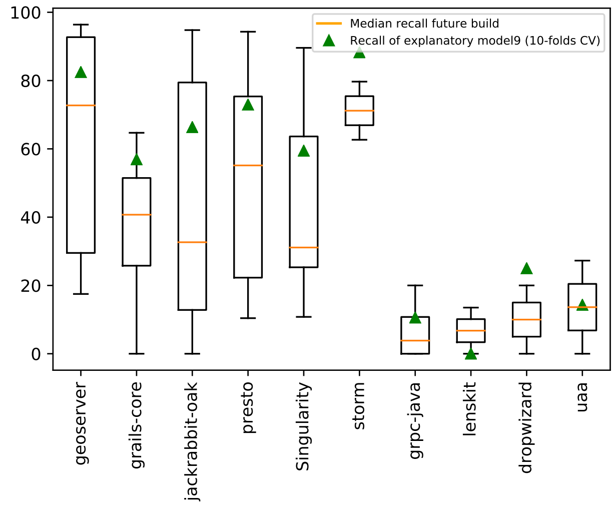

Figure 4: Distribution of recall of predicting Future Builds across the 200-commit folds compared to the precision of the 10-fold CV models (green triangle)

reasonably robust prediction results. A reasonable robust result in this context is expected to be better than a random guess (50% precision/recall). Secondly, we investigate if our models trained on balanced projects can be used to predict failures in different sets of projects (cross-project prediction).

Using sensitivity analysis, we analyze the impact of the number of failing builds on the performance of build prediction in our set of balanced projects. We repeatedly run our model from RQ1 on the balanced projects, each time randomly removing 5% of the failing builds in a project. For example, given the *storm* project with an initial 69% of failing builds, we run our model 14 times, each time reducing the percentage of failed builds till attaining a 5% proportion of failing builds. Figures 5 and 6, respectively, show the precision and recall trendlines of our analysis including a highlight (dotted green line) of the median precision/ recall of imbalanced projects.

**As shown, a project requires at least 15% to achieve a reasonably robust precision measure**; the median precision attained by imbalanced projects (from model9 RQ1) was 37.5% with a build failure proportion of 11%. On the other hand, **40% of failing builds are required for a reasonably robust recall measure**. Unfortunately, given our classification of imbalanced projects as those with less than 30% failed builds, such projects may always suffer from poor recall results as seen for the imbalanced projects with the median recall of model9 in RQ1 (12.4% recall based on a 12% proportion of failed builds).

Thus, we also investigate whether cross-project (inter-project) prediction may obtain better results in imbalanced projects than our current intra-project models. Using a model trained on the complete set of features (model9 in RQ1) of the six balanced projects, we attempt to predict the build outcome of the four imbalanced projects in our research question analysis. The precision and recall results are shown in Figures 7 and 8, respectively.

**We observe an improvement in both precision and recall for all imbalanced projects, except the *uua* project.** There is an average improvement of 26.7% (23%) in precision (recall) across three imbalanced projects. The precision (recall) of the uua project, however, decreased by 42.8% (3.8%).

Using the same cross-project models, we perform another investigation on the five projects with the lowest number of total builds. It should be noted that although these projects were not included in our research question analysis (see Section 3.1), we still run model9, our explanatory model from RQ1, on these projects. Figures 9 and 10 show a comparison of the precision and recall results, respectively, of model9 (RQ1) and cross-project prediction.

**There is little to no improvement in either precision or recall for the 5 projects with fewer total builds when using cross-project models compared to the explanatory models from RQ1.** Four out of the five analysed projects (*aws-sdk-java*, *hivemail*, *jcabi-github* and *optiq*) gain a slight improvement in precision (an average of 6.7%); the precision of the other projects (*helios*) decreased by 5.6%. Similarly, only two projects (*hivemail* and *optiq*) showed an increase in recall (an average of 7.7%); the recall of the remaining three projects decreased by 17%, on average.

As our results have shown, despite the potential improvements gained from cross-project prediction, imbalanced datasets and insufficient number of builds create a bottleneck in the performance of our models. This highlights the need for additional research for predicting build failures in imbalanced and small projects (cold starts).

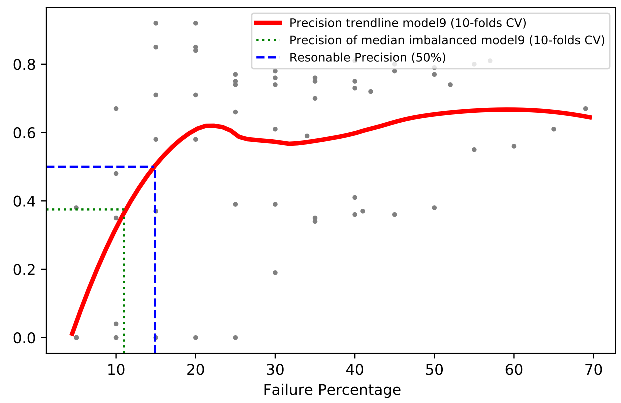

Figure 5: Sensitivity analysis of precision in terms of failure percentage in balanced projects.

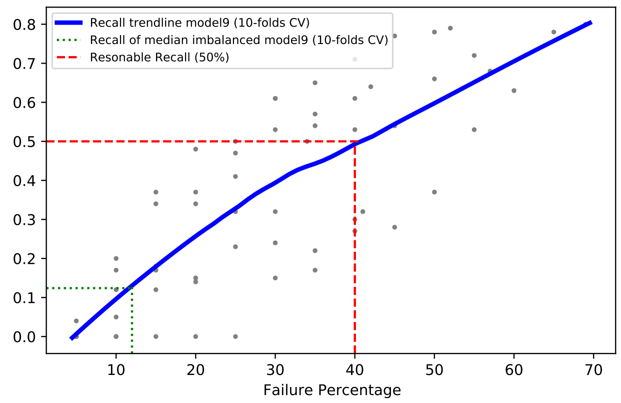

Figure 6: Sensitivity analysis of recall in terms of failure percentage in balanced projects.

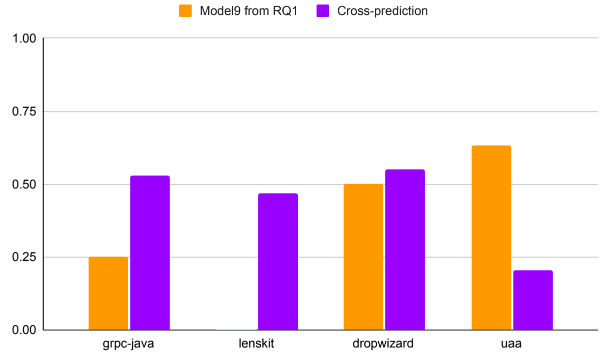

Figure 7: Comparison of the precision of build failure prediction in imbalanced projects based on the intra-project (model9 from RQ1) and cross-project prediction models.

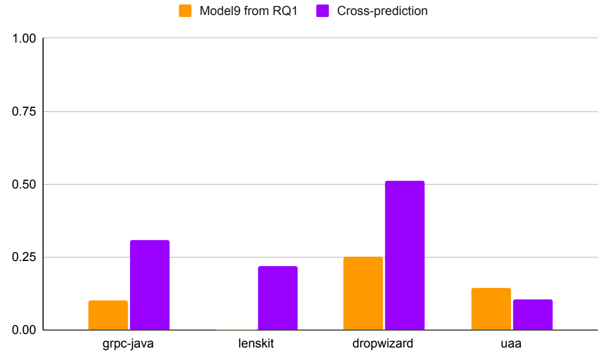

Figure 8: Comparison of the recall of build failure prediction in imbalanced projects based on the intra-project (model9 from RQ1) and cross-project prediction models.

### *5.2. Prediction on builds without Java source code*

As previously discussed in Section 3.1, we removed builds without Java sources or executed tests from our analysis due to the requirements of the code and test smell detection tools. Since these kinds of builds frequently occur, representing 36% of the total studied builds, we want to validate the usefulness of our models in such cases. We create an explanatory model (model-nosources) containing features from 7 dimensions (all dimensions except the code and test smell dimensions), but using only the dataset of builds without source code or executed test (column C in Table 1). Figure 11 shows a comparison of the performance (*i.e.,* precision, recall) between the new model (model-nosources) and the model with source code (model9 from RQ1) across the studied projects.

**We found that model-nosources obtains a median precision (recall) of 57% (41%).** Compared to a median precision (recall) of 70% (50%) for the source code models, our explanatory model (model-nosources) still performs adequately on entities without source code, including balanced and imbalanced projects. Developers can, therefore, use our proposed models to predict build failures on changesets even if they do not contain any source code or executed tests.

### *5.3. Comparison of original with replicated findings and its implications*

**Our results confirm most of Rausch *et al.*'s findings**, as shown in Table 6. The results in RQ2 affirm Rausch *et al.*'s observation that build failures mostly occur consecutively; we observe a positive effect size of the Previous Build feature. The results also show that Build Climate has an impact on build failure *i.e.,* the higher the ratio of build failures in the last 10 builds, the higher the probability of the next build failing.. Thus, maintaining a stable build environment is of utmost importance to all practitioners.

The results of our effect size (Figure 2) also confirmed the importance of Complexity features and Author Experience to build failure prediction. Regarding the author feature dimension, we found that the Core Team Member (CTM) has a positive impact on the build failure; authors that commit more frequently have a higher chance of introducing build failures in our study. This corroborates with the existing research on models and tools to recommend the best code reviewers for a given code change (e.g.,

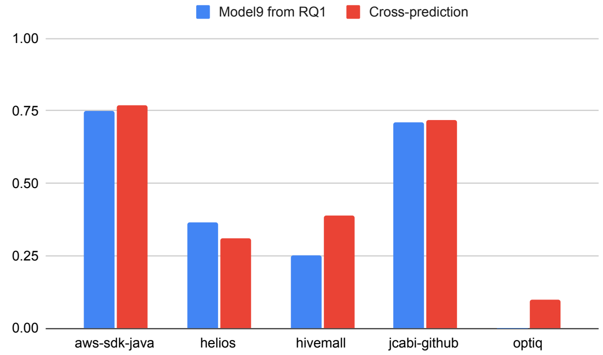

Figure 9: Comparison of the precision of build failure prediction in projects with fewer number of total builds based on the intra-project (model9 from RQ1) and cross-project prediction models.

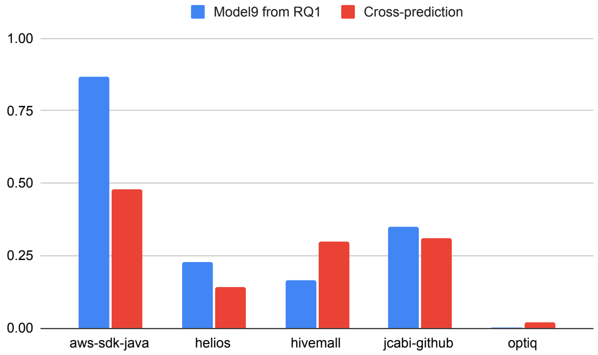

Figure 10: Comparison of the precision of build failure prediction in projects with fewer number of total builds based on the intra-project (model9 from RQ1) and cross-project prediction models.

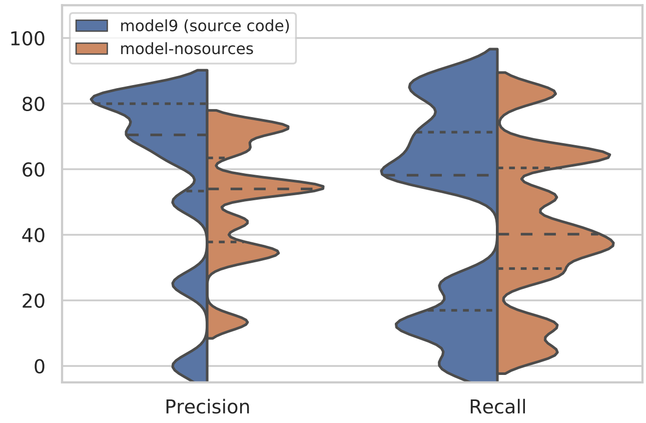

Figure 11: Comparison between performance of models created from dataset with (model9) and without (model-nosources) source code.

Table 6: Summary of replication results

| **Author** | **Key Findings in original study** | **Confirmed in replication?** |
|---|---|---|
| Rausch *et al.* [10] | High change complexity leads to an increase in software build failure | Yes |
| | No evidence on whether changes to a specific file type lead to errors more frequently | No |
| | Little evidence of correlation between the date or time of a change and the build results | Partially |
| | Authors that commit less frequently tend to cause fewer build failures | Yes |
| | Build type has a significant influence on the build outcome | Yes |
| | PRs tend to cause failures more frequently than changes pushed directly into a branch | No |
| | Previous build results have a significant effect of the current build's outcome,as build failures mostly occur consecutively | Yes |
| Zolfagharinia *et al.* [5] | 77% of builds succeed across all versions of the runtime environment | Partially |
| | 6% of builds fail across all versions of the runtime environment | Partially |
| | Results of the remaining 17% builds fluctuates based on the chosen runtime environment | Partially |

[52, 53]). Such tools will enable experts to review and validate changes before they are integrated into the main codebase, effectively decreasing the likelihood of build failures. Furthermore, given our observation that daily committers (based on the ACF feature) may not be experts, i.e., 91% of all failed builds were triggered by daily committers (compared to 4.7% by the monthly committers), more research is needed to streamline the identification of experts within a project.

We also found that while the Time of Day feature has a high positive effect size, there is no observed effect for the Weekday feature. This is similar to the findings reported in the replicated study [10]. Due to this high impact of the time of the day feature, we encourage teams to try to understand what happens at those risky moments, for example in terms of decrease in focus (e.g., late-night development, lunch time or Friday afternoon), or related to "peak moments" in terms of build activity. Managers should put measures in place to convince developers to instead hold on to those changes and perhaps revisit them at a a later time instead of submitting them to the build servers in the heat of the moment. Further studies should analyze Time of Day in more detail.

Similar to Rausch *et al.* [10], we found a relationship between file type changes and the build outcome. We observe a high correlation between changes in source/test files and build failure — 41.5% and 37.1% of failures were directly related to changes in source and test files respectively. Rausch *et al.* [10], however, were unable to find such conclusive evidence as they observed a high number of build failures related to other file types such as README.md.

Contrary to the findings of Rausch *et al.* [10], we observe pushed changes to cause more failures than pull requests. This emphasizes our initial recommendation for the implementation of effective code reviewing mechanisms.

**Our results partially confirm Zolfagharinia *et al.*'s findings.** While their study focused mainly on the influence of different operating systems (e.g., Windows, Linux) and runtime environments (Perl) on build outcome, our work studied the influence of the different JAVA runtime environments used within the builds (due to the studied projects that use limited environments only for java projects).

Despite the difference in analyzed environments, both approaches identify that builds do not always succeed on all environments. We observed 59% and 31% success and failure rates, respectively, across all environments. 10% of the builds fail on previous versions of the environments but succeed on latter versions (e.g., OpenJDK6 vs OpenJDK7). Given that multiple environments in a project introduce an inflation of builds, analyzing and identifying such cross-environment build failures is not a trivial task. Though significant strides have been made in the domain of build prediction, additional research is needed for cross-environment build prediction. Traceability tools, based on such research, can be used to help developers identify the build environments associated to any code change.

**The results of our multivariate analysis (not considered in the original studies) provides additional implications for stakeholders.** We found that the outcome of a build was impacted by code smells such as *Antisingleton (AS)*, *Long Parameter List (LPL)* and *Many Field Attributes But Not Complex (MFABNC)*, and test smells such as *Sensitive Equality (SE)* — the more we do have these smells, the higher the likelihood of failure. These three code smells were present in 18.5% of all failed builds, on average, while the SE test smell was present in 20% of all the builds across projects. The SE smell usually occurs when the implementation of a *toString()* method change might result in a test case failure [54] that breaks the entire build. In addition, we found that although the *Mystery Guest* smell has a negative effect size and no direct impact on build failure, it was present in 39% of build failures across the studied projects. Such tests are difficult to comprehend and maintain, due to the lack of information to understand them [54].

Code and test smells impact the maintainability and readability of systems, as well as, the frequency source code is changed [13, 55], ultimately, making systems more fault-prone. Developers should be aware of the correlation of these code and test smells and try to fix them as soon they emerge in the system; their presence may introduce inconsistent build outcomes due to the possibility of flaky test results. Projects and developers are recommended to implement a layer in their build system to detect and suggest refactorings based on the chosen code/test smells before executing the build.

Finally, as shown in our results, imbalanced datasets always pose a challenge to the performance of ML models, and using techniques such as SMOTE does not always provide additional benefits. Our analysis in Section 5.1 shows that cross-project prediction models (trained on balanced projects) can provide reasonable results on imbalanced datasets. However, more research is needed to help choose the right projects and datasets to train such cross-project prediction models; such approaches should take factors such as the domain of projects and category of build failures into account. We also suggest for researchers to use the sensitivity analysis in their machine learning models to better estimate the performance limits of their proposed models, especially with imbalanced datasets.

## 6. Threats to Validity

In this section, we examine the most significant threats to the validity of our study.

***Construct validity.*** 6.8% of all builds in the dataset did not have log files available. In these particular cases, we were not able to identify which test files were executed. To minimize this threat, we used the test cases executed in the preceded build. We were able to recover 5% of the executed tests; the remaining 1.8% were not resolved due to unavailable previous build information.

Also, it was not feasible to analyze smells for each change in the dataset (on each build) due to the amount of time the process takes. To mitigate this threat, we analyze code smells for a window of 200 builds, as described in Section 3.2. This could impact the results of our smell detection since a file (containing smells) could be modified within the same analyzed window. However, Chatzigeorgiou and Manakos [56] have shown that smells in source code persist over several versions unless refactoring activities are performed. When computing the build climate and previous build features, 0.4% of builds did not have enough previous builds. In such cases, we assign an unknown parent by replacing null values with "-1" [35] [36].

Although the grails-core project is characterized mainly as a Java project, it also contains almost 54% groovy files. However, the impact of this is mitigated since our analysis only considers builds that contained java sources and tests.

***Internal validity*.** Our results could be affected by the precision of the tools that we used to extract code smells [29] and test smells [16]. As a result, we could have missing results for both test and code smells. However, due to the large number of builds analyzed in this work, we are confident that the risk of bias is greatly reduced.

***External validity.*** In this work, we only considered Java-based projects utilizing Travis-CI and either the Maven or Gradle build frameworks. Further, we only explored projects utilizing Git and hosted on GitHub. Our results might not be generalized for other version control workflows. Also, since we only study projects of substantial size (based on their median build time and number of releases), our results may not be generalizable for smaller OSS projects, particularly those that have less number of CI builds.

## 7. Conclusions and Future Work

We perform a replication of two earlier studies [10, 5] using a different data set and additional features from two more dimensions using multivariate models based on these features to help explain and predict future builds. During our analysis we answer the following research questions:

1. How well do models based on the considered factors explain build failures? Our models achieve (median) **precision of 79%**, a **recall of 69.6%**, and **AUC of 75.3%** in *balanced projects*. However, in *imbalanced projects*, we attain (median) **precision of 37.5%**, **recall of 12.4%** and **AUC of 55.3%**.

2. Which features are the best indicators of build failures? **5 code and 1 test smells have a positive effect size with build failure. time of the day**, **complexity dimension entities**, **build history entities**, **author experience** have the highest effect size features. Developers can use our global model containing features with a high effect size for build failure analysis. Based on our findings, developers are invited to review their changesets that contain code smells (**long parameter list**, **many field attributes but not complex**) and test smells (**Sensitive Equality**).

3. Can we predict the outcome of future builds? Using the high-impacting features (from RQ2), we were able to predict the outcome of future builds with **precision (recall) between 25% (30%) and 80% (73%) for balanced projects**. **For imbalanced projects, our model achieved a maximum precision (recall) of 20% (15%)**.

The difference in accuracy between balanced and imbalanced projects show that the usability of prediction models for build failures depends on the project for which it is used. We explore the use of cross-project prediction models as possible approach to deal with imbalanced data sets and cold starts. Hence, as future work, we want to further investigate how to handle distinct minority classes especially when we have a small samples of classes. Given that our models did not find any correlation with the build environment with build failure, we aim to perform an extensive study on models trained with different environment-specific and environment-agnostic build failure datasets as proposed by Gallaba *et al.* [57]. We also aim to improve our on-line prediction model by adding other new features. In addition, our model can be extended to dynamically select the best features to use for prediction based on each project's contextual information.

## References

[1] J. Humble, D. Farley, Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation, 1st Edition, Addison-Wesley Professional, 2010.

[2] Y. Yu, H. Dayani-Fard, J. Mylopoulos, P. Andritsos, Reducing build time through precompilations for evolving large software, in: 21st IEEE International Conference on Software Maintenance (ICSM'05), 2005, pp. 59–68. doi:10.1109/ICSM.2005.73.

[3] M. Beller, G. Gousios, A. Zaidman, 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), 2017, pp. 356–367. doi:10.1109/MSR.2017.62.

[4] A. E. Hassan, K. Zhang, Using decision trees to predict the certification result of a build, in: 21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06), 2006, pp. 189–198. doi:10.1109/ASE.2006.72.

[5] M. Zolfagharinia, B. Adams, Y.-G. Guéhéneuc, Do not trust build results at face value: An empirical study of 30 million cpan builds, in: Proceedings of the 14th International Conference on Mining Software Repositories, MSR '17, IEEE Press, Piscataway, NJ, USA, 2017, pp. 312–322. doi:10.1109/MSR.2017.7.

[6] T. Wolf, A. Schroter, D. Damian, T. Nguyen, Predicting build failures using social network analysis on developer communication, in: Proceedings of the 31st International Conference on Software Engineering, ICSE '09, IEEE Computer Society, Washington, DC, USA, 2009, pp. 1–11. doi:10.1109/ICSE.2009.5070503.

[7] I. Kwan, A. Schroter, D. Damian, Does socio-technical congruence have an effect on software build success? a study of coordination in a software project, IEEE Transactions on Software Engineering 37 (3) (2011) 307–324. doi:10.1109/TSE.2011.29.

[8] J. Finlay, R. Pears, A. M. Connor, Data stream mining for predicting software build outcomes using source code metrics, Information and Software Technology 56 (2) (2014) 183 – 198. doi:https://doi.org/10.1016/j.infsof.2013.09.001.

[9] A. E. Hassan, Predicting faults using the complexity of code changes, in: Proceedings of the 31st International Conference on Software Engineering, ICSE '09, IEEE Computer Society, Washington, DC, USA, 2009, pp. 78–88. doi:10.1109/ICSE.2009.5070510.

[10] T. Rausch, W. Hummer, P. Leitner, S. Schulte, 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), 2017, pp. 345–355. doi:10.1109/MSR.2017.54.

[11] F. J. Shull, J. C. Carver, S. Vegas, N. Juristo, The role of replications in empirical software engineering, Empirical software engineering 13 (2) (2008) 211–218.

[12] V. Garousi, B. Küçük, Smells in software test code: A survey of knowledge in industry and academia, Journal of Systems and Software 138 (2018) 52 – 81. doi:https://doi.org/10.1016/j.jss.2017.12.013.

[13] F. Khomh, M. Di Penta, Y. Gueheneuc, An exploratory study of the impact of code smells on software change-proneness, in: 2009 16th Working Conference on Reverse Engineering, 2009, pp. 75–84. doi:10.1109/WCRE.2009.28.

[14] F. Khomh, M. D. Penta, Y.-G. Guéhéneuc, G. Antoniol, An exploratory study of the impact of antipatterns on class change- and fault-proneness, Empirical Software Engineering 17 (3) (2012) 243–275. doi:10.1007/s10664-011-9171-y.

[15] F. Jaafar, Y.-G. Guéhéneuc, S. Hamel, F. Khomh, M. Zulkernine, Evaluating the impact of design pattern and anti-pattern dependencies on changes and faults, Empirical Software Engineering 21 (3) (2016) 896–931. doi:10.1007/s10664-015-9361-0.

[16] G. Bavota, A. Qusef, R. Oliveto, A. Lucia, D. Binkley, Are test smells really harmful? an empirical study, Empirical Softw. Engg. 20 (4) (2015) 1052–1094. doi:10.1007/s10664-014-9313-0.

[17] J. Xia, Y. Li, Could we predict the result of a continuous integration build? an empirical study, in: 2017 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C), 2017, pp. 311–315. doi:10.1109/QRS-C.2017.59.

[18] Why do builds fail? – a conceptual replication study | zenodo, https://doi.org/10.5281/zenodo.4527601, (Accessed on 02/09/2021). doi:10.5281/zenodo.4527601.

[19] C. Zhang, B. Chen, L. Chen, X. Peng, W. Zhao, 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, ESEC/FSE 2019, Association for Computing Machinery, New York, NY, USA, 2019, p. 176–187. doi:10.1145/3338906.3338917.

[20] C. Vassallo, G. Schermann, F. Zampetti, D. Romano, P. Leitner, A. Zaidman, M. D. Penta, S. Panichella, 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), 2017, pp. 183–193. doi:10.1109/ICSME.2017.67.

[21] H. Seo, C. Sadowski, S. Elbaum, E. Aftandilian, R. Bowdidge, Programmers' build errors: A case study (at google), in: Proceedings of the 36th International Conference on Software Engineering, ICSE 2014, ACM, New York, NY, USA, 2014, pp. 724–734. doi:10.1145/2568225.2568255.

[22] X. Jin, F. Servant, A cost-efficient approach to building in continuous integration, in: Proceedings of the 42nd International Conference on Software Engineering, ICSE 2020, IEEE Computer Society, 2020, pp. 13–25.

[23] A. Sabané, M. Di Penta, G. Antoniol, Y. Guéhéneuc, A study on the relation between antipatterns and the cost of class unit testing, in: 2013 17th European Conference on Software Maintenance and Reengineering, 2013, pp. 167–176. doi:10.1109/CSMR.2013.26.

[24] R. Peters, A. Zaidman, Evaluating the lifespan of code smells using software repository mining, in: Proceedings of the 2012 16th European Conference on Software Maintenance and Reengineering, CSMR '12, IEEE Computer Society, Washington, DC, USA, 2012, pp. 411–416. doi:10.1109/CSMR.2012.79.

[25] M. Tufano, F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, D. Poshyvanyk, An empirical investigation into the nature of test smells, in: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Association for Computing Machinery, New York, NY, USA, 2016, p. 4–15. doi:10.1145/2970276.2970340.

[26] S. E. S. Taba, F. Khomh, Y. Zou, A. E. Hassan, M. Nagappan, Predicting bugs using antipatterns, in: Proceedings of the 2013 IEEE International Conference on Software Maintenance, ICSM '13, IEEE Computer Society, Washington, DC, USA, 2013, pp. 270–279. doi:10.1109/ICSM.2013.38.

[27] M. Beller, G. Gousios, A. Zaidman, Travistorrent: Synthesizing travis ci and github for full-stack research on continuous integration, in: 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR), 2017, pp. 447–450. doi:10.1109/MSR.2017.24.

[28] S. Baltes, P. Ralph, Sampling in software engineering research: A critical review and guidelines (2020). arXiv:2002.07764.

[29] Y.-G. GUÉHÉNEUC, sad [ptidej team], http://wiki.ptidej.net/doku.php?id=sad, (Accessed on 02/17/2018) (2007).

[30] F. E. Harrell Jr, Regression modeling strategies: with applications to linear models, logistic and ordinal regression, and survival analysis, Springer, 2015.

[31] F. Hassan, X. Wang, Change-aware build prediction model for stall avoidance in continuous integration, in: 2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), 2017, pp. 157–162. doi:10.1109/ESEM.2017.23.

[32] C. Macho, S. McIntosh, M. Pinzger, Predicting build co-changes with source code change and commit categories, in: 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), Vol. 1, 2016, pp. 541–551. doi:10.1109/SANER.2016.22.

[33] S. Suzuki, H. Aman, S. Amasaki, T. Yokogawa, M. Kawahara, An application of the pagerank algorithm to commit evaluation on git repository, in: 2017 43rd Euromicro Conference on Software Engineering and Advanced Applications (SEAA), 2017, pp. 380–383. doi:10.1109/SEAA.2017.24.

[34] J. Eyolfson, L. Tan, P. Lam, Do time of day and developer experience affect commit bugginess?, in: Proceedings of the 8th Working Conference on Mining Software Repositories, MSR '11, Association for Computing Machinery, New York, NY, USA, 2011, p. 153–162. doi:10.1145/1985441.1985464.

[35] P. Rotella, S. Chulani, Analysis of customer satisfaction survey data, in: 2012 9th IEEE Working Conference on Mining Software Repositories (MSR), 2012, pp. 88–97. doi:10.1109/MSR.2012.6224304.

[36] J. W. Graham, Missing data analysis: Making it work in the real world, Annual review of psychology 60 (2009) 549–576.

[37] G. Canfora, L. Cerulo, M. Di Penta, F. Pacilio, An exploratory study of factors influencing change entropy, in: 2010 IEEE 18th International Conference on Program Comprehension, 2010, pp. 134–143. doi:10.1109/ICPC.2010.32.

[38] A. Chatzigeorgiou, A. Manakos, Investigating the evolution of code smells in object-oriented systems, Innovations in Systems and Software Engineering 10 (1) (2014) 3–18. doi:10.1007/s11334-013-0205-z.

[39] J. Han, M. Kamber, J. Pei, 3 - data preprocessing, in: J. Han, M. Kamber, J. Pei (Eds.), Data Mining (Third Edition), third edition Edition, The Morgan Kaufmann Series in Data Management Systems, Morgan Kaufmann, Boston, 2012, pp. 83 – 124. doi:10.1016/B978-0-12-381479-1.00003-4.

[40] L. Breiman, Random forests, Machine Learning 45 (1) (2001) 5–32. doi:10.1023/A:1010933404324.

[41] R. Díaz-Uriarte, S. Alvarez de Andrés, Gene selection and classification of microarray data using random forest, BMC Bioinformatics 7 (1) (2006) 3. doi:10.1186/1471-2105-7-3.

[42] B. Efron, Estimating the error rate of a prediction rule: Improvement on cross-validation, Journal of the American Statistical Association 78 (382) (1983) 316–331. doi:10.1080/01621459.1983.10477973.

[43] T. G. Dietterich, Approximate statistical tests for comparing supervised classification learning algorithms, Neural Computation 10 (7) (1998) 1895–1923. doi:10.1162/089976698300017197.

[44] A. Dmitrienko, G. G. Koch, Analysis of clinical trials using SAS: A practical guide, SAS Institute, 2017.

[45] Y. Kamei, A. Monden, S. Matsumoto, T. Kakimoto, K. Matsumoto, The effects of over and under sampling on fault-prone module detection, in: First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007), 2007, pp. 196–204. doi:10.1109/ESEM.2007.28.

[46] K. Kotipalli, S. Suthaharan, Modeling of class imbalance using an empirical approach with spambase dataset and random forest classification, in: Proceedings of the 3rd Annual Conference on Research in Information Technology, RIIT '14, Association for Computing Machinery, New York, NY, USA, 2014, p. 75–80. doi:10.1145/2656434.2656442.

[47] G. Lemaundefinedtre, F. Nogueira, C. K. Aridas, Imbalanced-learn: A python toolbox to tackle the curse of imbalanced datasets in machine learning, J. Mach. Learn. Res. 18 (1) (2017) 559–563.

[48] E. Shihab, Y. Kamei, B. Adams, A. E. Hassan, Is lines of code a good measure of effort in effort-aware models?, Inf. Softw. Technol. 55 (11) (2013) 1981–1993. doi:10.1016/j.infsof.2013.06.002.

[49] M. W. Godfrey, A. E. Hassan, J. Herbsleb, G. C. Murphy, M. Robillard, P. Devanbu, A. Mockus, D. E. Perry, D. Notkin, Future of mining software archives: A roundtable, IEEE Software 26 (1) (2009) 67–70. doi:10.1109/MS.2009.10.

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

[51] J. Eyolfson, L. Tan, P. Lam, Do time of day and developer experience affect commit bugginess?, in: Proceedings of the 8th Working Conference on Mining Software Repositories, MSR '11, Association for Computing Machinery, New York, NY, USA, 2011, p. 153–162. doi:10.1145/1985441.1985464.
URL https://doi.org/10.1145/1985441.1985464

[52] E. Doğan, E. Tüzün, K. A. Tecimer, H. A. Güvenir, Investigating the validity of ground truth in code reviewer recommendation studies, in: 2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), IEEE, 2019, pp. 1–6.

[53] Z. Xia, H. Sun, J. Jiang, X. Wang, X. Liu, A hybrid approach to code reviewer recommendation with collaborative filtering, in: 2017 6th International Workshop on Software Mining (SoftwareMining), IEEE, 2017, pp. 24–31.

[54] A. V. Deursen, L. Moonen, A. Bergh, G. Kok, Refactoring test code, in: Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP2001, 2001, pp. 92–95.

[55] D. Spadini, F. Palomba, A. Zaidman, M. Bruntink, A. Bacchelli, On the relation of test smells to software code quality, in: 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, 2018, pp. 1–12.

[56] A. Chatzigeorgiou, A. Manakos, Investigating the evolution of bad smells in object-oriented code, in: 2010 Seventh International Conference on the Quality of Information and Communications Technology, 2010, pp. 106–115. doi:10.1109/QUATIC.2010.16.

[57] K. Gallaba, C. Macho, M. Pinzger, S. McIntosh, Noise and heterogeneity in historical build data: An empirical study of travis ci, in: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Association for Computing Machinery, New York, NY, USA, 2018, p. 87–97. doi:10.1145/3238147.3238171.