Die approbierte Originalversion dieser Diplom-/Masterarbeit ist in der Hauptbibliothek der Technischen Universität Wien aufgestellt und zugänglich.
http://www.ub.tuwien.ac.at

The approved original version of this diploma or master thesis is available at the main library of the Vienna University of Technology.
http://www.ub.tuwien.ac.at/eng

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Build Failure Prediction in Continuous Integration Workflows

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering und Internet Computing

eingereicht von

Thomas Rausch, BSc
Matrikelnummer 0726439

an der Fakultät für Informatik
der Technischen Universität Wien

Betreuung: Ass.Prof. Dr.-Ing. Stefan Schulte
Mitwirkung: Dipl.-Ing. Dr. Waldemar Hummer

Wien, 23. September 2016

Thomas Rausch   Stefan Schulte

FAKULTÄT FÜR !NFORMATIK

Faculty of Informatics

Build Failure Prediction in Continuous Integration Workflows

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering and Internet Computing

by

Thomas Rausch, BSc
Registration Number 0726439

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Ass.Prof. Dr.-Ing. Stefan Schulte
Assistance: Dipl.-Ing. Dr. Waldemar Hummer

Vienna, 23rd September, 2016

Thomas Rausch   Stefan Schulte

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.ac.at

Erklärung zur Verfassung der Arbeit

Thomas Rausch, BSc
Siebenbrunnenfeldgasse 26-30/21/6, A-1050, Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 23. September 2016

Thomas Rausch

Acknowledgements

First, I want to thank my advisors Stefan Schulte (who keeps me on my toes) and Waldemar Hummer. I'm especially grateful for Waldemar's continued support, even after he moved to the other side of planet Earth and started living under the constant threat of being eaten by crocodiles.

I want to thank the Universe, because it's great! It fuels our curiosity by providing us with an endless stream of interesting problems waiting to be solved.

Finally, I wish to express my deepest gratitude to Katharina Krösl, for the many fruitful discussions, for proof reading a large portion of this thesis, for her caring, her tremendous patience, and for supporting me in all my endeavors.

Kurzfassung

Kontinuierliche Integration (englisch „Continuous Integration“ (CI)), ist eine Methode der Agilen Softwareentwicklung, die das kontinuierliche Zusammenführen und Testen von Änderungen an der Software Codebasis vorsieht. Ein CI-Server überwacht fortlaufend das Quellcode-Repositorium und führt bei neu eingegangenen Änderungen automatisch den Software-Erstellungsprozess (englisch „Build“) durch. Schlägt der Build fehl, so muss die Ursache des Fehlers gefunden und behoben werden, was eine Verzögerung im Integrierungsprozess und der weiteren Softwareentwicklung zur Folge hat. In komplexen Softwareprojekten kann der Build-Prozess sehr langwierig sein, wodurch sich das Problem weiter verschärft.

Trotz des weitverbreiteten Einsatzes von CI ist nur wenig über die vielfältigen Ursachen für fehlschlagende Builds bekannt. Jedoch ist ein eingehendes Verständnis darüber wann und wie solche Fehler entstehen ein wichtiger Aspekt um die Produktivität in CI-Arbeitsabläufen zu verbessern. Durch die Identifikation von Entwicklungspraktiken, die häufig zu Ausfällen führen, wird es möglich, Vorhersagen über den Erfolg einer bevorstehenden Integration zu treffen. Eine solche Vorhersage erlaubt es Entwicklern auf mögliche Probleme einzugehen noch bevor der Build-Prozess ausgelöst wird, wodurch Zeit und Ressourcen gespart werden können.

In dieser Arbeit präsentieren wir eine Vorgehensweise für die Analyse des CI-Arbeitsablaufs sowie eine empirische Studie basierend auf Daten von 14 Open-Source-Softwareprojekten, welche CI einsetzen. Daten aus Quellcode-Repositorien und Build-Systemen werden untersucht um qualitative und quantitative Aussagen über die Vielfalt und Häufigkeit von Build-Fehlern zu treffen. Mittels statistischer Verfahren wird die Beziehung zwischen Merkmalen des Entwicklungsprozesses und Build-Fehlern analysiert und bewertet. Basierend auf den Ergebnissen dieser Untersuchungen wird ein Verfahren für die Vorhersage von Build-Fehlern vorgestellt.

Unsere Ergebnisse zeigen, dass fehlschlagende Modultests und das Verletzen von Programmierstil-Richtlinien die häufigsten Ursachen für Build-Fehler sind. Den statistischen Untersuchungen zu Folge sind die Anzahl und Art vorhergehender Fehler die stärksten Prädiktoren für künftige Ausfälle. Unsere besten Vorhersagemodelle für Ausfälle im Build-Prozess erzielen eine Trefferquote von 0.82 und eine Genauigkeit von 0.80. Darüber hinaus erlaubt unser Ansatz das Aktualisieren einer Vorhersage während der Ausführung eines Builds.

Abstract

Continuous integration (CI) is a practice where developers integrate their work into the main stream of development frequently. A CI server continuously monitors the source code repository of a project and automatically executes the software build process when new changes are checked in. If a build fails, developers have to identify and fix the cause of the broken build, leading to a delay in the integration process and stalling further development. Large software projects often have long running builds that exacerbate this problem.

Despite the widespread use of CI, little is known about the multiplicity of errors that cause builds to fail. Yet, understanding when and why build errors occur is an important step towards improving developer productivity in the CI workflow. By identifying characteristics of development practices that cause build failures, we can predict preliminary results for an integration. This helps developers react to possible problems even before a build is initiated, thereby saving time and resources.

In this thesis, we introduce CInsight, a comprehensive framework for analyzing CI workflows and build failures. We conduct an empirical study on real-world data from 14 open source software projects. Data from source code repositories and build systems are explored to gather qualitative and quantitative evidence about the multiplicity and frequency of CI build errors. Statistical methods are used to examine the relationship between development practices and build failures. Based on the results, we devise a method for CI build failure prediction.

Our results show that failing unit-tests and violations of code quality rules are the main causes for build failures. The statistical analyses reveal that the type and amount of previous errors are the strongest predictor for future failures. Our best prediction models yield average recall and precision values of 0.82 and 0.80, respectively. Furthermore, our approach allows to update a prediction during the execution of a build.

Contents

Kurzfassung **ix**

Abstract **xi**

Contents **xiii**

List of Figures **xv**

List of Tables **xvi**

1 Introduction **1**

2 Background **7**
2.1 Version Control 7
2.2 Build Automation 13
2.3 Continuous Integration 14
2.4 Machine Learning 18

3 Related Work **21**
3.1 Mining Software Repositories 21
3.2 Systematic Analysis of Software Build Errors 24
3.3 Software Defect Prediction 25
3.4 Build Failure Prediction 26

4 Solution Approach **31**
4.1 Overview 31
4.2 Methodology 32
4.3 Approach Outline 33
4.4 System Model 36
4.5 CInsight Data Analysis Framework 39

5 Systematic Analysis of Build Errors **41**
5.1 Build Error Categorization 41
5.2 Runtime Behavior of Builds 46

5.3 Discussion 55

6 Factors Influencing Build Results **57**

6.1 Data Processing 57
6.2 Factors to Explore 61
6.3 Change Characterization 63
6.4 Process Characterization 68
6.5 Statistical Analyses 73
6.6 Discussion 86

7 Predicting Build Failures **89**

7.1 Approach 89
7.2 Experiment Design 92
7.3 Results 94
7.4 Discussion 99

8 Conclusion **101**

8.1 Discussion of Research Questions 101
8.2 Future Work 103

A Statistical Data on Build Errors **105**

B Statistical Data on Factors Influencing Build Results **107**

C Statistical Data on Build Failure Prediction **115**

Acronyms **121**

Bibliography **123**

List of Figures

2.1 Changes to the codebase in a linear commit history 8
2.2 A commit history with two diverging branches, and the effect of merging them 10
2.3 A commit history after a rebase 11
2.4 The effect of commit squashing 11
2.5 Centralized and distributed version control systems 12
2.6 The fork & pull workflow 13
2.7 Logical processes of a build script according to [DMG07] 13
2.8 Components of a CI system according to [DMG07] 15
2.9 The basic workflow of a CI server 15
2.10 The Travis-CI build history dashboard of the Spring Boot project 17

4.1 Overview of the data science process according to Schutt and O'Neil [SO13] . 32
4.2 Solution approach overview 34
4.3 Relationship between builds and commits 38

5.1 A flowchart of the LogCat open coding workflow 43
5.2 Distribution of common error categories 46
5.3 Measuring the build execution duration (runtime) when multiple jobs fail . . 48
5.4 Time series scatter plots of builds from four different projects 50
5.5 Time series data of passed builds from Spring Boot 51
5.6 Windowed time series data of passed builds from Spring Boot 51
5.7 Boxplots of the build execution duration of different result categories 52
5.8 Runtime histogram of different error categories from Spring Boot 53
5.9 Runtime density of different result categories 54

6.1 Mapping build data to the VCS history topology 58
6.2 Topology mapping of an updated pull request 59
6.3 Pull request merge squashing 70
6.4 Example of a pull-request-update scenario 70
6.5 Distribution of build types 80
6.6 Previous build results and the percentage of build outcomes 84

7.1 Sampling approach for holdout evaluation 93
7.2 Probability of errors during the build execution 98

List of Tables

2.1 The 2x2 confusion matrix for a binary classification problem 20

3.1 Change activities according to Hindle et al. [HGH08] 23
3.2 Machine learning algorithms employed for build outcome prediction 29

4.1 Name and description of projects used as research subjects 33

5.1 Summary of error categories and their occurrence frequency across projects . 45

6.1 File type categories and associated glob patterns 65
6.2 Truth table for determining the pull request scenario 72
6.3 Results of the Mann–Whitney test of change complexity metrics 75
6.4 Contingency table of file type changes and build results for Spring Boot . . . 76
6.5 Results of the χ^2 test on file types 76
6.6 Results of the χ^2 test on time of day, and weekday measures 77
6.7 Results of the Mann–Whitney test on developer experience 78
6.8 Results of the χ^2 test on commit frequency 79
6.9 Results of the χ^2 test on build types 81
6.10 Failure ratio of different build types 81
6.11 Results of the χ^2 test on pull request scenarios and their predicates 82
6.12 Contingency tables for the previous-build test from the Spring Boot project . 83
6.13 Mean build climate ($k = 10$) values of projects 85
6.14 Results of the χ^2 test on days since last failure 86

7.1 Summary of binary result classification for different projects 95
7.2 Summary of multi-class prediction performance for the Spring Boot project . 96
7.3 Number of introduced false positives compared to confidence levels 98

A.1 Summary of builds by project and build state category 105
A.2 Summary of builds by project and build error category 106

B.1 Result of filtering observations that have linked change data 107
B.2 Summary of pull request scenarios per project 108
B.3 Summary of previous build results per project 108
B.5 Contingency tables of common file type changes 108

B.4 Mean and max values of change complexity metrics per project (before filtering) 111
B.6 Mean values of change complexity metrics per build outcome and project (after filtering) 112
B.7 Summary of builds per project and author commit frequency 112
B.8 Summary of failure ratio per project and author commit frequency 113
B.9 Summary of builds in days-since-last-failure intervals 113

C.1 Weighted average F_1-score results of binary build outcome prediction 115
C.2 κ-statistic results of multi-class build outcome prediction 116
C.3 Root-mean-squared error (RMSE) results of multi-class build outcome prediction 117

CHAPTER 1

Introduction

Motivation

The culture in which software is developed has changed dramatically over the past decade [DSTH12]. A completely new ecosystem of tools and development practices has evolved around social coding platforms [BCSD14, VYW+15] This ecosystem has allowed geographically dispersed teams to collaborate effectively, and increased the amount of contributions to open source software (OSS) projects [BRB+09]. Yet, many of these projects are subject to high delivery pressure [WP12]. As the frequency in which software is released increases, so does the need for early detection of software integration issues. It is unacceptable if the release process is stalled because of compilation errors or failing unit tests.

The effect of every change by developers to the source code should be checked to make sure the system remains in a functioning state. To that end, many modern software projects use version control systems (VCS) to manage their source code, and have adopted continuous integration (CI), a practice where team members integrate their work into the main stream of development frequently [DMG07]. Automating the building process, i.e., the compilation, linking of dependencies, packaging, etc. of software artifacts, and the execution of automated tests, are vital for an effective CI workflow. A dedicated infrastructure with a sophisticated build system is often employed to fully automate building and even deploying software after changes have been integrated. Such a CI server monitors the VCS repository, fetches new changes, executes the build suite, and notifies developers about the result of the build [DMG07].

Despite the widespread adoption of CI [GZSD15], little is understood about the multiplicity of errors that may occur during a build, and factors that lead to build failures. Yet, during development, a large amount of time and focus goes into finding such errors, and then fixing broken builds to allow the continued development on top of successfully built

and tested changes [KKA14]. Furthermore, our data show that large software projects often have long build chains that may run for several minutes, or even hours, before returning feedback on the outcome of a build. Long builds may not only be costly in terms of computational resources on the CI infrastructure, but also inhibit one of the key purposes of CI: to produce rapid feedback on the effects of an integration to the system [DMG07]. Waiting a long time on such feedback can have a negative impact on productivity when developers are required to build on top of changes that triggered a build [KKA14].

Substantial research exists on how data science methods (such as computational statistics or machine learning) can be used to predict future *software defects* from different quality measures [GKMS00, MPS08, ABJ10, DLR12, RHT+13, MJ15]. Only few efforts have been made to develop similar methods for predicting build failures, or to study factors that influence the outcome of builds. Studies that analyze build errors either focus on builds executed in a developer's private workspace [SSE+14], or do not consider the multiplicity of possible errors that can occur during a CI build [CH11, KKA14]. Existing build failure prediction approaches are either based on outdated assumptions (e.g., that the integration process may take several days) [HZ06], or focus on the socio-technical aspects of development [WSDN09, Sch10, KSD11].

Problem Statement

Build failures have a negative impact on the development process [KKA14]. When a build fails, developers need to manually identify the cause for the build failure, often by reading through the log output of the build automation system. This has a direct negative effect on developer productivity, as time is spent on fixing a broken build, rather than the task at hand. Despite the wide spread use of CI, we lack qualitative and quantitative evidence of why CI builds fail.

Similarly, only a limited amount of studies exists that explore factors that lead to CI build failures. Previous research on software quality has identified a variety of different measures that cause an increase in software defects. Such measures include code complexity, frequency of changes, etc. [DLR12, MJ15] It is unclear whether CI builds are affected by the same factors. Bad work practices that lead to an increase in build failures should be identified. This allows developers to improve existing workflows and reduce the amount of build failures.

CI builds should be fast in order to provide rapid feedback on the state of an integration [DMG07]. However, in many projects, long build chains are unavoidable, e.g., due to the size of the codebase, or the amount of tests in the test suite. Build failure prediction could complement the CI feedback mechanism. Being able to predict preliminary results for an integration would allow developers to react faster to possible defects before even starting a build, or, conversely, fetch changes that are safe to build upon before the build suite has finished.

An aspect of build failures previously unexplored is the influence of different error types as well as the frequency of occurrence of errors during the execution of a build. A CI build comprises different steps, and each subsequent step may produce their own specific kinds of errors. We hypothesize that errors of the same type will cause the build to fail at similar points in time during different phases of the build execution. Such accumulations points allow us to reason about the likelihood of specific errors during the build execution. It is clear that, as the build progresses, the likelihood for errors to occur continues to decline. So does the plausibility of any prediction made by a classification model. By leveraging this understanding of the temporal aspect of build errors it becomes possible to update an initial prediction during the build execution.

Research Questions

Based on the research gaps we have identified, the aim of this thesis is to answer the following four research questions:

RQ1. Why do CI builds fail?

The first research questions aims to provide qualitative and quantitative evidence about the causes of build failures. We report on the different error types and their distribution among software projects. These findings highlight which errors occur most often, and can help to prioritize efforts to reduce build failures. We also report on the frequency of occurrence of errors during the build execution. The findings on the temporal aspect of build errors motivate RQ4.

RQ2. What factors can be associated with CI build failures?

The second research question aims to provide evidence about the relationship between development practices and build failures. We report on the statistical significance of measurable properties of build data on the build outcome. The findings help us to determine which properties can be used for training statistical classification models, explored in RQ3.

RQ3. How well can statistical models predict the outcome of a CI build?

The third research question aims to provide evidence whether CI build failure prediction is feasible and how well different approaches perform. We report on the performance of classification models previously used for build failure prediction. The findings allow practitioners to effectively create prediction models from their project data.

RQ4. Can the temporal aspect of build errors be used for prediction?

The fourth research question aims to examine a previously unexplored aspect of build errors. We report on how the temporal aspect of build errors, i.e., their frequency of occurrence during build execution, can be used for build failure prediction.

Methodology

We conduct our research using real-world data gathered from 14 OSS projects that employ CI. To make the study reproducible, the data are gathered from publicly available data sources. We develop a data analysis toolkit that allows us to extract, link, and structure the heterogeneous data.

We examine the data in a mixed-method study using methods from both qualitative [CS14] and quantitative [SO13] research. The study comprises two parts that aim to answer RQ1 and RQ2, respectively. The goal is a) to gain new insights into the multiplicity of errors that can occur during the execution of a CI build; and b) to provide empirical evidence on factors that influence the outcome of a build. For part one, we develop a systematic process to determine build errors from the build data. First, we quantitatively explore the multiplicity of build errors across projects. We then examine the temporal aspect of build errors, i.e., their frequency of occurrence during the build execution. In part two of the study, we elicit and explore measurable properties of build data. Based on existing research, we define change and process metrics that are computable from the gathered data. We employ methods from the field of mining software repositories (MSR) and perform correlation analyses to examine the strength of the relation between these metrics and the build outcome.

Using the results of the study, we devise a system for predicting build failures. Methods from data mining and machine learning are used to create and evaluate statistical classification models. We conduct several experiments to determine the performance of our classification approach, and provide an answer to RQ3. We proceed to incorporate our understanding of the temporal dimension of build errors into our prediction approach. Using methods from probability theory, we reason about the build outcome during the build execution. We present these results to answer RQ4.

Structure

The remainder of the thesis is structured as follows. Chapter 2 describes the background and theoretical foundations of this work. We first give an overview of development practices in the context of CI, and briefly introduce machine learning methods used in our approach. Chapter 3 presents related work and state-of-the-art methods of empirical software engineering and software build failure prediction. We summarize previous studies on factors that influence software quality, and existing systems for build failure prediction. Chapter 4 provides a high-level overview of our solution approach and the employed methodology. We also define our system model, formalize domain concepts, and briefly describe the data analysis toolkit we developed for the various data science tasks. Chapters 5 and 6 present the methods and results of the two-part study. The first part is a systematic study of CI build errors, and is covered by Chapter 5. The second part is a statistical analysis of factors that influence build outcomes, and is covered by Chapter 6. The results of the two-part study are used as input for Chapter 7, in which

we present our approach to predict build failures. We describe our experiment design and discuss the performance of the prediction models. Finally, Chapter 8 concludes the thesis and provides an outlook on future research opportunities.

CHAPTER 2

Background

This chapter presents preliminary concepts and theoretical foundations that serve as basis for this thesis. The evolution of software development processes has spawned a variety of different tools and methods. We provide an overview of practices that have lead to the concept of continuous integration (CI), and how they are embedded in the CI workflow. Section 2.1 introduces the concept of version control, and elaborates on how modern software projects utilize version control system (VCS). Section 2.2 explains software build automation methods and terminology. Section 2.3 discusses how methods of version control and build automation coalesce in a modern CI workflow. Finally, Section 2.4 gives a brief introduction of machine learning and related methods relevant to this thesis.

2.1 Version Control

Software configuration management (SCM) is a set of practices that define how an organization builds and releases products, and identifies and tracks changes [BA02]. A key concept of SCM is version control (or revision control) of a software system's source code files (codebase) using a VCS. Such as system tracks changes made to the codebase, and maintains a history of changes in a version database. This history also contains metadata about changes, such as the author, date and time, etc. A VCS allows users to recall earlier version of files or a set of files, and view details about specific changes. This section introduces various concepts of VCS, and development practices when employing version control.

2.1.1 Basic Concepts and Terminology

We briefly introduce basic VCS concepts and terminology that we will use throughout this thesis. Terminology differs between VCS. Some terms (e.g., *commit* or *checkout*), may have different meanings depending on the type of VCS (see Section 2.1.5). Because

our approach uses a distributed version control system (DVCS), we will focus on concepts of this type of VCS. Specific terminology is drawn primarily from Git[1], a popular DVCS [CS15].

The codebase of a software project is the set of all source code and configuration files used to build the software system. Modifications made to the codebase are tracked by the VCS that maintains a history of these changes (or diffs, or detals) in a *repository*. The repository is a data structure that stores both the content of the change, as well as metadata for the file tree and the changes themselves (e.g. the author or the date and time of the change).

A commit is the act of recording changes to the repository, thereby creating a new *revision*. Commits occur sequentially over time and are organized as a tree or directed acyclic graph (DAG). The VCS provides means to specify and address revisions by giving a commit a unique id or revision number.

Example 2.1.1. Figure 2.1 illustrates the evolution of a codebase consisting of a single Java file `Main.java` that was changed three times. Circles indicate commits with their respective commit id. An arrow between commits indicates a predecessor relationship, meaning they are subsequent commits. In the given example a is a parent of b is a parent of c which is the latest commit.

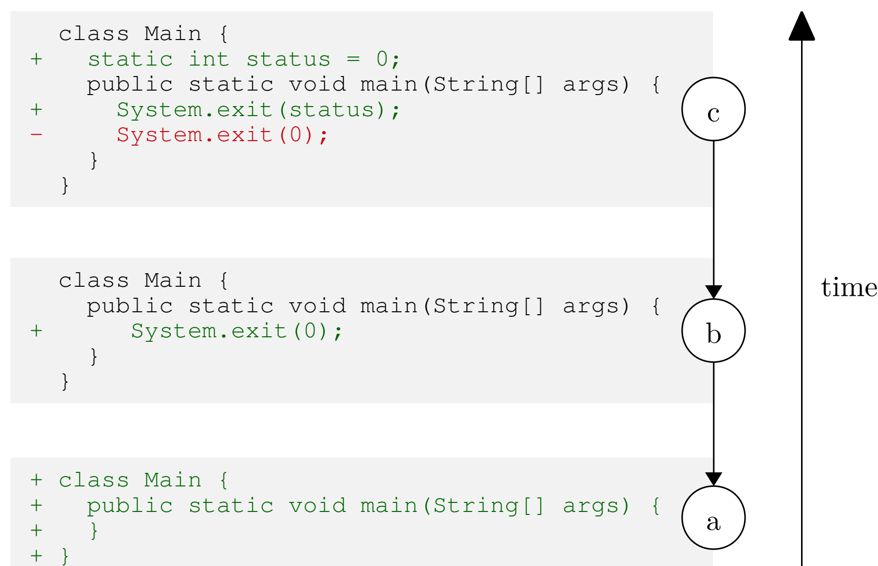


Figure 2.1: Changes to the codebase in a linear commit history

[1] https://git-scm.com/

The repository also stores metadata about commits. Some examples are the author of the change, the committer who recorded the change[2] , date and time, or a written description of the change (commit message).

The local working directory of a developer contains the version of the codebase they are currently working with. The working directory serves as a writable view to the repository. Developers can *checkout* specific revision from the repository, making their working directory contain the state of the codebase at that given point in the timeline.

2.1.2 Branching

Conceptually, a branch is a path through the development timeline, managed by the VCS. Berczuk and Appleton [BA02] call it a *codeline* and define it as follows:

> *A codeline is a progression of the set of source files and other artifacts that make up some software component as it changes over time. [...] A codeline contains every version of every artifact along one evolutionary path.*

In Git, a branch is simply a named symbolic reference (or *ref*) to a commit.

Typically, the development of logically cohesive units, such as features or individual bug fixes, will be isolated from the main stream of development (master branch) by creating a new development line (topic branch). Changes to the branch can then be made in parallel to the main development timeline, without having to duplicate the entire codebase.

2.1.3 Merging

Once changes have been concluded, the branch is merged into its *baseline*, and any conflicts are manually or automatically resolved. The baseline of a branch is the timeline that was branched off of. The merge base of a branch and its baseline is the first ancestor commit both branches have common. In terms of graph theory, the merge base of two commits is their lowest common ancestor (LCA).

Example 2.1.2. Figure 2.2 shows a commit history with two diverging branches `master` and `topic`, and the effects of merging `topic` into `master` (right). The merge base of e and d is b. Commit f is a merge commit that has two ancestors.

A branch may be merged into a baseline that has not advanced, i.e., the merge base is the latest commit of the baseline. In such a case, it is not necessary to create a merge commit. This type of merge scenario is known as a *fast-forward* merge.

[2] In most cases the author and the committer are the same person. In some workflows, changes are applied by someone other than the author. By storing these metadata separately, credit can be accurately attributed.

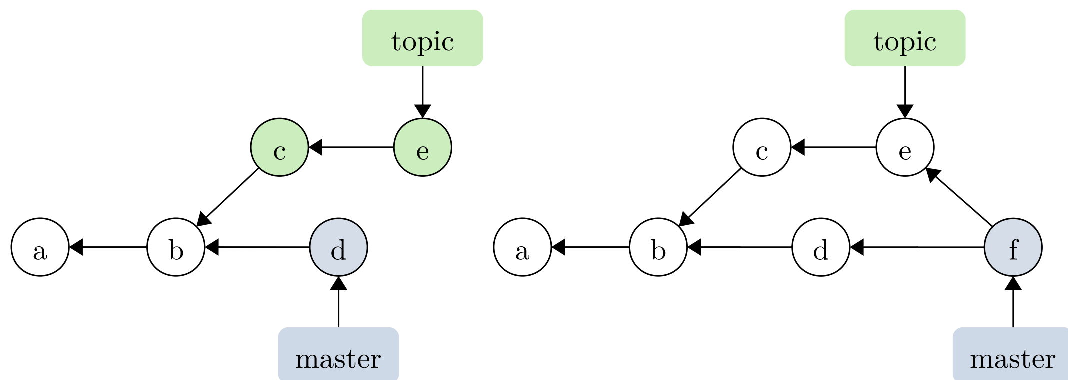


Figure 2.2: A commit history with two diverging branches, and the effect of merging them

2.1.4 History Rewriting

Maintaining a readable commit history is important to allow readers to understand how features or bugfixes were developed, and who was involved. Commits should isolate changes that belong together to reduce possible conflicts and allow granular reverting of changes. There are several situations when it can be beneficial to manipulate the commit history to make sure the history remains readable. Git provides different ways of rewriting a commit history, e.g., reordering of commits, amending changes to commits or merging commits together.

Rebasing Rebasing is an alternative way of integrating changes of a branch. By rebasing one branch on top of another, the merge base of the two branches is changed to the tip of the branch being rebased onto. By changing the ancestors of commits, they receive new commit ids, and are effectively different commits.

Example 2.1.3. Figure 2.3 shows the history of Example 2.1.2 after rebasing *topic* onto *master*. Commits c' and e' are new versions of c and e respectively.

Amending Amend applies the current uncommitted changes to a previous commit. This is useful when you have committed changes to the repository, and later realize that you have changes the previous commit should include.

Squashing Squashing a set of subsequent commits combines their changes into a single commit. Squashing can be used in merge scenarios to merge the changes of a branch, but not its commit history.

Example 2.1.4. Figure 2.4 shows the state of the tree from Example 2.1.1 after the commits b and c were squashed into a single commit d.

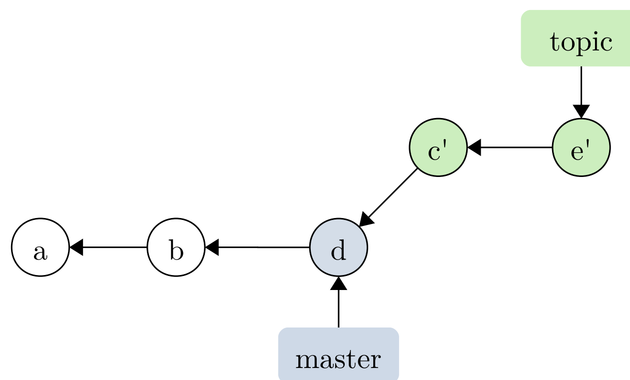


Figure 2.3: The commit history from Example 2.1.2 after a rebase

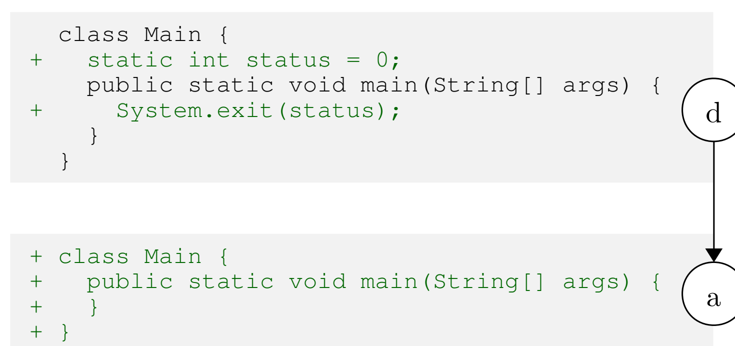


Figure 2.4: The change history from Example 2.1.1 with squashed commits

History rewriting has consequences for mining software repositories (MSR) techniques, because not all data of rewritten commits are retained. Specifically, process data gets lost because the structure and order of the history may be changed completely [BRB+09].

2.1.5 Centralized vs. Distributed Version Control Systems

Two main types of VCS are distinguished: centralized version control systems (CVCS) and DVCS. In a CVCS, such as Subversion[3], the history of changes are kept solely on a centralized server, and every command executed on the history (such as listing all changes), requires a connection to the server. Conversely, in a DVCS, such as Git, developers have a copy of the history in their local workspace, and regularly synchronize with a remote repository to publish local and fetch remote changes. Figure 2.5 illustrates the difference between CVCS and DVCS.

In the remainder of this thesis, we focus entirely on concepts and workflows of DVCS.

[3] http://subversion.apache.org/

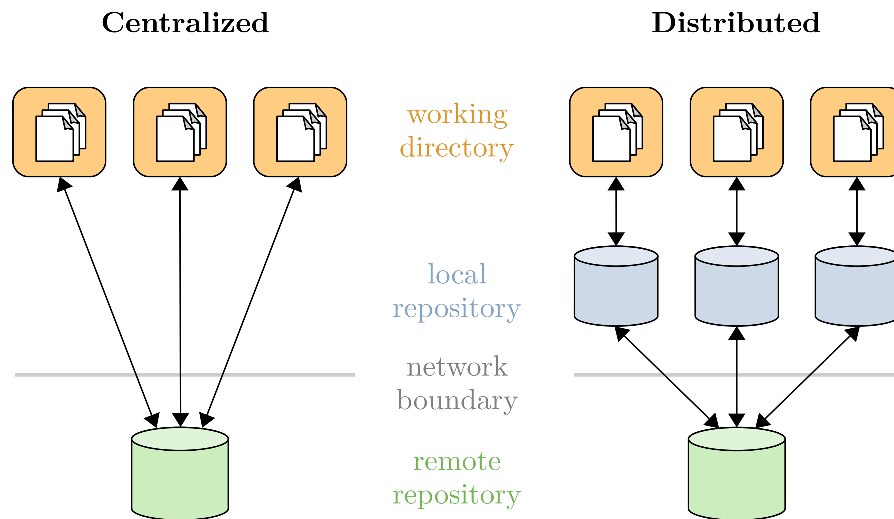


Figure 2.5: Centralized and distributed version control systems

2.1.6 Distributed Workflows

DVCS allow for many different types of distributed workflows. Public repository hosting services such as GitHub[4] or Bitbucket[5], provide an additional layer of tools and processes on top of those provided by DVCS. The *fork & pull* workflow, based on Git's *Integration-Manager Workflow* [CS15], is made possible by these services and has become a common practice [GZSD15].

A fork is a personal clone of a repository, which allows developers to make changes and publish code contributions, even if they do not have direct write access to the forked repository (upstream). To contribute to the project, developers publish changes into their own fork (which can be seen as a branch), and once the changes should be integrated into the project, a *pull request* is opened. A pull request makes the contribution public and opens it up for debate. This allows developers to review the changes and suggest improvements before they are integrated. The author can now iteratively update the pull request to add the necessary improvements. Pull requests are subsequently merged or declined by integration managers (integrators) or privileged developers. Figure 2.6 illustrates this model.

As we have established earlier, Git provides different ways of integrating changes. GitHub allows integrators to chose the way they merge pull requests. The GitHub web application provides two basic ways: standard merging, or squashing commits of a pull reuquest into one and then merging[6]. The integrator can also chose to manually merge the pull request in his local repository by using any method that Git provides to integrate changes.

[4] `https://github.com/`
[5] `https://bitbucket.org/`
[6] `https://help.github.com/articles/about-pull-request-merge-squashing/` (accessed 2016-08-20)

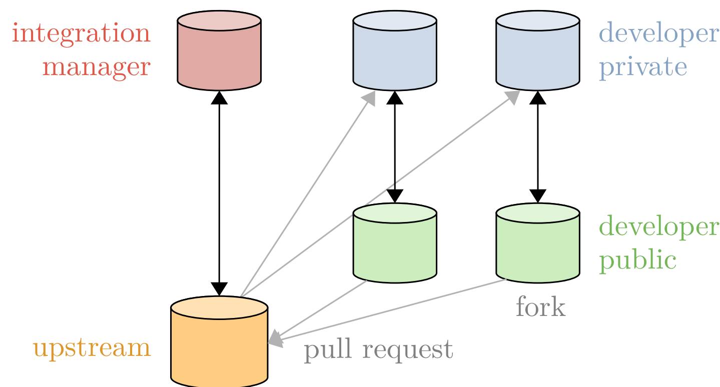


Figure 2.6: The fork & pull workflow

2.2 Build Automation

Making distributable software binaries from source code is an integral part of software engineering. Automating the software build process is vital for efficient development. Build automation utilities like Make[7], Ant[8] or Maven[9] have been around for many years. Their purpose is to make the build process configurable (via *build scripts*) and automate it to minimize or eliminate necessity for user interaction. The build script (or build configuration) contains directives for the various build tasks, and may specify software dependencies required to build the software. Figure 2.7 shows tasks typically performed by such a build script.

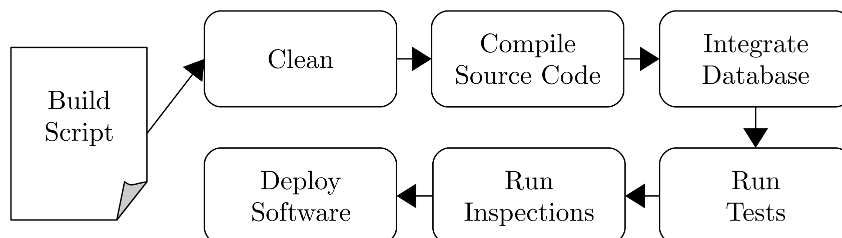


Figure 2.7: Logical processes of a build script according to [DMG07]

2.2.1 Build Types

Compiling and running unit tests after making changes to the source code is a common task during the development process, and is known as the *edit-compile-test* cycle [SSE^{+}14]. A software build executed during an edit-compile-test iteration has to be fast in order to allow the developer to maintain focus on the task at hand. In literature, these are referred to as private builds [BA02, DMG07], and will typically only compile the source

[7] https://www.gnu.org/software/make/
[8] http://ant.apache.org/
[9] http://maven.apache.org/

and run unit tests. Many integrated development environment (IDE) facilitate the edit-compile-test cycle with private builds by natively integrating build tools.

When parallel changes in different modules of a software are integrated into the baseline, regressive effects of the integration may not be detected by a private build of individual developers. For example, unit tests can, as such, not assert the correctness of complex interaction behaviors of components. An integration build (or full software build) runs additional build tasks that aim to avoid both functional and qualitative regression. Such tasks can include: running integration tests, code inspections (e.g., quality metrics), deploying the system, etc. These builds typically take significantly longer to complete than private builds, and are therefore impractical for individual development tasks.

2.2.2 Build Server

Effectively facilitating integration builds requires a centralized system with dedicated hardware [DMG07]. A build server is the reference environment for integrating parallel changes, and building a shippable software product from the codebase. It eliminates build errors that originate from any possible mismatch of individual developer environments, such as differing codebase revisions or software dependencies. Providing dedicated hardware that runs integration builds efficiently reduces build time which will allow the build process to run more often.

2.3 Continuous Integration

CI is a practice where team members integrate their work into the main stream of development frequently [FF06]. The main goal of CI is to reduce the amount of problems resulting from integration tasks. The central idea of applied CI is a (figurative) *integrate* button [DMG07], similar to a *build* button that many IDEs have. Developers push the *integrate* button when they have changes they wish to integrate into the software system. A process is then triggered that performs this integration in a fully automated and safe way, and then provides feedback to the developer whether or not the integration was successful.

Using a VCS is elementary to CI, because it centralizes software assets and provides a central point of integration. Typically, a dedicated infrastructure is used that continuously monitors the source code repository for changes, and runs integration builds against the baseline. Once changes are committed to the repository, an integration build may check whether the changes build correctly when merged into the baseline. Figure 2.8 illustrates how the different components of a CI system work together.

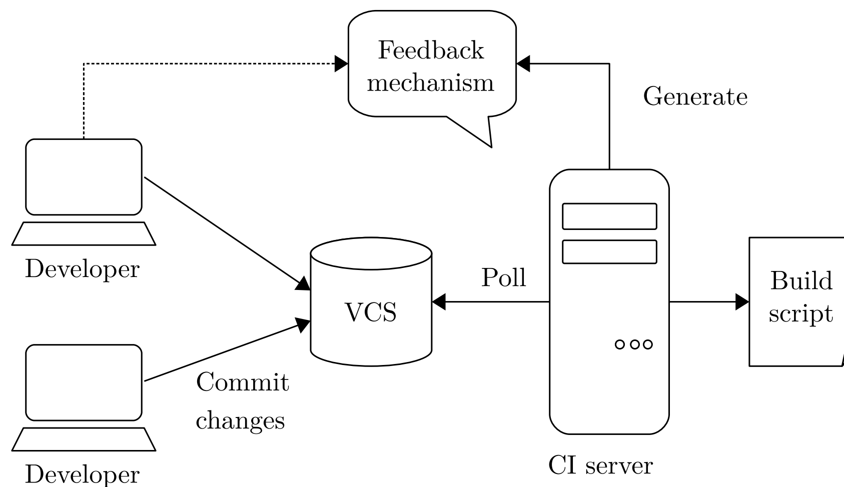


Figure 2.8: Components of a CI system according to [DMG07]

2.3.1 CI Server

Build centralization is essential for implementing CI. Build servers[10] are used together with VCS to fully automate and centralize the integration process.

At specific triggers (e.g., manual, polling or push notifications), the CI server will fetch changes from the VCS, simulate an integration, execute the build process, and publish the build outcome through an appropriate communication channel (see Section 2.3.3). Figure 2.9 depicts the basic CI server system workflow.

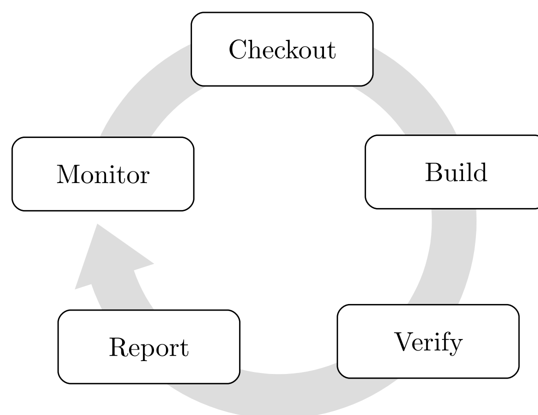


Figure 2.9: The basic workflow of a CI server

As stated earlier, CI builds may also execute and verify integration tests, and inspections such as checking code quality metrics. Software builds that cover all these aspects

[10] Although build servers have existed before CI, the terms CI server and build server are now often used synonymously.

may take significantly longer to complete. Another benefit of using a CI server is that, providing dedicated hardware allows these builds to be executed faster and more often.

Build Triggering

Builds on a CI server can be triggered in different ways, and systems usually provide configuration possibilities. A person can trigger a build manually, or some automatisms can be employed. Typically, the CI server continuously monitors the VCS for incoming changes to trigger the builds. Some VCS also provide *commit-hooks* that can be used to trigger build executions after a change has been pushed to the repository.

Systems

Many different CI server systems exist. Tools like Hudson[11] or Jenkins[12] can be set up on-premise. Hosted services such as Travis-CI have made CI available even for small open-source projects.

2.3.2 Travis-CI

Travis-CI[13] is a hosted, open-source, distributed build system[14], which has gained enormous popularity in the open source software (OSS) community [VYW+15]. Integrated deeply with the GitHub repository hosting services, Travis-CI monitors changes made to repositories, and executes the configured build suite against the baseline of the repository. We briefly present different aspects of Travis-CI.

Build Database

Travis-CI records all meta-data and logs for every executed build, and provides both UI and API based access to this history. Figure 2.10 shows Travis-CI's web based build history dashboard. Each row represents a build. Colors indicate the state of the build (green means *passed*, red means *errored* or *failed*). The middle column features the incremental build number within the project, and the commit SHA that was built.

Meta-data of builds include properties such as build number, build state, date and time, runtime duration, and VCS commit information[15].

[11] `http://hudson-ci.org`
[12] `https://jenkins.io`
[13] `https://travis-ci.org`
[14] `https://github.com/travis-ci/travis-ci/blob/2ea7620f4be51a345632e355260b22511198ea64/README.textile` (accessed 2016-08-09)
[15] `https://docs.travis-ci.com/api#builds` (accessed 2016-08-09)

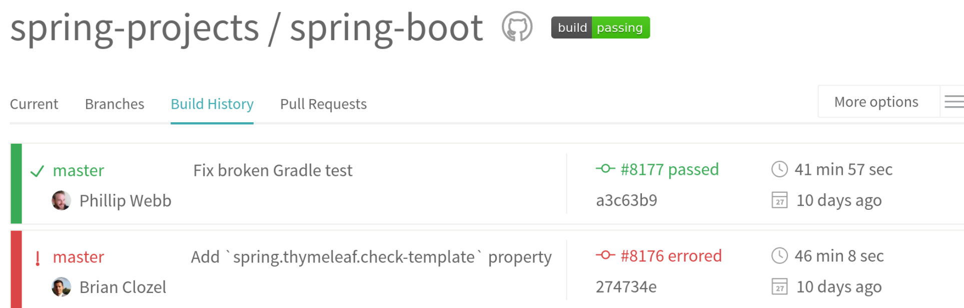


Figure 2.10: The Travis-CI build history dashboard of the Spring Boot project

REST API

Travis-CI provides a REST API[16] to access most of the recorded build meta-data, as well as logs captured during the build execution. We later exploit this API for our data acquisition tasks (See Section 4.3.1)

Pull Request

Travis-CI supports GitHub's pull-based workflow, and distinguishes between changes pushed directly into a branch, and those published via a pull request. When a pull request on GitHub is updated, a hidden merge commit is created that simulates a merge into the branch being pulled into. This merge commit is then checked out by the Travis-CI worker which subsequently executes the build.

Build Life Cycle and Status

The status of a Travis-CI build can be *started*, *canceled*, *errored*, *failed*, or *passed*. Running builds have the status *started*, and can be canceled by users manually, giving the build the *canceled* status. Travis-CI workers split the build execution into different phases (in addition to the phases executed by the build script itself)[17]. If a build does not pass, depending on which phase it terminates in, it gets the status *errored* or *failed*. Lastly, if a build successfully exist, it gets the status *passed*.

2.3.3 Feedback

A key purpose of CI is to identify and fix potential problems at an early stage. To facilitate this, developers have to be provided with feedback on the system-wide impact of their integration as soon as possible. Until it is clear whether changes will build successfully, development on top of these changes is risky. The longer it takes to produce feedback, the longer it will take to continue normal development activities [DMG07].

[16] https://docs.travis-ci.com/api
[17] https://docs.travis-ci.com/user/customizing-the-build/

After a build is completed, its *status* is reported through some communication channel (e.g., a dashboard, email or instant-messenger notifications). What type of information is included in a build status report differs greatly between systems. The build status itself is commonly binary: passed (successful) or failed (broken). Sometimes the status *unstable* is also used to indicated builds where some secondary quality measures have not been satisfied.

During the build, the server captures the log output of the build system, and stores it for later usage. In case of a failure, the build log may include markers for warning or error messages emitted by the build system, which are aggregated or highlighted when the report is presented to a human. The developer then has to read through the messages and try to determine why the build has failed.

Improving feedback mechanisms, e.g., by providing a preliminary risk assessment based, can help to decrease the waiting time for developers that depend on specific changes that have not yet been built.

2.4 Machine Learning

Machine learning is a field of study closely related to computer science and computational statistics. It is concerned with methods and algorithms for constructing software that can perform data predictions (e.g., classification or clustering), without being explicitly programmed. Based on exemplary observational data (training set), such algorithms create statistical models that describe this data.

The remainder of this section builds mainly upon definitions and terminology of Alpaydin's *Introduction to Machine Learning* [Alp14], in which he motivates machine learning as follows:

> *Machine learning is programming computers to optimize performance criterion using example data or past experience. We need learning in cases where we cannot directly write a computer program to solve a given problem, but need example data or experience.*

Typical machine learning use cases include speech or handwriting recognition, email spam detection, determining the topic of a document, regression problems, etc.

2.4.1 Machine Learning Approaches

Three broad categories of machine learning tasks are distinguished:

- **Supervised learning**: an algorithm constructs a model based on correctly labeled training data supplied by a *teacher* or *supervisor* (e.g., classification or regression)

- **Unsupervised learning**: an algorithm learns autonomously directly from unlabeled data, and the aim is to recognize patterns in the input (e.g., clustering)
- **Reinforcement learning**: an algorithms learns in a dynamic environment and operates on a *sequence* of observations rather than on individual data (e.g., game playing)

We focus on methods relevant for this thesis; i.e., techniques of supervised learning such as classification using decision trees or neural networks. Build failure prediction is primarily a supervised learning task, because algorithms learn from historic build data that are labeled with the individual build's outcome. Section 3.4.3 gives an overview of the different methods used in existing research.

2.4.2 Classification

Classification is the task of determining the category to which a specific data belongs, based on measurable properties of the data; i.e., explanatory variables. A training algorithm builds a *classifier* from existing observations that have already been assigned its correct category (training set), either manually or through some other means. Because some algorithms are, by their nature, limited to separating the data into two categories, classification problems are divided in *binary* and *multiclass* classification problems.

In binary classification, data are separated into two categories. A prominent example is email spam detection: assign a given email the category *spam* or *non-spam* automatically based on its content. Algorithms that perform binary classification include logistic regression ir support vector machines[18].

Multiclass classification is the problem of separating data into more than two classes. Decision trees and neural networks have been effectively used for these types of problems.

2.4.3 Classifier Model Validation

There are several methods to validate and calculate measures to compare machine learning models. For classification problems, a popular method to test an algorithm is the K-fold cross-validation method [FHT01]. To compare the performance of models, a variety of accuracy measures exist.

K-Fold Cross-Validation

In K-Fold cross-validation, the dataset is divided into K equal parts. $K-1$ parts are used to train the model, and the remaining part is then used as validation set. This is repeated K times, each time leaving out a different one of the K parts. K is typically 10,

[18] It should be noted that extensions of these algorithms exist that also allow multiclass classification. Giving a detailed explanation of these machine learning algorithms would go beyond the scope of this work.

as it has been established that this number generally provides a good balance between training and validation set size.

Evaluation Metrics

When classifying data, there are four possible classification result cases, shown in Table 2.1. Testing a binary classifier means counting the number of true positive, false positive, true negatives, and false negatives. The values are arranged in a grid (the confusion matrix).

Table 2.1: The 2x2 confusion matrix for a binary classification problem

	p' (Predicted)	n' (Predicted)
p (Actual)	tp (True Positive)	fn (False Negative)
n (Actual)	fp (False Positive)	tn (True Negative)

From the numbers in the confusion matrix we can derive the following measures: precision: $\frac{tp}{p'}$ and recall: $\frac{tp}{p}$. The F-measure (or F_1-score) is creates a distinct measure to describe the classifiers performance, and is calculated using precision and recall. Specifically, it is the harmonic mean of precision and recall, i.e., $F_1 = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$

CHAPTER 3

Related Work

This chapter presents related work relevant in the context of this thesis, and state-of-the art methods in software build failure prediction. Section 3.1 presents state-of-the-art methods in the area of mining software repositories (MSR) which are commonly employed in empirical software engineering research. Section 3.2 summarizes studies that quantitatively and qualitatively analyze software build errors. Section 3.3 outlines methods in the adjacent area of software *defect* prediction. Section 3.4 finally presents state-of-the-art work in the area of software *build failure* prediction.

3.1 Mining Software Repositories

MSR is a field of research dedicated to the extraction and analysis of data available in software repositories. Software repositories, such as version control system (VCS) repositories, mailing list archives, or issue tracking systems, provide a rich source of data for research tasks. Such tasks include studies in the area of software evolution [KCM07], or failure prediction [NBZ06]. We employ MSR techniques for mining from the combined data of VCS repositories and continuous integration (CI) build platforms. We begin by discussing methods of mining VCS repositories, in particular from distributed version control system (DVCS).

3.1.1 Mining Distributed Version Control Systems

Having access to version history data of software artifacts is elementary for studying software evolution [KCM07]. In this context, many techniques and tools exist to extract and analyze data from VCS repositories.

In recent years, DVCS (see Section 2.1.5) have become increasingly popular and widely used [BRB+09]. A great benefit of mining DVCS is having the entire repository and its history locally. This makes most operations much faster compared to centralized version

control systems (CVCS), which increases the flexibility of mining methods. However, decentralizing software versioning also significantly impacts development processes and workflows [VYW+15]. How this affects change and process metrics (see Section 3.1.2) is largely unknown [BCSD14].

Mining from a DVCS such as Git brings its own set of challenges. The methods Git provides, e.g., to rewrite commit histories (see Section 2.1.4), have significant impact on the way we need to interpret data mined from repositories. Bird et al. [BRB+09] identified several aspects that require special consideration when mining historical data from Git, e.g., that one cannot always determine what branch a commit was made on, or that the accessible data may not include all commits that were made by developers. We later discuss how we addressed such issues in our approach.

3.1.2 Process Metrics

Process metrics are calculated from the software change history [RHT+13], typically by mining historical data from a VCS. In contrast to code metrics which quantify properties of software artifacts (e.g., size and complexity), process metrics aim to quantify properties of software evolution. These metrics are popular for prediction models in software defect prediction (see Section 3.3) because the cost of mining them is lower, and the performance has been found to be equal or better compared to code metrics [RD13, MJ15].

Process metrics can be calculated for sets of commits, individual files, or authors. Examples include the number of modified lines in a commit, number of revisions of a file, or the experience of an author (e.g., total number of contributions). Many different process metrics have been developed and studied for a variety of different research tasks [HZ06, Has08, MPS08, ABJ10, NZZ+10, RD13, MJ15]. A summary of these studies (many of which are very comprehensive), and process metrics would go beyond the scope of this work.

In our approach, we use common process metrics to quantify properties of commits for the purpose of build failure prediction.

3.1.3 Change Characterization

Knowing the characteristics or type of a change made to the source code of a program can be useful for many different analysis tasks. Especially in software evolution research it is a common problem to extract meaning from information contained in VCS commits. For example, it was found that large commits are more likely to come from code management activities [HL08]. Another challenge is to automatically determine the *intention* of a commit, e.g., whether the change was to fix a bug, to introduce a new feature, or to update documentation. Such intents may have different implications in a process, e.g., in CI where updating a documentation file is unlikely to result in a compilation error.

Change Activities and Commit Types

Swanson [Swa76] laid the groundwork for characterizing commits by their intent in the context of software maintenance. He proposed that maintenance activities can be categorized as a) *corrective*, b) *adaptive*, and c) *perfective*. Various researchers later amended these categories for their specific use cases [HGH08, HL08, YFZ+16]. Hindle et al. [HGH08], for example, extended Swanson's categories by a) *implementation*, and b) *non-functional*. Table 3.1 lists the categories and the issues addressed by the categories.

Table 3.1: Change activities according to Hindle et al. [HGH08]

Type	Description
Corrective	Processing failure, performance failure, implementation failure
Adaptive	Change in data environment, change in processing environment
Perfective	Processing inefficiency, performance enhancement, maintainability
Implementation	New requirements
Non functional	Legal, source control system management, code clean-up

To find the category fitting a specific commit, they developed a taxonomy of 26 change types that were each mapped to a category. Change types include, Legal (e.g., a change in the license file), Merge (when a VCS branch was merged), Testing (when unit or integration tests were added or modified), and other typical development commit types. To determine the change type, commits were classified manually by reading the commit log messages, looking at the filename extensions of the changed files, and studying the diff of the commit. Each commit was then assigned a list of types determined by the procedure.

Methods on classifying the change activity are typically based on analyzing the commit message, i.e., a textual summary of the change written by the developer. Most approaches simply scan the message for predefined keywords. Such algorithms work under the assumption that a single commit is being classified. We will later see that this is impractical for our purposes, because our observations contain multiple commits (see Section 4.4.1), and we can therefore not assign a distinct change activity to an observation. We later adapt these change activity measures by capturing the changes made to specific file types (see Section 6.3.2). Also, we translate the concept of change activities into the context of CI builds (see Section 6.4.1.

Quantitative Characteristics

Many process metrics (see Section 3.1.2) exist that can be used to quantify properties of commits or sets of commits. Typically, quantitative analysis of changes are interested in the size and complexity of changes, or the amount of authors involved [Has09, MJ15].

3.1.4 Other Software Repositories

MSR techniques are not limited to VCS repositories. Data from other software repositories, such as issue trackers [KSD11], or mailing lists [BDL10], have also been used for various research tasks. We omit such sources in our approach, because the additional dimensions would increase the complexity beyond the scope of this work.

Relevant to this thesis, besides VCS repositories, are build automation systems. In the broader sense of MSR, build automation systems that track historical data can also be seen as software repositories. Methods for mining such repositories have not yet been systematized in the same manner as those developed for mining VCS. Only very recently, resources such as TravisTorrent[1], a database of build data gathered from Travis-CI (see Section 2.3.2), have appeared that support analyses of CI build systems.

3.2 Systematic Analysis of Software Build Errors

Build software automation is important for efficient development. Although build failures have a large impact on development efficiency [KKA14], the nature of automated software build failures has not been researched thoroughly. This thesis is novel in that it analyzes both the multiplicity of error types of CI builds, and the runtime behavior of such error categories. Previous studies either focus on errors of private builds in the edit-compile-test cycle, or consider only a binary build result outcome.

A study by Seo et al. [SSE^{+}14] focused on analyzing error categories and frequencies in the edit-compile-test cycle of developers. The goal was to understand the build process of a large and distributed organization, and the different errors that may occur within this process. Google's development environment includes a centralized cloud-based build system from which build data was gathered over a period of nine months. Over 26 million builds triggered by around 18,000 developers were analyzed from historical data of this build system. To understand the frequency of specific errors, a taxonomy of error *kinds* (such as syntax errors, or semantic errors), and a mapping strategy of build messages to these error kinds were created. Methods of qualitative research were employed to elicit initial categories for errors from the log data created by the build system. Compiler experts were then interviewed to decide which category was the best fit for an error message. With this knowledge, a parser was developed to automatically categorize new builds based on their log output. They found that, approximately 10% of the error types account for 90% of the build failures, where dependency errors are the most common. It takes developers at average one and at most two build iterations to fix a broken build.

[1] `https://travistorrent.testroots.org/` (accessed: 2016-08-10)

Kerzazi et al. [KKA14] conducted both quantitative analysis and interviews to understand build errors in a commercial enterprise web application with around 200 employees. A total of 3,214 builds produced over a period of six months were analyzed. The analyzed build data was gathered from the centralized build server used in a CI setting. A quantitative analysis was conducted to investigate the frequency of build failures, and the cost of fixing them. In a qualitative study, 28 software engineers were interviewed to understand which circumstances lead to build failures, and how failures impact the productivity of the team. They find that, from the 17.9 % of broken builds, the main causes for failures are missing referenced files, mistakenly checked in work-in-progress, and transitive dependency errors. The quantitative study revealed that the most important factors related to build failures include the authors role in the project (developer, integrator, etc.), the build type (integration build or continuous build), and the nature of the work (bugfix, feature development, etc.).

3.3 Software Defect Prediction

Software defect prediction has been subject of intense research over the past two decades [GKMS00, MPS08, ABJ10, HBB+11, DLR12, MJ15]. Understanding where and why software defects occur is a key research question for software quality assurance. Predicting future defects can potentially help to focus quality assurance efforts on the most defect-prone system components or problematic development processes [RD13]. Creating prediction systems involves the measurement of software defects, and using this measurement as dependent variable for statistical modeling.

3.3.1 Measuring Software Defects

Different definitions and methods of measuring *software defects* can be found throughout research. Often, measuring defects relies on analyzing user submitted reports to a bug database. Bug reports are then linked to specific components within the system [DLR12]. Such components can be software artifacts, such as source code files and modules, or logical software components, such as methods and classes.

A common way of quantifying software defects is to calculate the amount of *post-release defects*. These are measured by counting the amount of bug reports submitted in a specific time window after the software is released, e.g. six months after a release [NB07]. This measurement is then used as a numeric dependent variable to train statistical models to predict future defects, typically using regression or Bayesian probability models [HBB+11].

3.3.2 Software Metrics

Software metrics are the foundation for training statistical models to predict future defects [NBZ06]. The most common metrics used as independent variables for defect prediction in previous research can be categorized into product, process and metrics.

Product metrics quantify properties of a software product. Complexity measures of software artifacts are popular product metrics and have been subject of research for many years [BBM96]. Such complexity metrics include the number of methods of a class, object-oriented coupling, etc. There are a vast variety of product metrics which have been studied in great detail [RHT+13]. Because their calculation involves analyzing the source or byte code of a program, mining them can be costly [MPS08]. In our approach, we therefore focus on metrics that can be calculated directly from the version history of the VCS, i.e. *process metrics*.

Process metrics calculated from the software change history are popular for prediction models. Recent research suggest that process metrics have better defect prediction accuracy than traditional product and code-complexity metrics [RD13, MJ15]. Additionally, mining process metrics can be more cost-effective when compared to code-complexity metrics [MPS08].

A comparative analysis of process and complexity metrics by Rahman and Devanbu [RD13], strongly favors process metrics for defect prediction, in a release-oriented setting. These metrics are calculated per file of its 'release-duration', which means they are aggregated in the temporal window between releases, i.e. a specific set of commits. Additionally, neighbor metrics are employed, based on the co-commit history approach introduced by Kim et al. [KZWZ07]. For a given file F and release duration R (number of changes between releases), these metrics are calculated from the files co-committed (changed and committed together) with F, weighted by the frequency of co-commits in the history of R.

In an empirical study about the impact of process metrics in defect prediction models, Madeyski and Jureczko [MJ15] compared product and process metrics collected from a large amount of previous publications. They identified the four most popular process metrics to be: number of revisions, number of distinct committers, number of modified lines, and the number of defects in previous versions. They found that, the number of modified lines and the number of distinct committers are the most useful metrics with regard to the defect prediction models. We later adapt these four metrics to fit our CI data, and study their influence on build failures.

3.4 Build Failure Prediction

While software *defect* prediction has been studied to a great extent, software *build failure* prediction has not seen the same amount of attention. Little is understood about factors influencing build failures, especially in the context of CI. In a broader sense, a build failure can be seen as a type of software defect. Methods used in existing approaches are therefore similar to those used for defect prediction. In contrast to measuring software defects, measuring build failures is straightforward: a build can either be successful or unsuccessful. In the context of CI, we are interested in the outcome of an *integration*.

3.4.1 Measuring Integration Outcome

What constitutes a *successful* integration is subject to interpretation. The criteria that have to be satisfied in order to conclude a successful integration may vary depending on the project, the used technology stack, and the employed quality assurance metrics. Various definitions of *integration*, *build result*, *integration outcome*, etc. exist in research.

Cataldo and Herbsleb [CH11] studied a project in which an integration and testing team was responsible for the integration of new features. After merging the source code changes, a collection of integration testing suites were executed, and their outcome documented. The study then interprets the outcome of an integration as a binary result:

> *Our outcome measure is a dichotomous variable where a 1 indicates that at least one of the tests performed by the [integration and testing] team at the time of integrating a feature failed. Otherwise, the variable is set to 0.*

The organization investigated by Kerzazi et al. [KKA14] uses a centralized CI server that runs an automated build for an integration. The build server reports either success, partial success, or failure. Partial success means that the changes were merged correctly and the software compiles, but not all automated tests have passed. The study considers this outcome as a *build failure*, resulting in a binary outcome variable.

Duvall et al. [DMG07] give a more general definition:

> *A broken build is anything that prevents the build from reporting success. This may be a compilation error, a failed test or inspection, a problem with the database, or a failed deployment.*

There is no clear definition of a *successful integration*. Consequently, there is also no clear definition of when a build should fail. Generally though, we can assume that, *at the least*, successful integration means that a) there are no conflicts when merging the changes into the source code repository, b) the source code can be compiled correctly, c) all automated tests pass. The list of criteria can be extended to suit the project's needs. For example, code quality metrics can be checked for specific thresholds (e.g., test coverage or code complexity).

The hosted CI system Travis-CI records three levels of build outcome: errored, failed, or passed (see Section 2.3.2). In our approach, we use this measure for various experiments. For analysis where a dichotomous variable is required, we combine Travis' errored and failed category. A detailed explanation of how the outcome of an integration is modeled is given in Section 4.4.2.

3.4.2 Predicting Software Build Failures

In an early study in 2006, Hassan and Zhang [HZ06] attempted to use decision trees to predict the result of a build. They categorized factors impacting build results into: social (such as work habits), technical (such as software structure and complexity), coordination (such as parallel changes to files by different developers), and prior-build-result factors. However, the study is based on various outdated assumptions, e.g., that test suites are executed manually and not on dedicated infrastructures such as CI build servers. From historical VCS and build data, they mine metrics that describe the defined influence factors. Using these metrics as independent variables, they train a decision trees to predict a binary build result. Their best decision tree model to predict that a build will fail has an average accuracy of 69%.

Cataldo and Herbsleb [CH11] examined different factors leading to integration failures (see Section 3.4.1). The outcome of a feature integration was recorded by the integration and testing team, who were responsible for running test suites against the new features. Technical factors of integrated features, such as lines of code changed, as well as characteristics of the teams developing features, such as the ownership of features, were examined using logistic regression. Furthermore, to study the effect of cross-feature interaction, the amount of architectural dependencies between components of two features was extracted from the project's software architecture documentation. They found that a) organizational attributes (such as the amount, and geographic distance of developers involved in developing a feature) have a higher impact on integration failures than technical attributes; and b) that a high number of cross-feature dependencies increases the likelihood of integration failures.

Kerzazi et al. [KKA14] conducted an empirical study in the context of a large software company, and the impact of build failures on the development process. The study was conducted in an unnamed software organization, distributed across Canada and India, which is claimed to have 200 employees that maintain a large .net code base since 2004. The organization is said to employ CI and to use a central build server that builds the project when changes are triggered in the VCS. A key research goal was to elicit factors impacting *build breakage* (build failures) from both a quantitative analysis of the gathered data, and interviews conducted with developers. A set of eight hypothesized factors were tested with different statistical methods against the data, whether they significantly contributed to build failures or not. Using Random Forest as statistical model for measuring variable importance, their quantitative analysis revealed that, *"the type of role, the number of simultaneous contributors in the branch, the nature of the work (Feature, Bug fix, etc.), the build type ([integration build] vs. [continuous build]), and the period of the project are the most important factors related to build breakage."*

In a patent application, Bird and Zimmerman [BZ14] describe a system and methods for predicting software build errors. Much like established methods in software defect prediction, their approach is to extract characteristics of software changes, and calculating probabilities for these characteristics to produce build errors using logistic regression.

Socio-Technical Factors

Other studies in the context of build failure prediction mostly analyze socio-technical factors, such as congruence [Sch10, KSD11] or developer communication [WSDN09].

Using social network analysis on developer communication, Wolf et al. [WSDN09] were able to utilize graph metrics to build a predictive model for CI builds that achieves precision values between 0.5 and 0.76, and recall values between 0.55 and 0.75. The data used in the study was obtained from a single closed-source project at IBM, making it difficult to reproduce the results.

Extending the work by Wolf et al. [WSDN09], Schröter [Sch10] defined the notion of *socio-technical congruence* as the alignment between technical dependencies and the social coordination of the project. He found that interactions between developers and work items, modeled as socio-technical networks, are strong predictors for build outcomes.

Kwan et al. [KSD11] substantiated the hypothesis that socio-technical congruence is related to build outcome. Specifically, they believed that high congruence leads to a higher probability of build success. A build is considered successful, if the software compiles without errors, and passes every automated test case. By modeling social interaction and relationships between work items (e.g., issues, builds, change sets), graph metrics were calculated and correlated with build outcome. An empirical study conducted by Kwan et al. on a project at IBM, showed that, a high congruence correlates with high success probability for continuous builds, and, conversely, high congruence correlates with low success probability for integration builds. There are also some conflicting results, e.g., that a large portion of zero-congruence builds are successful.

3.4.3 Machine Learning Methods

In the past, a variety of different machine learning algorithms have been used to train classifiers that predict build outcome. Table 3.2 shows an overview of the different algorithms, and the publications they were employed in.

Table 3.2: Machine learning algorithms employed for build outcome prediction

Algorithm	Publication
C4.5 (Decision Trees)	Hassan and Zhang [HZ06]
Bayesian classifier	Wolf et al. [WSDN09]
Logistic Regression	Catadlo and Herbsleb [CH11] Kwan et al. [KSD11] Bird and Zimmerman [BZ14]
Random Forest	Kerzazi et al. [KKA14]
Support Vector Machines	Schröter [Sch10]

Most prediction approaches use a binary build result (failed or passed) as dependent variable. In our approach, we use different-sized variables to describe the build outcome. Some algorithms, such as support vector machines, by default only support binary classification, or assume that the classes are linearly separable. Standard methods of logistic regression or support vector machines are therefore not applicable. We later focus on methods that naturally support non-linear multi-class classification (see Section 2.4), such as decision trees and random forests.

CHAPTER 4

Solution Approach

This chapter presents the overall research goals, the employed methodologies, and solution approach. Section 4.1 gives a high-level overview of the work. Section 4.2 describes the core methodologies and the research setting. Section 4.3 outlines our solution approach and summarizes the main parts of the thesis. In Section 4.4, we define the system model and formalize domain concepts. Finally, Section 4.5 briefly introduces the polyglot data analysis toolkit we developed to perform the various data science tasks.

4.1 Overview

The overall goal of our approach is to create multi-categorical runtime-aware build failure prediction models for software projects from publicly available data. To that end, we first examine data gathered from 14 open source software (OSS) projects in a mixed-method study. We later use the results of the study for our build failure prediction approach. The study comprises two parts. In part one, we explore the multiplicity of build errors and their frequency of occurrence during the build execution. We later incorporate this temporal dimension of build errors into our prediction approach. In part two, we study aspects of the continuous integration (CI) workflow that can be correlated with build results. Building on existing research, we elicit measurable properties of build data. We perform correlation analyses to examine the strength of the relationship between these properties and the build outcome. The results of the study are used in our approach to create a system for build failure prediction. First, we train classification models using machine learning algorithms to predict the outcome of a build. Various machine learning algorithms are evaluated in terms of their basic prediction performance. We then proceed to incorporate our understanding of the temporal dimension of build errors into our prediction approach. The approach is evaluated using common statistical model validation techniques.

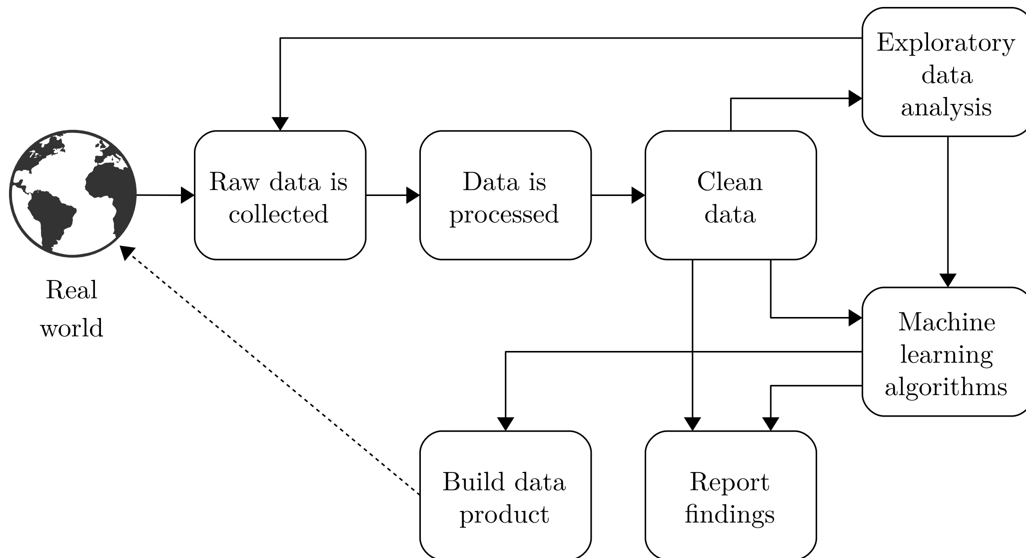


Figure 4.1: Overview of the data science process according to Schutt and O'Neil [SO13]

4.2 Methodology

We follow the general data science process according to Shutt and O'Neil [SO13], illustrated in Figure 4.1. For each step we employ methods from respective research fields. Web data extraction techniques are used to acquire data from publicly available sources. The gathered data come from different sources and heterogeneous data models. A large structural gap exists between build metadata, build logs, and the graph-based version control system (VCS) history. To perform analyses, the data first have to be homogenized, linked, and cleaned of outliers. This process is referred to as *data wrangling* [McK12, SO13]. Using our developed data analysis toolkit CInsight, introduced in Section 4.5, we build wrangling pipelines to structure, link, and clean the data. We explore the normalized dataset in a mixed-method exploratory study, using methods from both qualitative and quantitative research. Qualitative research methods, such as *coding* [CS14], help us to analyze the qualitative data, and to formulate grounded theories; e.g., for an error type categorization from the build log data. Methods from quantitative research are used to explore the different hypothesized influence factors, and substantiate or refute these hypotheses. Statistical tools, such as R[1] and WEKA[2], allow us to explore and visualize the data in different ways. We employ mining software repositories (MSR) techniques to calculate metrics from the build data and source code repositories. Machine learning methods are used to build and evaluate different classification models for predicting build outcome.

[1] https://www.r-project.org/
[2] http://www.cs.waikato.ac.nz/~ml/weka/

4.2.1 Research Setting

The study and evaluation of our approach is conducted using real-world data gathered from publicly available data sources. We examine data from 14 OSS projects that employ CI using GitHub and Travis-CI (see Section 2.3.2). To contain the complexity of the data analysis process, we restricted the study to systems written mainly in the Java programming language, and projects that use either Maven or Gradle (see Section 2.2) as their build automation tool. This way, we can build on various working assumptions about the data, e.g., codebase structure, build log output format, or testing environment.

High-profile projects fitting these requirements were selected by first querying the GitHub repository list. The initial set of repositories was then filtered by their use of Travis-CI and their level of activity (high number of commits, contributors, and builds). Table 4.1 list the projects that were ultimately selected for the study.

Table 4.1: Name and description of projects used as research subjects

Name	Description
Apache Storm	Distributed Computation Framework
Crate.IO	Scalable SQL database
JabRef	Graphical Java application for managing BibTeX databases
Butterknife	Android Dependency Injection Library
jcabi-github	Object Oriented Wrapper of Github API
Hystrix	Latency and fault tolerance library for distributed systems
Presto	Distributed SQL query engine for big data
Openmicroscopy	Microscopy data environment
RxAndroid	RxJava bindings for Android
SpongeAPI	Minecraft plugin API
Spring Boot	Java Application Fraemwork
Square OkHttp	HTTP+HTTP/2 client for Android and Java
Square Retrofit	HTTP client for Android and Java
WordPress-Android	WordPress for Android

4.3 Approach Outline

The overall goal of our approach is to create a system for predicting the outcome of a CI build. Figure 4.2 gives an overview of the entire process. In the following subsections we outline the overall approach and briefly summarize the steps involved. First, real-world build and VCS data are collected (Section 4.3.1). The data are processed and structured using wrangling pipelines and our developed method of *topology mapping*. We then conduct an exploratory study consisting of two parts. The goal of our two-part study is to gain a deeper understanding of CI build errors (Section 4.3.2), and factors that influence the outcome of a build (Section 4.3.3). The results of the study are then used to build a system for predicting build failures (Section 4.3.4). Using the acquired data, we train and evaluate different classification models using machine learning algorithms.

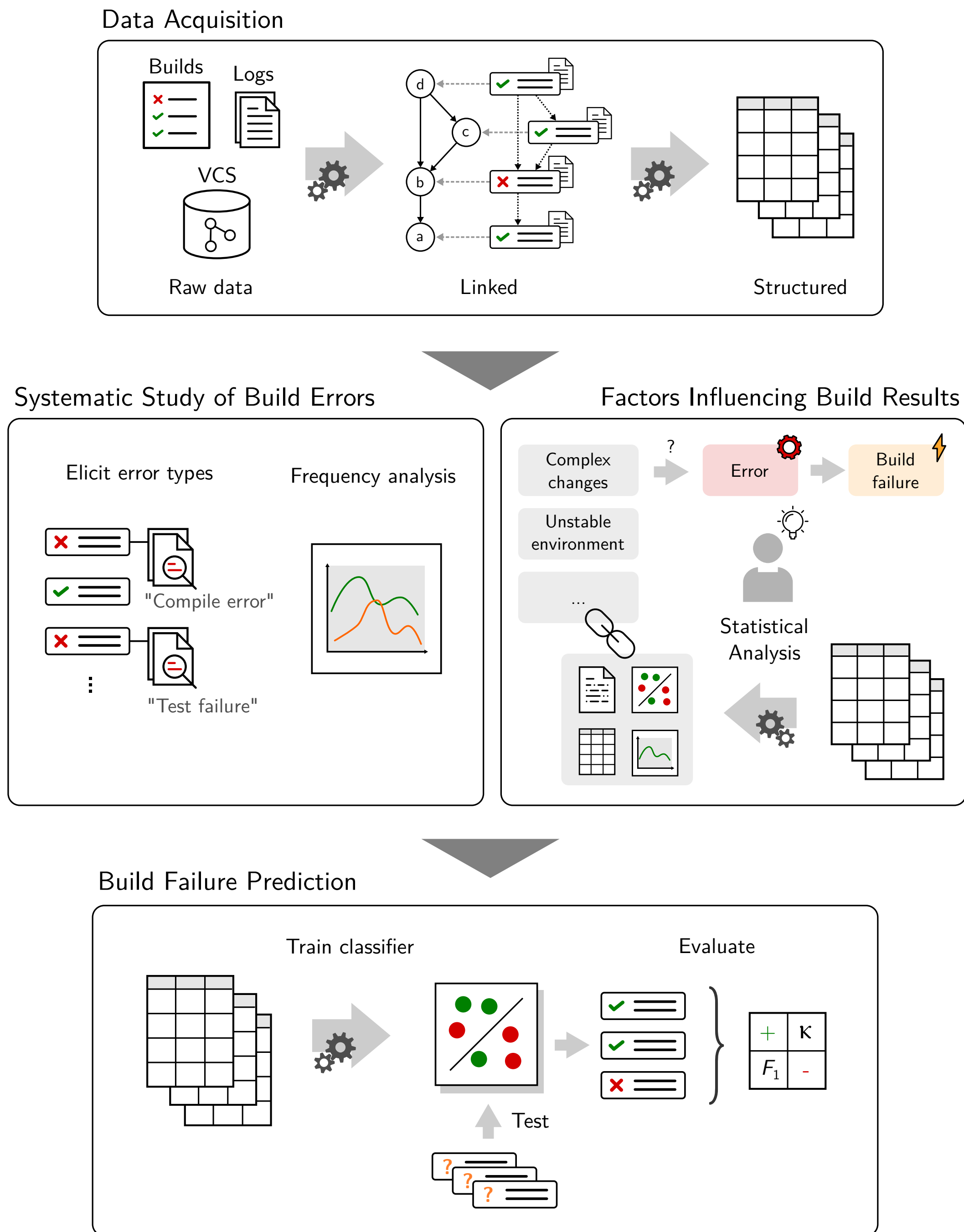


Figure 4.2: Solution approach overview

4.3.1 Data Acquisition

Our goal is not only to gain new insights into the multiplicity of build errors on CI systems, but also to understand how changes to the source code impact build results. We therefore require two main kinds of real-world data: build data from CI servers, and change data from VCS repositories. Metadata of build runs (outcome, runtime duration, etc.) as well as log output of the build system are the basis for our study. Most VCS provide a rich change history and methods to access different kinds of information about changes. Our analyses require metadata about commits (author, date, etc.), as well as revision deltas, that allows us to analyze the actual changes.

Many publicly available CI systems provide APIs to access exactly these kinds of data. However, our data analysis approach relies heavily on ad hoc queries. Because of the large volume, running such queries on the on-line data would be impractical. Using web data extraction methods, we target a CI platform and extract the data we require for running analyses locally. Distributed version control systems (DVCS) are ideal for local analysis, because, as such, they provide a local copy of the entire change history [BRB+09]. Public repository hosting platforms have emerged, on which many open source projects publish their source code [DSTH12]. Because of their popularity and easily accessible APIs, we choose Travis-CI and GitHub as the concrete data sources for build data and VCS data respectively.

We process the raw data using wrangling pipelines built with our data analysis toolkit. Using our developed method of *topology mapping* (see Section 6.1.1), we map change data from the project's source code repository to the CI build data. The normalized and cleaned data are then used as input for our part study. We discuss concrete wrangling techniques in individual chapters as the techniques become relevant.

4.3.2 Systematic Study of Build Errors

The first part of our study is discussed in Chapter 5. In this first part, we analyze both qualitative and quantitative data on build errors. Build automation is a key aspect of CI. However, little is understood about the multiplicity of errors that can occur during a CI build. To gain an understanding of such errors, we first devise a method for determining the build error type (e.g., *compile error* or *test failure*) from the log output generated by build tools. Furthermore, we examine the temporal aspects of build results. We analyze frequency of occurrence of different build errors within the build execution duration. Using methods from probability theory, we later calculate the likelihood of such build errors occurring at specific points in time during the build execution.

4.3.3 Systematic Study of Factors Influencing Build Results

The second of our study is discussed in Chapter 6. The goal of this second part is to gain a deeper understanding of factors that influence software build results in CI workflows. We elicit a set of measurable properties (metrics) of CI builds, that can be calculated

from CI and VCS domain data. These are later used as features for training machine learning models that predict the outcome of a build. Metrics used in existing research on software defect [MJ15] and build failure prediction [HZ06, KKA14] are used as a starting point for the study. By performing correlation analyses, we determine the strength of the relationship between these metrics and the build outcome. We proceed to use our data analysis toolkit to extract the metrics as feature vectors, and to export training data used as input for creating build failure prediction models.

4.3.4 Build Failure Prediction

Based on the results obtained from our two-part study, we aim to create machine learning models for predicting build failures. Our approach is discussed in Chapter 7. We first examine how well existing build failure prediction approaches are applicable to the CI domain. We build classification models using different algorithms and training features. The models are evaluated using standard model validation techniques [Alp14, Dem06]. Next, we incorporate the temporal dimension of CI builds into our prediction approach. The results from our analyses on the frequency of occurrence of different error types enable us to reason about a prediction during the build execution. The experiments are performed using WEKA, a popular machine learning toolkit that provides a variety of training algorithms for classification tasks and parameterization of these algorithms.

4.4 System Model

The following section gives an overview of the system model and of different dimensions of the CI domain relevant for our approach. We also present a formalization of domain concepts that will later enable us to derive and concisely define functions that quantify properties of these concepts.

4.4.1 Domain Model

The CI domain comprises several dimensional layers. Concepts of build automation and revision control coalesce in a development process model (see Section 2.3). A CI domain model must therefore incorporate concepts of build automation, VCS, and the CI workflow. We now formalize these concepts and introduce term definitions.

Revision Control

A VCS repository manages a history of changes (commits) to files (or more generally, content) in the form of a directed acyclic graph (DAG). We define a repository R as the collection of files F, and a directed acyclic commit graph $G_C = (C, P)$, where $C = \{c_0, c_1, ...\}$ is the set of commits uniquely identified by their SHA-1 hash sum. $P \subseteq \{(c_v, c_u) \mid c_v, c_u \in C, c_v \neq c_u\}$ is the set of directed edges that describe the parent relation between commits. A directed edge from c_1 to c_2, i.e. (c_1, c_2), denotes that c_2 is a parent of c_1. A commit with more than one parent (out-degree ≥ 2) is a *merge*

commit. We define a commit $c \in C$ as a tuple: $c =$ (sha, date, author, message, D), where D is the set of diffs, i.e., the set of changes encapsulated by the commit. Formally, we define a diff $d \in D$ as a tuple $d = (f, s, l_+, l_-)$, where $f \in F$ is the file being changed, $s \in \{add, del, mod\}$ is the modification status of the file (added, deleted, or modified), and l_+ and l_- are the number of lines added to and deleted from the file, respectively.

Given a tuple t, the notation $t.x$ is used to reference the element in tuple t denoted by the symbol held by x. For example, $c.sha$ references the SHA-1 hash sum of commit c.

We define $F_c \subseteq F$ as the set of distinct files that were changed by commit c, i.e., $F_c = \{d.f \mid d \in c.D\}$.

Build Automation

Build automation systems allow the configuration and automation of the software build process (see Section 2.2). A build consists of various consecutive tasks, e.g., the compilation of the source code, or running automated unit tests. Build tasks that do not depend on each other can also be parallelized (e.g., static code analysis, integration testing, or running the build in different configurations). Each task reports the result of its execution to log output in human readable form. The final outcome of a build can be interpreted in different ways. Section 4.4.2 discusses how we model the build outcome for our approach. In our study we examine metadata about the build process, and log output of the build execution. Travis-CI has devised an entity model that makes these concepts explicit[3]. Our definitions are based on their model.

A build b may comprise several *jobs*. A job is the execution of a specific set of build tasks. The log output of each job is stored in a separate logfile. In the remainder of this thesis, when we refer to properties of a *build*, unless explicitly stated, we refer to build metadata (e.g., the person who triggered the build, or the time it was started), as well as properties of the individual job that was identified to cause the build failure (e.g., the result or execution duration).

We define the execution duration (or runtime) of a build, as the time it takes for the build to report a result. Because jobs are executed in parallel, determining both the build outcome and execution duration in cases where multiple jobs fail may be ambiguous. A detailed description of our disambiguation approach and how the build execution duration is measured is given in Section 5.2.1.

CI Workflow

In a CI workflow, a VCS and a build automation server are used conjointly. Our model has to reflect this relationship. When developers publish their changes (as commits), the build automation server checks out the commits from the VCS repository, and executes the build process. Several commits may be checked out since the last build. These

[3] `https://docs.travis-ci.com/api#entities` (accessed: 2016-08-10)

commits are the *change set* of a build. We write C_b to denote the commits in the change set of build b. The latest commit of the change set *triggers* the build.

Example 4.4.1. Figure 4.3 shows an example scenario to illustrate the concepts of trigger commits and change sets. Commit a triggers build b_1 and commit d triggers b_2. The change set of b_2 includes all changes commited since the last build, i.e., $C_{b_2} = \langle b, c, d \rangle$

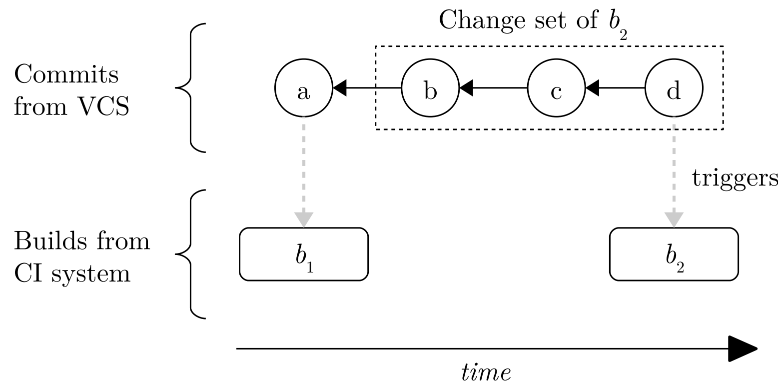


Figure 4.3: Relationship between builds and commits

If the trigger is a merge commit, we consider the *effective changes* to be all commits in the change set, except the merge commit. In case the change set includes *only* a merge commit, the effective changes are made up of that commit. This distinction is later used for determining the main author of a change set. Furthermore, differentiating trigger commits allows us to define the concept of a build type (see Section 6.4.1).

4.4.2 Modeling Build Outcome

Typically, the outcome of a build is interpreted as a binary result: *failed* or *passed*. A build *failure* denotes the fact that a build has not passed. We define a build *error* to be the root cause for a build failure. For example, a build may fail because of a compilation error, unit-test failure, etc. If no error occurred, we consider the build to be *successful*. Unrelated to our definitions, Travis-CI distinguishes between *errored* and *failed* builds, based on the execution phase the build terminated in (see Section 2.3.2). Based on these observations, we model the build outcome in three different ways. The build outcome is later used dependent variable for our statistical analyses.

- The *binary* outcome: did the build pass or fail?
- The *state* of a build in terms of the Travis-CI lifecycle: *errored*, *failed*, or *passed*. Builds in the state *errored* or *failed* are considered build failures.
- The build *result*: either *successful*, or one of the different error types (*compile error*, *test failure*, etc.)

4.5 CInsight Data Analysis Framework

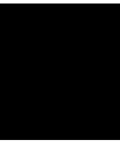
In order to perform the multitude of data science tasks, we developed CInsight: a polyglot system that facilitates collection, structuring, extraction and finally preparation of CI domain data for further processing with data mining tools. It leverages a variety of other technologies, such as pandas[4] (a Python data analysis toolkit), R[5] (statistical computing environment), whatthepatch[6] (a library for parsing diff patches for python), etc.

The system comprises the following modules:

- **Crawler:** A Web crawler application written in Java that gathers CI domain data. Specifically, it crawls build metadata from the Travis-CI API, and stores it into a local MongoDB instance for ad hoc analysis. Furthermore, it fetches the raw build log data from Travis-CI's storage service, namely Amazon S3.
- **Marvin:** A bot written in Python that fetches pull request data from a GitHub repository when builds are triggered on Travis-CI. The bot's purpose is to retain historical VCS data that may get lost during the development process (see Section 6.1.2).
- **LogCat:** A tool to aid the process of categorizing build results based on build log data. It is written in bash and uses Redis for set operations.
- **PyMunger:** A toolkit, written in Python, to facilitate data processing, linking, and extraction of CI data gathered by the Crawler, and the locally cloned Git repositories.
- **RLoupe:** A set of R programs to facilitate exploration, visualization, and statistical analysis of the extracted data.

[4] http://pandas.pydata.org/pandas-docs/stable/
[5] https://www.r-project.org/
[6] https://github.com/cscorley/whatthepatch

CHAPTER 5



Systematic Analysis of Build Errors

Software build automation is an integral part of efficient software development. Build automation is also a key aspect of continuous integration (CI). There has been some research on build errors of private builds, i.e., builds that developers execute in their local environment to compile the software [SSE+14]. However, little is understood about the multiplicity of errors that can occur during a CI build. In this chapter, we present an empirical study of both qualitative and quantitative data on build errors, gathered from 14 open source software (OSS) projects that employ CI. In Section 5.1, we first present methods and results on analyzing the multiplicity of error types of CI builds. In Section 5.2, we examine the runtime behavior of such error kinds during the build execution. The gathered insights are later used in our multi-categorical runtime-aware build failure prediction approach.

5.1 Build Error Categorization

As discussed in Section 3.4.1, build results are typically represented as binary variable: *failed* or *passed*. However, builds may fail due to a variety of different errors. The impact of a build failure on the development process depends on the error that caused the failure [SSE+14]. One goal of this thesis is to uncover what different errors occur during the build process, and how this knowledge can be incorporated into a prediction mechanism.

A CI build comprises different steps, i.e., build phases (see Section 2.2). In an initial phase, the codebase is checked out from the version control system (VCS) repository. The source code is then compiled, and the software is tested using unit tests. Depending on the project's build configuration, many other steps may be executed. Each of these build phases produce their own specific kinds of errors. Some errors will consequently occur more often at certain points in time during the build execution.

To further study the runtime behavior of build errors, such error categories have to be found. The log output of build tools is analyzed by searching for patterns that match error messages. Search patterns are then grouped to describe a specific error category. This allows us to associate logfiles to an error category based on the search patterns of the category. Our developed *open coding* workflow allows us to efficiently and systematically categorize a large number of logfiles. Next, we present this methodology in more detail.

5.1.1 Methodology

Parsing Logfiles

The projects we analyze use either Maven or Gradle as build tool. Both create log output of the entire build process, and report results of each phase. Typically, when a build fails, the build tools will terminate at the point that the error occurred, or report the error at the end of the phase. To identify a specific error kind, it is thus often enough to inspect the last few lines of the log output. To find all logfiles that exhibit the same error kind, we search for a distinct message in the log that describes the respective error. From such a message we generalize a regular expression, and search for logfiles matching this pattern. The process of finding patterns that can isolate an error category is not trivial and requires knowledge of both the build tool, and the build configuration.

Example 5.1.1. Compilation errors are a common reason why builds fail. Listing 5.1 shows the log output of a build from the Spring Boot project. The maven-compile-plugin, responsible for compiling the source code, outputs a distinctive string (highlighted) to indicate this error. We know from the internal workings of the maven-compiler-plugin[1], that this message is reported if and only if the compilation process fails. Therefore, unless the project uses a custom compilation procedure, searching for this string is sufficient to identify errors of this kind among all logfiles.

Listing 5.1: Log output of a build from the Spring Boot project that caused a compilation error

```
1  [INFO] Compiling 67 source files to /home/travis/build/spring-projects/spring
2  -boot/spring-boot-autoconfigure/target/classes
3  [INFO] -------------------------------------------------------------
4  [ERROR] COMPILATION ERROR :
5  [INFO] -------------------------------------------------------------
6  [ERROR] /home/travis/build/spring-projects/spring-boot/spring-boot-autoconfig
7  ure/src/main/java/org/springframework/boot/autoconfigure/redis/RedisAutoConfi
8  guration.java:[143,10] cannot find symbol
9  symbol:   class ConfigurationProperties
10 location: class org.springframework.boot.autoconfigure.redis.RedisAutoConfi
11 guration
12 [INFO] 1 error
```

[1] https://svn.apache.org/repos/asf/maven/plugins/trunk/maven-compiler-plugin/src/main/java/org/apache/maven/plugin/compiler/AbstractCompilerMojo.java (accessed 2016-08-28)

LogCat Open Coding Workflow

To elicit error categories the build data have to be systematically explored. A well known method in qualitative research, employed for the systematic exploration and labeling of data, is the concept of open coding [CS14]. In the work by Seo et al. [SSE+14], open coding was also employed to elicit error categories from compiler messages. In their study, after categories were defined, experts were interviewed to map error message kinds to those categories. We developed *LogCat*, a tool to assist the process of analyzing logfiles based on the concept open coding. LogCat systematizes the exploration process of open coding, and facilitates the efficient categorization of build logs from a project in a short period of time.

Figure 5.1 illustrates the LogCat workflow as a flowchart. From the set of logfiles that have not yet been associated with a category, one random file is selected as sample. The log output is examined, and a tentative hypothesis is formed about the message that captures the (possibly yet undefined) error category. Once a pattern has been defined, it is tested by sampling a different logfile that contains this message. If enough evidence exists to support the hypothesis, all logfiles containing the message are assigned the given error category. This process is repeated until all logfiles (or certain percentage thereof to an acceptable margin of error) have been successfully assigned a category.

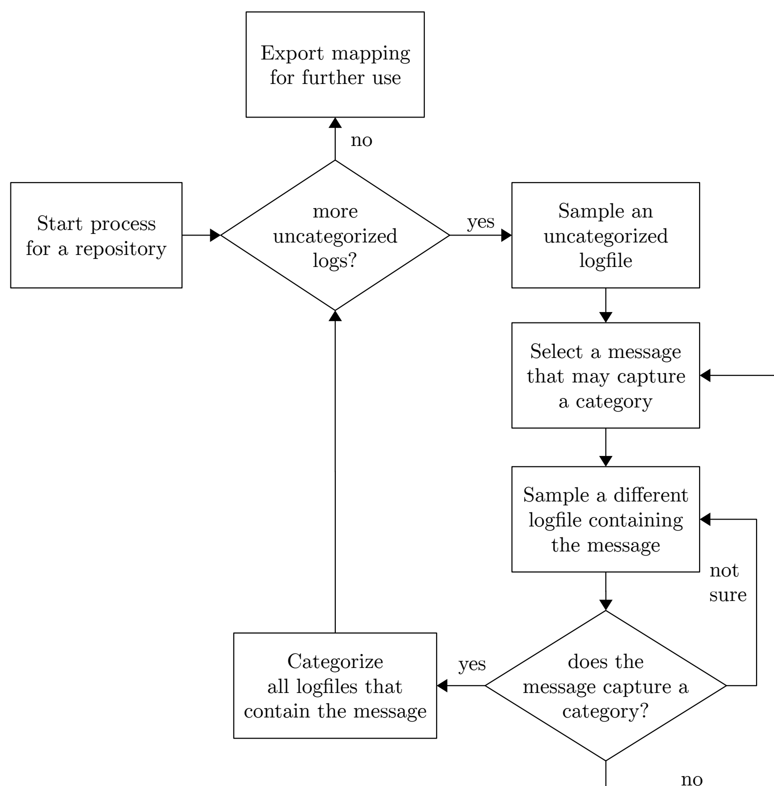


Figure 5.1: A flowchart of the LogCat open coding workflow

LogCat stores the results of the categorization as a mapping of category labels to a set of job IDs, and exports it as simple JSON for further processing. It also stores the patterns of each error categories. A log file matching at least one pattern is assigned the respective category. This way, an existing categorization model can be updated by simply applying the search patterns again.

Example 5.1.2. This example shows a LogCat categorization model as JSON output. This specific model captures all messages exhibited in Example 5.1.1. The `jobs` property holds the IDs of all jobs in our build database that were identified with the specific error category.

Listing 5.2: Example JSON output of LogCat

```
1  {
2    "categories": [
3      { "label": "compile", "pattern": ["^\[ERROR\] COMPILATION ERROR"] },
4      ...
5    ],
6    "jobs": {
7      "compile": [1,2,3,...],
8      ...
9    }
10 }
```

Systematic Categorization of Collected Log Data

The categorization process was performed for each project individually. Using LogCat and its parser, we first explored the data gathered from the projects' build data repository to create a taxonomy of error categories. We then extracted appropriate search patterns and labeled matching logfiles with the associated category. To reduce the amount of logs that have to be parsed, we gathered only logfiles of builds that have failed. In total, we analyzed 54 248 logfiles from failed jobs.

Maven and Gradle both support a variety of plugins that allow heavy customization of the build process. Naturally, individual build configurations will cause a certain disparity of error categories across projects. However, some build phases are common across build configurations, e.g., a compilation phase, as the compilation is a prerequisite for further build tasks. When we started the categorization process for a project, we focused on these commonalities first. As the categorization progressed, we developed new categories by studying the log output, and the project's build configuration. Some projects use an elaborate build configuration that makes use of plugins to solve specific task, e.g., measuring code quality. If we could not determine the error kind from the log data based on the examination of the build configuration, we analyzed the change that led to this specific build failure. We repeated this process until we were confident that the defined message pattern is indeed sufficient to capture the entire error category.

We omitted the error categorization for WordPress-Android. The project uses a very elaborate Gradle build configuration with a high number of build goals. Many of these, specifically those used for the Android packaging and release process, report vague or

incomprehensible error messages. We were not able to make sound decisions about the error types. Similarly, Openmicroscopy uses a large variety of programming languages, third party modules, and, consequently, a heterogeneous build environment. Additionally, their build environment has been subject to drastic changes over the project's lifetime. We were therefore unable to produce a concise error categorization.

5.1.2 Results

Overall, projects exhibit a mean failure ratio of 37%, as made evident by Table A.1. Only four projects have a failure ratio above 50%. In terms of Travis-CI build state results, most failed builds are in the state *failed*. We analyzed a total of 54 248 logfiles from failed jobs, roughly 92% of which were successfully assigned an error category. A total of 21 different categories were created, some of which were subsequently collapsed. For example, some projects use multiple static code analysis tools for measuring different aspects of code quality. We labeled errors generated by each tool separately (e.g., *checkstyle* or *findbugs*). While this may be desirable for prediction, for the purpose of comparing projects we grouped these categories into a single *quality* category. This resulted in a total of 14 and an average of 9 categories per project. Table 5.1 lists the 14 labels we created, the amount of projects they occurred in, and a short description of each error kind.

Table 5.1: Summary of error categories and their occurrence frequency across projects

Label	Projects	Description
testfailure	12	An automated test did not pass
compile	12	The compilation process failed
git	12	The build worker could not fetch the changes that triggered the build, e.g., because the pull request was merged before the build started
buildconfig	11	The build configuration has an error, e.g., a syntax error in the XML structure of the Maven pom.xml
crash	11	The build environment crashed or exceeded some time limit
dependency	11	The build configuration requests a dependency that could not be resolved, e.g., because of a wrong version number
quality	10	A code inspection reported that a quality criterion was not met
other	9	Errors without an identifiable cause
itestfailure	4	An automated integration test failed
doc	3	An error occurred while processing the documentation, e.g., because a method had undocumented parameters
license	3	A plugin that checks license criteria reported an error, e.g., because not all source files include a correct license header
incompatibility	2	An API incompatibility was detected, e.g., because of a version conflict
androidsdk	1	An error associated with the Android SDK occurred, e.g., because it could not be downloaded
buildout	1	An error of a specific build tool used for a submodule of Crate.IO written in Python

A build may comprise several jobs, and each job may fail independently. It is also possible that all jobs fail because of the same error. A build failure may therefore be caused by several different errors (see Section 5.2.1 for a more detailed explanation). If multiple jobs of a build fail, we only count each error types once. Table A.2 in Appendix A lists all 14 error categories per project, and the amount of builds categorized with the given error. Figure 5.2 shows a boxplot of the frequency of error categories common to 10 or more projects.

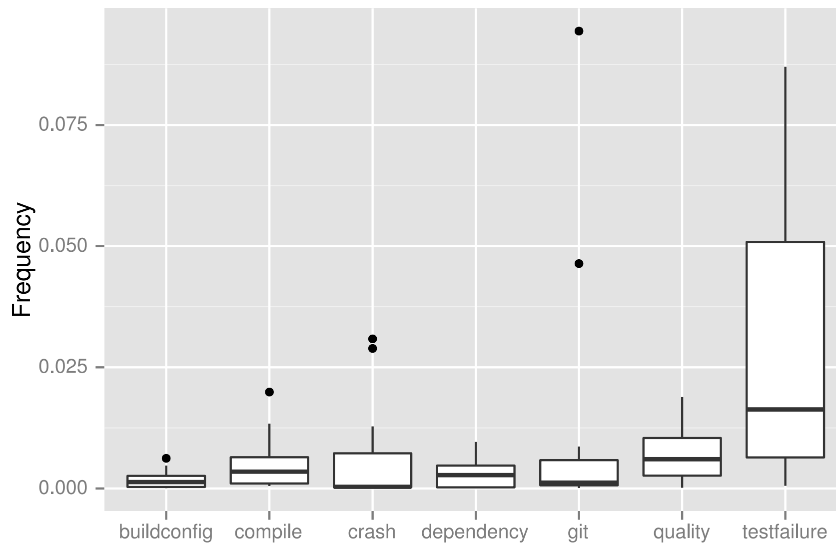


Figure 5.2: Distribution of common error categories

Among all categories and projects, the most frequent error types are: *testfailure*, *quality*, and *compile*. These errors account for approximately 62% of all reported errors. Together with the error categories *git*, *buildconfig*, and *itestfailure*, they make up more than 80% of all errors.

On average, 41% of builds fail because of test failures. Further splitting up this error category would require the dimensioning of test-configuration details, e.g., labeling which submodules are tested, or grouping kinds of tests. This would involve significantly more effort during the categorization, and require deep knowledge of the systems test configuration. It would also increase the number of defined categories, which may be undesirable for prediction purposes.

5.2 Runtime Behavior of Builds

This thesis is novel in that it analyzes the runtime behavior of errors in CI builds. We wish to understand 1) how the build execution duration evolves during the course of the

project, and 2) the frequency of errors at specific points during the build execution. By explaining build error kinds by their frequency of occurrence during a build, it becomes possible to update a prediction during the execution time. For example, when the most likely error predicted by a model is a compilation error, and we know that these errors most likely occur in, e.g., the first 20 seconds of the build, then we can update the prediction after 30 seconds saying that it is unlikely that the build will fail.

To get an initial understanding of the temporal dimension of the data, we first study how the execution durations of builds change as the examined project evolves. We then use the labeled data generated by the build categorization process to examine the frequency of occurrence of build errors during the execution time. That is, we study at which points in time during the build execution specific errors occur most frequently.

Next, we describe our data extraction and analysis methodology in more detail.

5.2.1 Methodology

To perform the analysis, we first need to extract the necessary properties from our data. Each build of a project is numbered by a sequential identifier, starting with 1. We calculate the runtime of a build using the date information of the build metadata provided by Travis-CI. Using these two properties, we effectively generate time series data that allow us to use methods from time series analysis to interpret the data [CM09]. We enrich the data with the labels generated by the error categorization process discussed in Section 5.1. With the runtime property and the error category association we can analyze the frequency of occurrence of specific errors during the build execution. We first explain how we measure the execution time (or runtime) of a build.

Measuring Runtime

Measuring the execution time (or runtime) of a Travis-CI build can be ambiguous. Travis-CI supports a concept called *build matrix*[2]. It allows to create a setup of multiple build environments with different configurations. For example, a Java project could be tested in both the Java 7 and 8 environment. Additionally, the project could be built in different configuration modes, such as development and production. This 2×2 configuration matrix expands to four individual jobs. When the build starts, Travis-CI spawns four workers that execute these jobs in parallel.

Some projects in our dataset make heavy use of this feature. For example, build #15037[3] of the Presto project included five jobs, one failed job and four with a runtime of roughly 30 minutes. The Travis-CI build metadata contains the accumulated total time over all jobs, in this case about 2.5 hours. As soon as a single job fails, by our definition of a successful integration (see Section 3.4.1), the build can no longer be considered successful.

[2] `https://docs.travis-ci.com/user/customizing-the-build/#Build-Matrix` (accessed: 2016-08-10)

[3] `https://travis-ci.org/prestodb/presto/builds/153706379` (accessed: 2016-09-10)

Although Travis-CI continues to execute the other jobs, for our purpose, the runtime duration of the build should be that of the first failed job. Conversely, when all jobs pass, the runtime duration should be that of the longest running job, and not the sum over all jobs. Travis-CI does not explicitly store this information. However, job metadata contain the date and time they were started and finished. This allows us to calculate the runtime duration of individual jobs.

Jobs may be scheduled at different points in time after the build initialization. Measuring the runtime from the beginning of the build initialization may thus not capture the true execution duration of the individual job that performed the build process. We therefore consider the runtime of each job individually. When multiple jobs fail, we consider only the first failing job in terms of execution duration. The overall runtime and state of the build is then the runtime and state of that job. In case there are no failing jobs, the runtime duration of the overall build is the temporal window between the build initialization and last finished job. Figure 5.3 shows a timing diagram of a build with three jobs, j_1, j_2, j_3, where job j_1 and j_3 failed at a specific point during the build execution, indicated by the red bars. In such a case, we define the build runtime as the shortest duration from the start of a job to the point the error occurred.

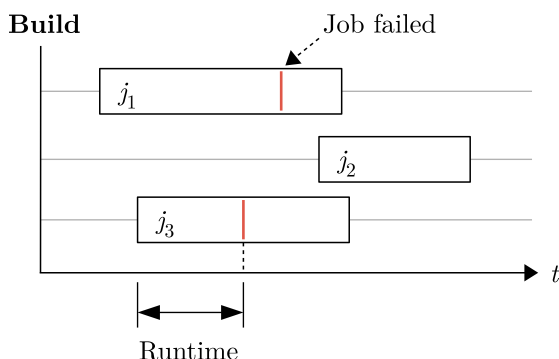


Figure 5.3: Measuring the build execution duration (runtime) when multiple jobs fail

Cleaning Outliers

The error category *crash* (see Section 5.1.2), includes builds that were terminated after 50 minutes of inactivity. Because we can not determine the point in time the build became unresponsive, these data skew our statistics. We clean the dataset of builds of these extreme outliers by removing data with a runtime above the 0.99 percentile of the particular project.

Besides the states *failed*, *errored*, or *passed*, builds may also be labeled with the states *canceled* (when the build was manually canceled by the user), or *started* (when the build is currently running during data extraction). Because the latter two are both rare and

uninteresting events for our analysis, we clear the dataset of any builds that are labeled with *canceled* or *started*.

With the cleaned dataset, we can now perform meaningful analysis.

5.2.2 Runtime Evolution

We first analyzed how the execution duration of builds changes as the examined project evolves. There are many events in the course of a project that can drastically change the execution duration of a build, e.g., adding or removing complex tests, changing the build configuration, or changes in the build infrastructure. To effectively use runtime occurrence frequency of errors, these changes have to be taken into account.

Because each build is identified by a sequential identifier, we can use the execution duration measurement to create a time series. We explored the data by creating plots of this time series, and examining the runtime behavior of specific error types. We next present results of this initial analysis.

Initial Results

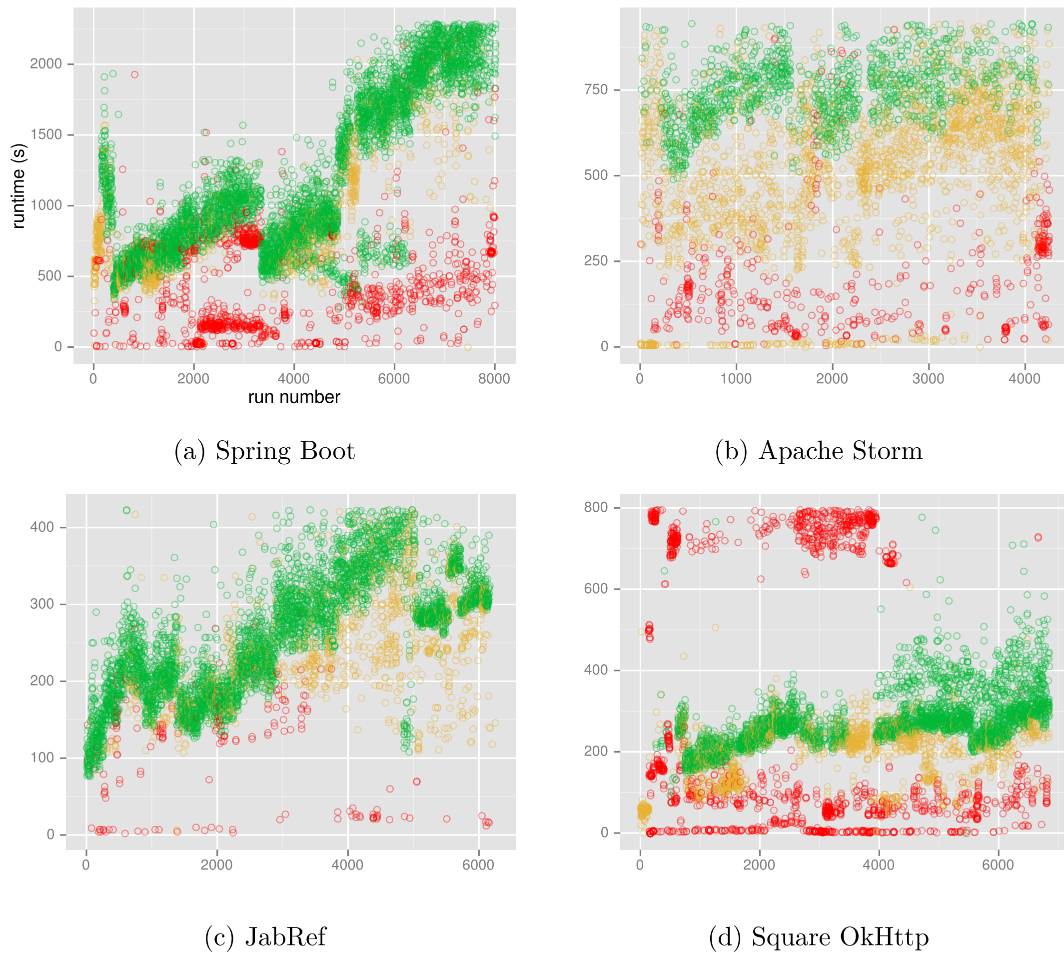
We plotted the time series data as a scatter plot, and highlighted result types returned by Travis-CI (errored, failed, or passed). Figure 5.4 shows scatter plots of data from four different projects. The x-axis describes the sequential run-number, beginning from the first build. The y-axis describes the build execution duration. The colors red, orange, and green indicate the build result: errored, failed, or passed respectively.

Errors in a build will cause it to terminate before any other phases are executed. Consequently, most errored builds have a shorter runtime than, e.g., passed builds. In Figure 5.4d, there is a cluster of errors that run significantly longer than any other builds during this timespan. Travis-CI workers may terminate builds that take longer than 15 minutes, and if a build, e.g., crashes during an early phase and the build is later terminated, this build will be labeled as errored, and have a runtime of 15 minutes. Such errors should be isolated by their error kind for any further detailed analysis of error frequencies to be meaningful.

All 14 projects analyzed in this thesis have points in their lifespan, where the average build runtime drastically changes. The four plots in Figure 5.4 are exemplary of this behavior. This is an important discovery for later stages when we analyze the frequency of occurrence of specific build errors.

Detailed Analysis

We present our detailed analysis using the Spring Boot project. The characteristics of this project are exemplary of the other 14 in our dataset. For the remaining projects, our analysis follows the same principles, although the actual results vary slightly from project to project, as made evident in Section 5.2.3 and Figure 5.7.



(a) Spring Boot

(b) Apache Storm

(c) JabRef

(d) Square OkHttp

Figure 5.4: Time series scatter plots of builds from four different projects

One would intuitively expect the runtime distribution of passed builds to be approximated by a normal distribution. However, as we have shown, average build execution durations may change drastically during a project's lifetime. When analyzing the runtime over the entire project-lifetime, it is therefore easy to explain why the runtime is not normally distributed. Figure 5.5 shows this disparity. The plot to the left shows a simple moving average (of the previous 50 data) over the time series of passed builds. The plot to the right shows a density histogram of build execution durations with a 24 second bin width.

We later reason about the execution duration of future builds based on the probability density of historical observations (see Section 7.1.3). However, as we can see from the histogram, the constructed probability density is not representative for builds from the later stages of the project. Therefore, using the entire dataset to construct the probability density is not desirable for our purpose. Instead, we want to prune outdated data about the build execution time from the dataset.

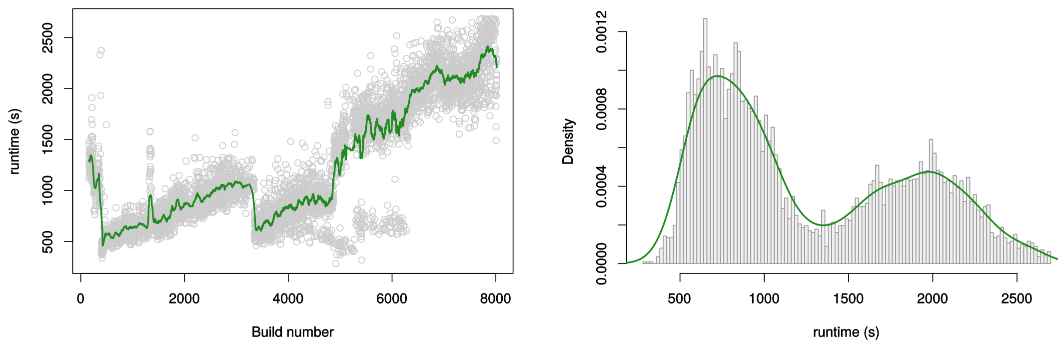


Figure 5.5: Time series data of passed builds from Spring Boot

Given a specific time window, the runtime duration distribution now resembles a normal distribution. Figure 5.6 shows the same information as the previous plots, only in a window of 1100 passed builds from the later stages of the project. From this distribution we can determine that 50% of builds that pass do so after 2120 seconds.

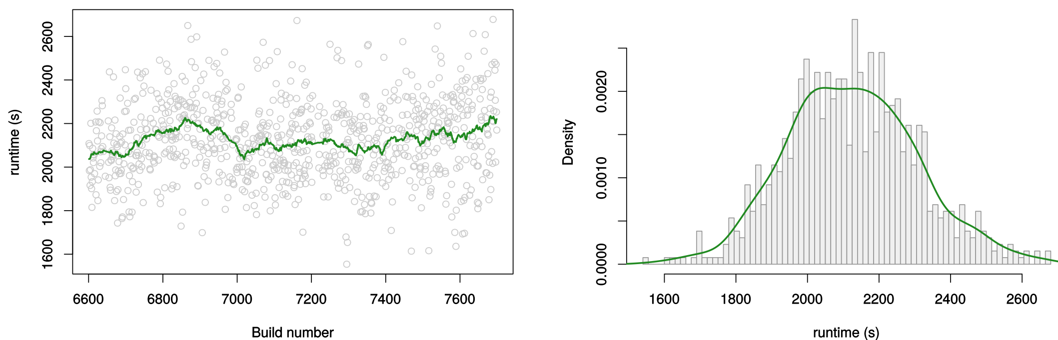


Figure 5.6: Windowed time series data of passed builds from Spring Boot

Failed or errored builds exhibit similar behavior, but require a more careful interpretation. We now examine the temporal aspects of different error types in more detail.

5.2.3 Frequency of Occurrence of Error Types

As we have established in Section 5.1, there exists a multiplicity of error types. A build runs in different phases, and each phase may produce its own specific kinds of errors. Consequently, error types that belong to different execution phases also have different frequency of occurrence accumulation points. For example, compilation errors will naturally occur during the compilation phase, which is executed before the unit-testing

phase. Our goal is to describe this behavior for individual error categories. This will allow us to calculate the likelihood that a specific error can occur at a given instant during the build execution.

To gain an understanding of the data, we first explore the basic build result categories provided by Travis-CI. We then proceed to repeat the analysis with the error types generated by the categorization process.

Initial Results

We initially analyze the frequency of occurrence of the three basic build result categories provided by Travis-CI: *errored*, *failed*, and *passed*. As shown in Section 5.2.2, the average build execution time may change during the course of a project. Using a limiting window over the time series yields a more expressive runtime distribution. For the purpose of this initial experiment, we select the last 10% of the builds for the respective project.

Figure 5.7 shows a summary of the build execution duration of eight different projects. The three boxes per plot indicate the runtime of the respective build result categories.

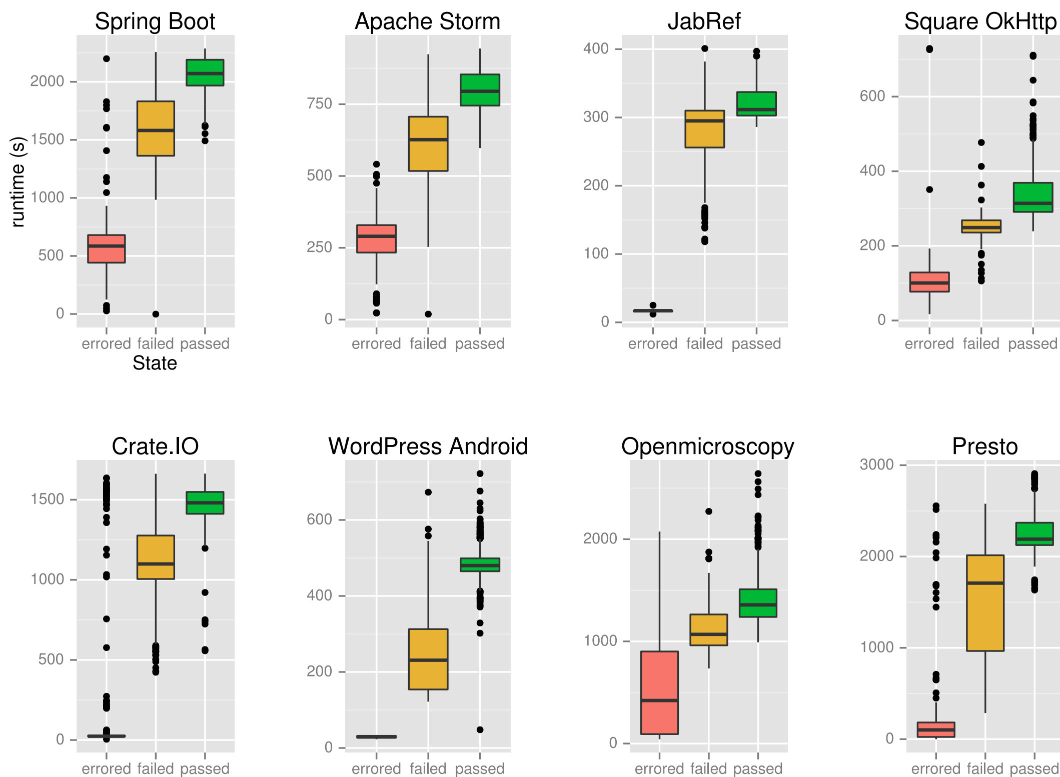


Figure 5.7: Boxplots of the build execution duration of different result categories

We observe that, on average, errored builds terminate before failed builds. In most cases, the runtime distribution of failed builds is wider than that of passed or errored builds. However, errored builds exhibit a higher outlier rate. We also observe that the gap between the medians of failed and passed builds is typically smaller than that of failed and errored builds. We later discuss how this affects our runtime-aware prediction approach (see Section 7.3.3).

These initial results show that with our approach we can locate specific points during the build execution at which errors are likely to occur. Interpreting the temporal dimension of build results in this way allows us to make assumptions about the result of a build during the execution. Given the observed runtime distribution of an error type, we can calculate the likelihood that this particular error has occurred at a specific point in time during the build execution. For example, in the Spring Boot project, if a build has already been executed for 600 seconds, it becomes increasingly unlikely that the build will terminate with the *errored* status (see Figure 5.7).

Detailed Analysis

We now study the runtime behavior of result categories we have gathered through the categorization process (see Section 5.1.2). First, the build data are enriched with the result categories we extracted using LogCat. This allows us to examine the frequency of occurrence of different result categories in more detail. The analysis is again exemplified on the Spring Boot project.

As we have seen, selecting build data from a time window yields a more expressive runtime distribution. For this experiment, we select the last 10% of builds from each error category. Selecting the overall last 10% of builds for the following analyses, without

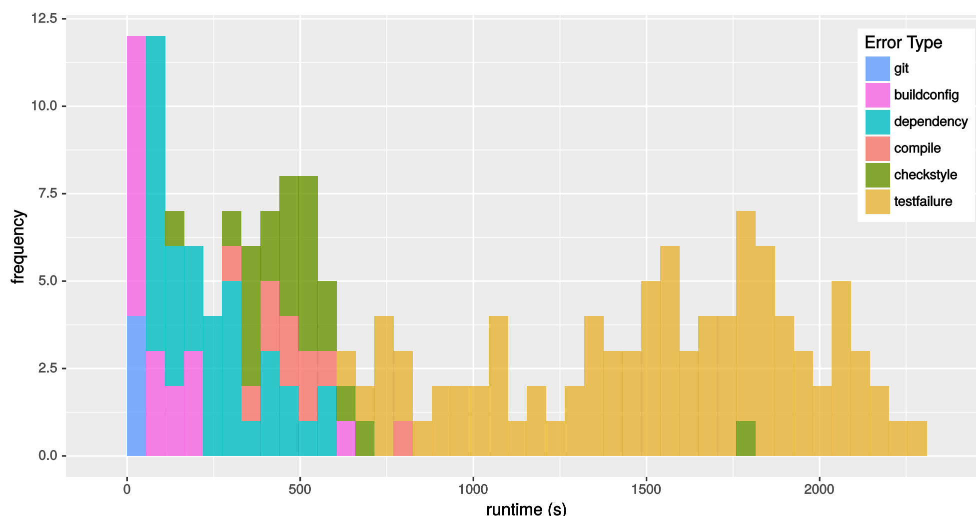


Figure 5.8: Runtime histogram of different error categories from Spring Boot

consideration of the build result, may lead to certain result categories being truncated. For example, if none of the last 10% of builds resulted in a specific error type, the selected subset will not contain data about this particular result category. While this could be due to the root cause for the particular error type having been eliminated, it could also be by chance. As a solution, we perform stratified subsampling by selecting a window over each category individually. This way, builds from all error categories are included, and the runtime distribution will be more representative than using the entire dataset.

We analyze the frequency of error categories extracted from the Spring Boot project (see Table A.2 in Appendix A). Figure 5.8 shows a histogram of the execution duration (with a bin width of 60 seconds) for different error categories. We omitted the categories *other* and *crash* because, as such, they contain both spurious and extreme values that are not purposeful for the illustration.

We want to answer questions of the form: how likely is it that error type e occurs before the given runtime point t. To that end, we examine the probability density of the execution duration of individual error categories. We define the random variable X_e as the execution duration of a build that terminated with the error e. From the observed data, we use kernel density estimation (KDE) [KM14] to estimate the probability density function (PDF) f_{X_e} for each X_e.

Figure 5.9 shows the estimated PDF of two time windows from the sample also used for Figure 5.8. The first plot (left) shows the time window between 0 and 50 seconds. We can see that in this time only the two error types *git* and *buildconfig* occur. The second plot shows the time window between 50 and 640 seconds (before the first *testfailure* occurred). Most builds exceeding the second window are associated with *testfailure* (see Figure 5.8).

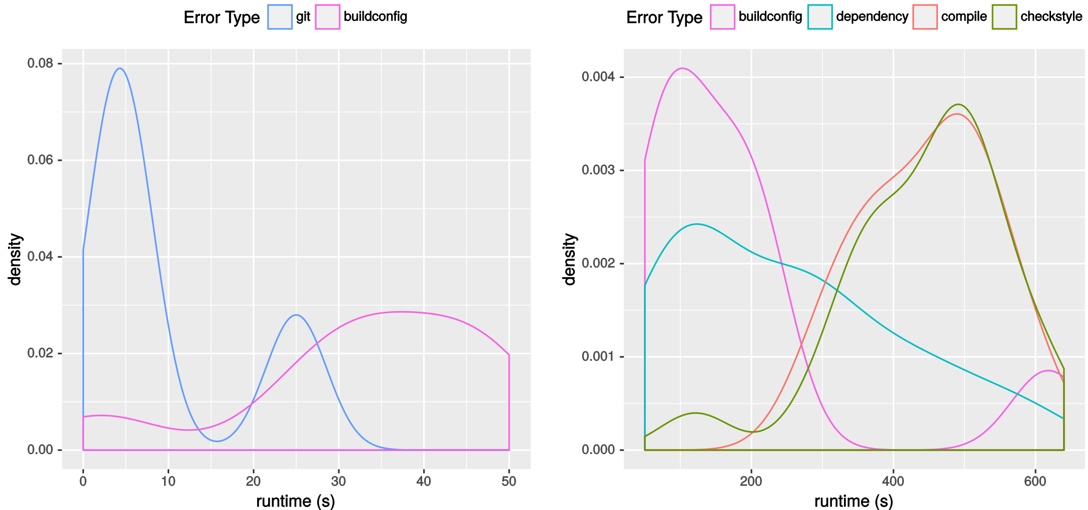


Figure 5.9: Runtime density of different result categories

The cumulative distribution function (CDF) F_{X_e}, i.e., the area under f_{X_e}, allows us to calculate the probability, given a runtime point t, that the error e has occurred: $P(X_e \leq t)$. For example, from $F_{X_{git}}$, we can determine that, if $t = 20$, $F_{X_{git}}(t) \approx 0.77$. In other words, 77% of builds that failed with a *git* error did so after 20 seconds. If we assume that the build will terminate with $e = git$, the tail distribution $\bar{F}_{X_e}(x) = 1 - F_{X_e}(x)$ lets us determine that the remaining probability for the build to fail is ≈ 0.23. As the build execution progresses, the probability for the build to terminate with this error continues to decline.

The CDF of individual error categories are later incorporated in the prediction mechanism we propose (see Section 7.1.3). The prediction that a build is likely to fail with a given error is determined by our classifier. As the build execution progresses, we can update this determination with the probability of individual errors to have occurred.

5.3 Discussion

There are many reasons why builds fail. Seo et al. [SSE+14] uncovered this fact in the context of the edit-compile-test cycle of developers (see Section 3.2). The purpose of this chapter was to provide quantitative and qualitative evidence about the multiplicity of errors types that cause CI builds to fail. We also examined the runtime behavior of these different error types, to identify accumulation points of certain errors during the runtime.

Categorizing build errors is a challenging task where many trade-offs have to be considered. Categories have to be expressive enough to capture error kinds that allow the isolation of influence factors (e.g., finding types of changes that induce specific errors more often than others). However, if the amount of categories (which later serve as dependent variables) is too high, the amount of observations per category will be lower. Consequently, classification will also become more imprecise (see Section 7.3). Finding a balance between the amount of categories, and their expressiveness (i.e., how well the category isolates specific errors) is challenging.

Using our developed LogCat method, we elicited 21 different error kinds by analyzing a total of 54 248 log files. We found that, on average, the most common reasons for builds failures are failing unit-tests, code quality measures being below a certain threshold, and compilation errors. A surprising amount of builds fail because the build worker can not fetch the change data from GitHub. We observed that pull requests are often updated and immediately merged. The pull request update causes a build to be triggered, however when Travis-CI initiates the build worker, the pull request data on GitHub is already gone, causing the build to immediately fail.

We have demonstrate how the build execution duration (runtime) is affected by certain types of build errors, and how the runtime can be used to reason about errors (or vice versa). However, accurately determining the runtime distribution of builds is challenging. The average runtime of builds changes as the project evolves. Events in the project's life time, such as adding or removing tests, or changes in the build configuration, can make

the average runtime increase or decrease drastically. This makes it difficult to precisely reason over the runtime distribution of builds that terminate with different errors.

5.3.1 Limitations

Categorizing build errors is a challenging task where many trade-offs have to be considered. We discuss some of the problems and limitations of our categorization approach.

Parsing Accuracy

Categorizing based on a single message string may not always be accurate. For example, the message "*No output has been received in the last 10 minutes*", indicates that the build failed because it has timed out. Why the build has timed out is not captured by the message. It is possible that a deadlock occurred during the execution of a test, or that the build environment crashed. The category may therefore contain different types of errors. This is a clear limitation of the automatic classification of build errors based on parsing the log files based on a single string.

Runtime Accuracy of Build Phases

Software build tools like Maven support the configuration of multiple modules that may depend on each other. Each module contains its own build sub-configuration and source folder structure. Modules are built in dependency order, i.e., after their respective dependencies. Parts of a build chain (e.g., compilation, code metrics validation) may be executed for an entire module, before the next module is built. This means that runtime accumulation points of phase-specific errors do not have to adhere to the order in the build chain.

Similarly, Gradle can be configured to incrementally compile parts of the source code. For example, integration tests may be compiled *after* unit tests were compiled and executed. When categorizing errors for the purpose of runtime analysis, this type of build configuration poses a dilemma. Assigning each incremental compilation its own category may increase the number of error categories to an undesirable amount. Conversely, by unifying all compilation phase errors into one category, it may become difficult to isolate code changes that lead to this type of error category. For example, changes to test source files may be incorrectly linked with integration-test compilation errors.

CHAPTER 6

Factors Influencing Build Results

A main goal of this thesis is to provide evidence of the relationship between aspects of the development process and build failures. This chapter, we present an empirical study on factors that influence build results. In Section 6.1 we describe the methods used to process the raw data we have gathered. Section 6.2 outlines the different characteristics of build data we will examine. Section 6.3 and 6.4 explain in detail our definition and extraction methods of the change and process characteristics. Section 6.5 we examine the relationship between the defined characteristics and the build outcome. Finally, Section 6.6 discusses the overall results and concludes the chapter.

6.1 Data Processing

Data from version control system (VCS) repositories and continuous integration (CI) systems are gathered to investigate how development practices impact CI build results. To elicit measurable properties of CI builds for the purpose of build failure prediction, these data have to be integrated, structured, and cleaned. Furthermore, various challenges that arise from mining process data from distributed version control system (DVCS) have to be addressed. This section describes the data integration process, and how we deal with missing or incomplete data.

6.1.1 Linking Change and Build Data

Process data in DVCS is naturally graph-based, and the software configuration management (SCM) workflow employed by the team has a large impact on the structure of the version history and change deltas. In contrast, build data from CI servers consist of log files, and a traditional relational data model for build metadata. To generate datasets that can be used to train machine learning models, data from the different heterogeneous sources have to be aggregated and normalized. Specifically, it is necessary to map change

data from the project's source code repository to the CI build data. We call this process *topology mapping.*

Topology Mapping

From previous research, we know that the result of the previous build is a strong predictor for the outcome of a build [HZ06]. In this context, the term *previous* implies that, given a set of builds, there is some ordering function to determine the predecessor of a build. Using the execution timestamp of the build as ordering is insufficient when considering the graph-based nature of the commit history of the VCS, because different branches may be built in parallel. Furthermore, we want to study the effects of code changes on build results, but the build data does not contain change information. These challenges are overcome by mapping build data to the graph structure of the VCS.

Travis-CI build metadata contains some information (e.g., the SHA-1 identifier [CS15], or the author) about the trigger commit, i.e., the tip of the change set that was pushed since the last build. With this information, we can locate the trigger commit in the Git history, and link build data from the CI system with commit data from the VCS. We call this process topology mapping. Figure 6.1 illustrates a branching scenario and how build data is mapped to commits. Green and red elements indicate successful and failed builds, respectively.

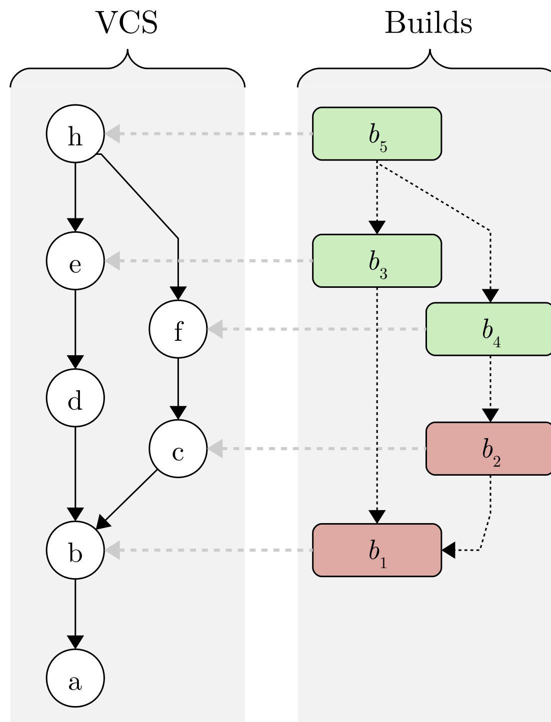


Figure 6.1: Mapping build data to the VCS history topology

The linked build data inherits the graph structure of the Git history, which allows us to do two things in particular: define an ordering of builds based on the VCS history topology (allowing us determine build predecessors), and precisely determine the change set of a build.

6.1.2 Retaining Rewritten History Data

At this point, we rely on historical data of both build and source code repositories. This is problematic in DVCS repositories maintained by teams that employ workflows involving history rewriting (see Section 2.1.4). During early experiments, we realized that we cannot rely solely on historical data available in repositories stored at GitHub. We briefly describe the arising issues and how we addressed them.

Pull Requests When a pull request is updated, GitHub creates a transient merge commit that simulates the merge, and is checked out and built by Travis-CI (see Section 2.3.2). Unless explicitly pulled, these commits are not part of the local history, and are discarded after a certain amount of days on the remote repository. This prevents any detailed analysis of pull requests from historical data. Figure 6.2 shows the topology mapping for pull request builds. Commits g and e are pull-request-merge commits.

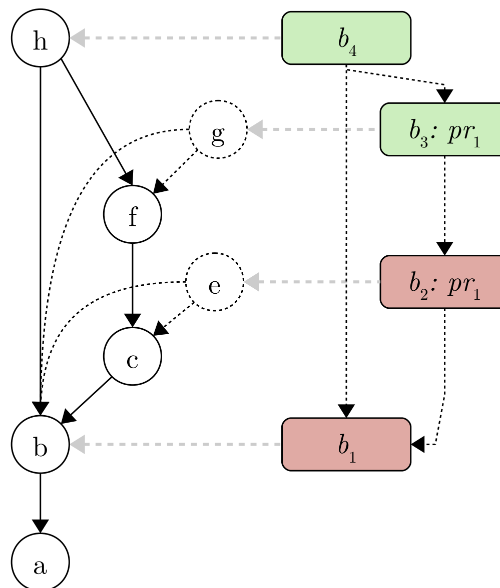


Figure 6.2: Topology mapping of an updated pull request

Rebase Workflows A common practice, when implementing changes suggested in code reviews in the context of a pull request, is to amend the suggested changes to the respective commit, and overwrite the remote history. The updated commits are then built by the CI server, but the commits being amended are lost on the remote repository.

Locally, such rebased commits are stored up to a certain amount of days in the *reflog*[1]. However, unless there is a specific reference that links the changed commits to the tree (e.g., in the form of an annotated tag), these commits are inevitably lost.

Repository Monitoring

When we analyze historical data, and a given pull request has been updated at least once, it is no longer possible to retrieve the changes from the remote repository that triggered the previous builds. This makes analysis of changes that triggered builds in the context of updated pull requests impossible or imprecise. To address this problem, we built a bot (see Section 4.5) that fetches the remote repository immediately after a new build is triggered, and then creates an annotated tag on the commit that triggered the build, thereby maintaining a link to the tree. This way, our local copy of the repository retains, indefinitely, all changes that triggered a build, in contrast to the limited amount of days they remain in the reflog. While this method allows us to precisely analyze changes that were built, it also limits the analysis to the data collected by the bot.

In general, such workarounds for mining DVCS data are unavoidable. The potential loss of historical data has been found to be an inherent aspect of the distributed workflow made possible by Git and GitHub [BRB+09, KGB+14]. This loss of data also has implications on the overall data quality. To perform meaningful analyses, we therefore require additional methods for cleaning the data.

6.1.3 Data Cleansing

As we have established, build and VCS data are both noisy (see Section 5.1) and incomplete (see Section 2.1.4). Limitations in topology mapping on historical data further exacerbates this problem. Next, we give a more detailed explanation of how the data is cleaned of outliers and missing data, to retain a set of observations that allow sound reasoning. The first problem we address is that of *dangling builds*, i.e., builds that can not be associated with change data from the VCS.

Dangling Builds

A consequence of the limitations in topology mapping using historical data (see Section 6.1.2) are *dangling builds*. These cannot be linked to VCS data because the commits do not exist in the history anymore. Changes that triggered these builds can either not be determined accurately, or not at all. Consequently, any metrics involving change or process characterization cannot be calculated on such build data. We introduce a flag *is_in_tree* into our feature vector, that indicates whether a build can be linked to changes in the commit history. This allows us to easily filter relevant observations during data processing. Table B.1 in Appendix B lists the amount of observations in total, the observations after the filtering, and the ratio of the retained data.

[1] `https://git-scm.com/docs/git-reflog` (accessed: 2016-08-10)

Missing Data

Dangling builds have, as such, no change data associated with them. This inhibits many of our conducted tests. For example, it is no longer possible to determine any complexity metrics (see Section 6.3). We mark missing data in our feature vector as follows. All of our measurements are either categorical, or numeric values with values ≥ 0. Values of missing categorical variables are marked with NONE or UNKNOWN. Missing numerical data are assigned the value -1. This allows us to filter data where these values would be relevant for the particular test.

Outliers

Besides missing data, extreme data can also be problematic in statistical analyses. An outlier is a data value that is not representative of the population from which the sample is derived [She03]. Many of our data exhibit high variance values. An example we have already discussed is the build execution duration 5.2. With change and process data we have similar issues. Often, Travis-CI was introduced long after the project's inception. Consequently, the first build will be linked to the entire history up until then. This has an effect on many measurements, including any type of change metric. Some categorical variables also require filtering. Apart from the three build states *failed*, *errored*, and *passed* (see Section 4.4.2), the state of builds may also be labeled *canceled* (build was manually canceled by the user), or *started* (when the build is currently running during data extraction). In general, we only consider builds with the state *failed*, *errored*, or *passed*.

6.2 Factors to Explore

Using the processed, linked, and cleaned dataset, we now wish to explore different measurable properties of the data. This section gives a summary of the factors we explore. In later sections, we give a detailed explanation of how they can be measured and extracted from the data for the purpose of training machine learning models for build result prediction. We distinguish two main categories of measurements (metrics): process and change characteristics. First we explore how well metrics from existing research can be applied to our problem, and subsequently introduce improvements and new metrics that we develop from our observations.

Terminology There is no consistent terminology about different types of metrics throughout research. The term *change metrics* is sometimes used for measurements that qualify changes in the structure of the software system, e.g., moving methods, renaming classes, or changing loop parameters. Our approach does not include such analyses, instead we focus on metrics of VCS commits, e.g., lines of code added or removed, amounts of files changed, or change entropy [Has09] across files. These are typically termed *process metrics*, however we want to distinguish between properties of the development *process* (e.g., measurements of the employed workflows), and properties of

commits (change metrics). We use the following definitions. For our purposes, we use the term change metrics to describe what previous researchers often term “process metrics”. Furthermore, we re-define process metrics to be measurements (typically qualitative) of the development process.

6.2.1 Change Characteristics

A major goal is to understand the impact of changes to the codebase on CI build results. To that end, we need to categorize and quantify characteristics of commits that trigger automated builds, to find predictors for the plurality of build results. Complexity of changes (e.g., size, churn, or entropy), have already been found to impact software quality [Has09, KKA14]. Furthermore, we study the variety of file types in a software system, and how changes to files of a certain type can be linked to specific errors.

We study the following categories of change characteristics:

- **Complexity of changes**

  This category includes size and complexity measurements of the change set. The underlying rationale is that complex change are more error prone [Has09].

- **File types**

  This category aims to describe the intent of the commit by measuring which types of files (e.g. a system, test, or documentation file) were changed, and is inspired by the concept of *change activities* (see Section 3.1.3). Changes made to system-critical files may be more error prone that those made to documentation files.

- **Date and time**

  This category includes measurements of work habits of developers. For example, researchers have suggested that changes made on a Friday night are more error prone [ŚZZ05].

- **Author classification**

  This category aims to describe the different levels of experience of developers. It has been suggested that authors who commit more frequently produce fewer faulty commits [ETL11].

6.2.2 Process Characteristics

Modern SCM tools allow a high degree of freedom when it comes to implementing workflows [GZSD15]. The context of steps within the workflow can give additional meaning to the intent of CI builds. For example, repository hosting services, such as GitHub, provide an additional layer of features on top of the DVCS process: the concept of pull requests (see Section 2.1.6). A build can be executed in the context of a pull request (e.g., when it is updated), or when users with write-access push directly into a

branch. Our goal is to investigate different process measures, and how they affect build results.

We study the following categories of process characteristics:

- **Build types** Builds are executed during different stages of the workflow. This category aims to describe the context in which a build is triggered.
- **Pull request scenarios** The pull requests workflow allow a high degree of freedom in its implementation. This category aims to describe the qualitative characteristics of a pull request.
- **Build history** This category describes characteristics of the build history. For example, the previous build result has been found to be a strong predictor for future failures [HZ06].

6.2.3 Dependent Variables

We use three different types of dependent variables to the describe the outcome of a CI build: A) the binary result type of an automated build, i.e., failed or passed; B) the build result reported by Travis-CI, i.e., errored, failed, or passed; and C) the error kind determined by our build error categorization approach (see Section 5.1). The failed category of variable type A is a union of the builds in the error and failed category of variable type B.

In almost all of the conducted experiments, we study the impact on the build result. In some, we are interested only in the dichotomous nature of the result: failed or passed. This allows us to use binary classification algorithms, such as logistic regression, on the data. A finer grained analysis can be done on the three result states Travis-CI provides. Finally, when we study, e.g., the impact of changes on certain file types, we use the categories determined by the error categorization as dependent variable. The possible values of this categorical variable are therefore determined for each project separately.

6.3 Change Characterization

Capturing the characteristics of a change that leads to a specific build result is essential to building the prediction models we propose. Based on existing research described in Section 3.1 and Section 3.4, we adopt various measures that capture such characteristics. This section describes the measures we later use to create feature vectors for the machine learning step.

6.3.1 Complexity of Changes

In software defect prediction, the most widely used change metrics are: number of revision (NR), number of distinct committers (NDC), number of modified lines (NML),

and number of defects in previous versions (NDPV) [MJ15]. Complexity measurements using entropy functions from information theory have also been used to effectively predict faults [Has09]. The underlying hypothesis is that changes with high complexity are more error prone and will consequently cause more defects.

We adapt these common metrics to fit our data, and define them as follows. Each metric is calculated over the change set C_b of the observed build b (see Section 4.4).

- **Number of commits:** The size of the change set C_b, i.e., the amount of commits that are pushed since the last build.

$$\mathrm{NC}(b) = |C_b| \tag{6.1}$$

- **Number of authors:** The number of distinct authors involved in the change set.

$$\mathrm{NA}(b) = | \bigcup_{c \in C_b} c.\mathrm{author} | \tag{6.2}$$

- **Number of lines added:** The number of lines added across all commits of the change set.

$$\mathrm{NLA}(b) = \sum_{c \in C_b} c.l_+ \tag{6.3}$$

- **Number of lines removed:** The number of lines deleted across all commits of the change set.

$$\mathrm{NLR}(b) = \sum_{c \in C_b} c.l_- \tag{6.4}$$

- **Number of modified files:** The total number of distinct files modified across all commits of the change set.

$$\mathrm{NMF}(b) = | \bigcup_{c \in C_b} F_c | \tag{6.5}$$

- **Change scattering across files:** We adapt the Shannon entropy $H = -\sum_i^k p_i * \log_2(p_i)$ to measure how changes are scattered across files. For a change set C_b, we define $k = \mathrm{NMF}(b)$, and p_i to describe the percentage the i-th file was changed in the change set. We follow the approach by Hassan [Has09] and normalize the entropy value with the total amount of files n in the system at the observed time.

$$\mathrm{CX}(b) = H * \frac{1}{log_2(n)} \tag{6.6}$$

Extraction

From our structured dataset, we retrieve for each observation (build) b the change set $C_b = \{c_1, ..., c_n\}$. Auxiliary functions to extract properties of commits, e.g. $c.l_+$ or c.author, are provided by our data analysis toolkit. These functions ignore changes that involve only whitespaces or moving of files. We now simply calculate the functions described above over C_b, and add them to our feature vector. NC, NA, NLA, NLR, NMF are functions that return integer values, CX returns a real value.

6.3.2 File Types

It is reasonable to assume that changes to specific types of files will lead to some types of errors more frequently than others. For example, it is unlikely that a single change to a plain text resource (e.g., a README file), will lead to a compilation error. It is also unlikely that such a change would fix an already broken build. Similarly, errors regarding the build configuration will typically be caused by changes to a build configuration file.

To study this relation between file and error types, we first need to classify files into their different types. We leverage the project structure convention dictated by Maven[2] or Gradle[3], to identify types of files based on their location in the file tree. Table 6.1 lists the globally defined file type categories and the glob pattern we use to identify them.

Table 6.1: File type categories and associated glob patterns

Category	Glob Patterns
system	`**/src/main/java/*.java`
system_resources	`**/src/main/resources/*`
test	`**/src/test/java/*.java`
test_resources	`**/src/test/resources/*`
webapp	`**/src/main/webapp/*`
benchmark	`**/src/jmh/java/*.java`
build_config	`*pom.xml`, `mvnw`, `mvnw.cmd`, `mvnw.bat`, `*.gradle`, `gradlew`, `gradlew.bat`, `*checkstyle.xml`
git	`.gitmodules`, `.gitattributes`, `.gitignore`, `.mailmap`
ci_config	`.travis.yml`
properties	`*.properties`
documentation	`*.md`, `*.markdown`, `*.txt`, `*.rst`, `*.adoc`, `*NOTICE`, `VERSION`, `README`, `AUTHORS`, `*LICENSE`, `DEVELOPERS`

Projects can customize this structure, or use, e.g., scripts in programming languages that are not covered by the patterns. We consulted the documentation of individual projects for guidance when assigning additional glob patterns to file types. For example, parts

[2] https://maven.apache.org/guides/introduction/introduction-to-the-standard-directory-layout.html (accessed: 2016-08-10)

[3] https://docs.gradle.org/current/userguide/java_plugin.html#N152C8 (accessed: 2016-08-12)

of Apache Storm are written in the Clojure programming language, so we extend the respective patterns with a `.clj` suffix.

To avoid mismatches in the size of the feature vector, we do not add additional categories for projects that would require a rare kind of file type. Instead, we create a category *unknown*, that contains any unclassified files. Also, we do not give special consideration to evolving patterns (e.g., integration tests may be restructured into other modules during the course of the project).

Extraction

To create a feature vector, we calculate the percentage of changes for a file type category within a commit or set of commits. For example, if a system and a test file were changed by adding seven and deleting three lines respectively, the feature vector would contain 0.7 for the *system* category, 0.3 for the *test* category, and 0 for all others. Change amounts are calculated by summing up added and deleted lines. To resolve possible ambiguities (e.g. `.gitignore` files should always be categorized as such, even if they reside in a resource directory), we run the more specific patterns first.

6.3.3 Date and Time

The effects of work habits, such as the time of day or the weekday changes are committed, on software quality have been studied with some conflicting results [ŚZZ05, HZ06, ETL11, KKA14]. Eyolfson et al. [ETL11] found that late night commits (after midnight) are significantly buggier (fault inducing) than commits authored in the morning hours. Kerzazi et al. [KKA14] found that working hours do not significantly impact software build results. In the build failure prediction approach by Hassan and Zhang [HZ06], working hours and weekdays were used in their decision tree approach with some success.

We test whether this controversial hypothesis holds for CI builds. To that end, we calculate the time-of-day (0-23) and day-of-the-week (0-6) for every build. We follow the approach by Eyolfson et al. [ETL11] to account for different time zones of developers.

Time Zone Adjustment

In open source software (OSS) projects, developers from all over the world contribute to the codebase. All dates in the build metadata are recorded in timezone of the build server, i.e., UTC. To reason about the effects of time-of-day of individual developers on the build result, we first need to adjust the build time to the developer's local timezone.

Timezone information of the author is encoded in the commit data. A lookup of the build commit date recorded in the Git history is sufficiently accurate for time of day or weekday reasoning. If the build commit is not in the Git tree, we guess the author's timezone by checking all available historical data, and determine the timezone the author is most frequently working in.

Extraction

From the observed build b, we extract the date of the commit that triggered the build (see Section 4.4). We run our time zone adjustment, and extract the time of day TD $\in [0, 23]$, and the weekday WD $\in [0, 6]$, 0 being Monday.

6.3.4 Author Classification

It has been established that developers have different habits when running software builds and tests in their local development environment [KKA14, SSE$^+$14, BGPZ15]. Eyolfson et al. [ETL11] suggested that developer experience and commit frequency influences commit *bugginess*. Jiang et al. [JTK13] argued that the uneven distribution of coding habits and experience levels among developers should be expressly considered when building defect prediction models.

It is clear that if a developer runs the entire test suite locally before pushing, it is less likely that the CI build will fail. We did not find a way to adequately describe this circumstance using our data. However, because author classification has been successfully employed in the past, we adapt classification approaches (experience and commit frequency) of [ETL11] to fit our data.

We define the author of a build the be the *main* author of the effective change set (see Section 4.4). The main author is the developer with the most commits in the effective change set of the build.

- **Experience**: The time delta between the first time the developer was determined to be an author of a build, and the current observation.
- **Commit frequency**: The authors most occurring time difference between two consecutive commits. We distinguish between daily, weekly, monthly, other (less than 20 commits, but more than 1 commit), and single (a single commit)

Extraction

The experience of a developer is extracted by calculating the time delta between two specific events: the current observation, and the first time the developer authored a build. If the effective change set contains more than one author, we determine the main author to be the person with the most commits in the change set. In the rare case that a change set has the same amount of authors, we break the tie by choosing the developer that authored the latest commit. The time delta is calculated in days and is therefore an integer value.

The commit frequency of a developer is a nominal value as described in the section above. We determine the commit frequency by first calculating the time delta between every consecutive commit by that developer. If the time delta between two commits is less than 30 minutes, the two consecutive commits are considered a single commit. The

time delta δ_t (calculated in days) information is then grouped into four intervals: $\delta_t \leq 1$ (daily), $1 < \delta_t \leq 7$ (weekly), $7 < \delta_t \leq 30$ (monthly), and $\delta_t > 30$ (other). Finally, the most frequently occurring interval denotes the developers commit frequency. We also consider the special case that a developer only has a single commit (single).

6.4 Process Characterization

The employed SCM workflows, particularly branching and pull-based workflows, have been found to impact both software quality and work productivity [SBZ12, VYW$^+$15]. Our goal is to investigate different process measures, specifically those relating to CI practices, and how they affect build results. This section describes the process characteristics we explore, and methods we use to measure them.

6.4.1 Build Types

When changes are pushed to the VCS repository, Travis-CI is notified to check out the changes and execute the build process. Travis-CI distinguishes between *push* and *pull request* triggers, i.e., when changes are pushed directly into a branch, or a pull request is updated. This distinction is important when reasoning about the build outcome, because the build process of these two build types is fundamentally different: when pull requests are updated, a merge into the baseline is simulated by creating a merge commit, which is then built.

Every time a pull request is updated, a merge into the baseline is simulated. The resulting merge commits are not retained in the history because they are only relevant for the build server. Travis-CI labels builds of this type as *pull-request* builds. When pull requests are finalized, an integrator manually merges the pull request into the baseline. The resulting merge commit (which remains in the history) triggers a build that is labeled as a *push* build.

When a developer merges two successfully built branches locally, the resulting merge commit will typically contain no changes. Builds that are triggered by such commits are also labeled as *push* builds by Travis-CI.

To more accurately describe these different types of builds, we extend the build types introduced by Travis-CI, with *merge*, *integration* and *pull-request merge*:

- **Push:** The class of *push* builds retains those builds triggered by regular commits pushed directly into a branch.
- **Merge:** A *merge* build is a *push* build triggered by a merge commit, i.e., a commit with two or more parents. The change set C_b of a merge build contains only the merge commit.

- **Integration**: An *integration* build is a *push* build triggered by a merge commit, but the build change set C_b contains more than one commit (i.e., unbuilt non-merge commits).

- **Pull request:** A *pull request* build is triggered by creating or updating a pull request. It simulates a merge into the baseline.

- **Pull-request merge:** A *pull-request merge* build is triggered when a finalized pull request is merged by an integrator. Such builds can be seen as a special case of *merge* builds.

Determine Merge Commits

Merge builds are triggered by merge commits. There are two ways of determining whether a build was triggered by a merge commit: either looking up the trigger in the Git tree, or examining the commit message.

Checking the commit tree is the easiest and also most accurate method: we simply check if the commit has two parents. As we have discussed, not all commits that triggered a build are retained in the history (see Section 6.1.2). However, Travis-CI stores some metadata about the trigger commit, including the commit message. Git creates a default message for merge commits of the format "*Merge branch 'source' into target*", which is typically not changed. This allows us to determine merge builds directly from their metadata with reasonable accuracy.

A general limitation is that we cannot detect fast-forward merges (see Section 2.1.3), and there is, to the best of our knowledge, no data about how often branches are updated via manual fast-forwards merges.

Determine the Merge Commit of a Pull Request

GitHub's pull request mechanism facilitates different workflows. Per default, branches are merged with the '--no-ff' option, which forces the creation a merge commit with a default message of the format "*Merge pull request #<pr-nr> [...]*". This convention allows us to easily write a parser that determines the commit that eventually merged a specific pull request into the baseline.

Pull requests can also be merged by squashing all commits of the branch into a single commit, with a new message, and then performing a rebase (see Section 2.1.6). Figure 6.3 illustrates how a commit history changes through this process. Because this method leaves no apparent reference to the pull request in the history, it is effectively impossible to precisely determine the commit that merged the respective pull request from the historical build or VCS data. An exception is a pull request that contained only a single commit, in which case there is nothing to squash and the commit SHA can therefore be found in the target branch.

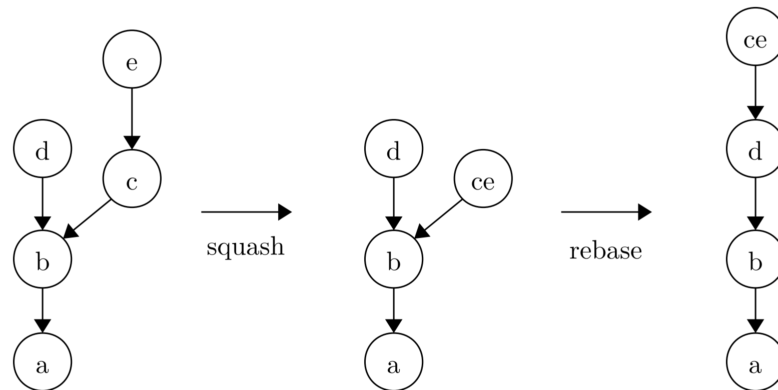


Figure 6.3: Pull request merge squashing

Extraction

We extract this information into the categorical variable BUILD_TYPE ∈ {PUSH, PR, MERGE, INTEGRATION, PR_MERGE, UNKNOWN}. We first determine the event type (*push* or *pull request*) from the Travis-CI build metadata. If possible, we check the commits in the Git history, as it is more reliable. As a fallback, we use the commit information supplied in the build metadata.

6.4.2 Pull Request Scenarios

Pull-request-based workflows on hosted repository services have recently become subject of intense research [VYW+15, GZSD15, YWF+15]. Git and GitHub provide a variety of ways to update pull requests, and we examine how differences in this workflow affect build results. To that end, we first explore different update scenarios. Figure 6.4 illustrates an example of a simple update scenario. The left tree shows a pull request scenario where e is the pull-request-merge commit that simulates the merging of c into d. The right tree shows the same pull request updated with f, where the baseline has not advanced.

We create a taxonomy of pull request update scenarios by defining a set of four predicates:

1. **Pulling into foundations:** The target of the merge is equal to the merge base of the source and target. This means there are has been no parallel development that would have to be considered when merging.
2. **The PR has advanced:** Changes have been introduced to the PR since the last build. The source of the previous build is therefore different from the source of the current build.
3. **Upstream has advanced:** Since the last build, changes were made in the branch being merged into. The target of the previous build is therefore different from the target of the current build.

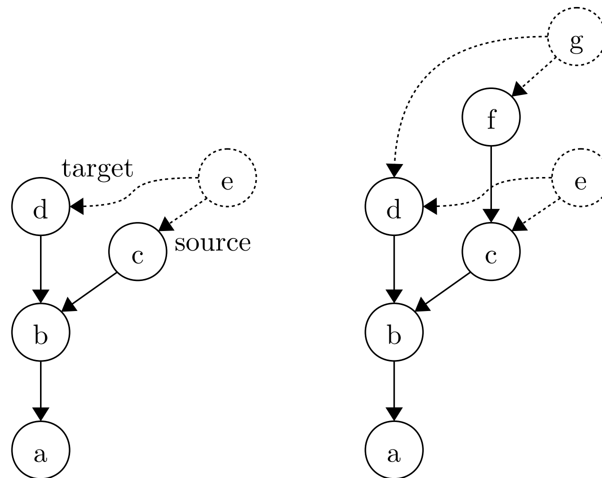


Figure 6.4: Example of a pull-request-update scenario

4. **The PR was rebased:** A special case of 2., which states that the branch of the history of the PR was somehow re-written, either via rebase, amend or similar commands. This means that the previous build has commits that are not reachable from the current build.

Extraction

We create five variables to accurately describe the pull request scenario. A categorical variable stores the pull request scenario: PR_SCEN ∈ {A,...,P,SIMPLE_1, SIMPLE_2, NONE}. For the categorization, we create a truth table (see Table 6.2) with all four predicates, and enumerate the update scenarios from A to P. For the first build of a pull request we can determine only predicate 1. We label these builds with SIMPLE_1 or SIMPLE_2. The other four variables hold the truth value for each predicate respectively. This allows a more detailed analysis.

Scenarios marked with a * are implausible. For example, predicate 4. implies 2., therefore scenarios E, G, M, and O are not possible. The pull request scenario is determined exclusively for PR builds (see Section 6.4.1), any others are marked with NONE. If the build has no previous build, the update scenario cannot be accurately determined, and is therefore also marked with NONE.

6.4.3 Build History

Hassan and Zhang [HZ06] considered three types of measurements of the build history: days since the last build failure, number of passed builds prior to a given build, and the previous build result. They found that these factors outperform all others when training decision trees to predict build failures. Based on these insights, we adapt the metrics to our data, and define them as follows:

Table 6.2: Truth table for determining the pull request scenario

Label	Predicates 1.	2.	3.	4.
A	T	T	T	T
B	T	T	T	F
C	T	T	F	T
D	T	T	F	F
E*	T	F	T	T
F*	T	F	T	F
G*	T	F	F	T
H	T	F	F	F
I	F	T	T	T
J	F	T	T	F
K	F	T	F	T
L	F	T	F	F
M*	F	F	T	T
N	F	F	T	F
O*	F	F	F	T
P	F	F	F	F
SIMPLE_1	T	-	-	-
SIMPLE_2	F	-	-	-

- **Previous build result:** The result of the previous build(s). As we have discussed, when mapping builds to the VCS history structure, a build may not have a single distinct predecessor (see Section 6.1.1). Because most builds have either one or two predecessors, we use two fields: `prev_left` and `prev_right`, indicating the previous build result in the left or right most subtree respectively. For example, in the scenario shown in Figure 6.1, build b_5 has two predecessors, where b_3 is of the left most, and b_4 is the right most subtree. This distinction is particularly interesting for merge commits: merging a working branch into a broken branch may indicate an attempt fix the broken build.

- **Climate:** The build climate, or build stability, is a moving ratio window over the fail rate. We measure the fail ratio of the last k builds.

- **Days since last fail:** This variable describes the amount of days that have passed since the last build failure. In our analysis, we later create time intervals to transform the variable into a categorical type.

Extraction

Through the process of topology mapping, the build history inherits the graph structure of the VCS history. This allows us to use simple graph search algorithms, such as breadth first search (BFS), on the structured build data. The predecessors of a build are determined by traversing each branch until a build is found. When we determine k predecessors of a builds, e.g., for calculating the climate, we traverse the graph using

BFS from right to left, until all branches have been traversed and Fat least k builds have been found. The resulting set of builds is then ordered by their occurrence in the topology, and the first k builds are selected.

Pull Request Builds Pull requests are built differently than normal push builds (see Section 6.1.2). When a pull request is updated, a merge commit is created that simulates a merge into the baseline. Pull-request-merge commits are not reachable through the ancestry path of a later commit. Consider the example in Figure 6.2. The previous build of b_3 is b_2. As we traverse the graph from node g to find the previous build of b_3, a normal graph traversal would find commit b, and therefore b_1. By *shadowing* the build link to the source of the pull request (in this example, the source of e is commit c), we can proceed to use our graph traversal approach to find the previous commit of a pull-request-build. When a pull request is rebased, this approach also fails to find the previous pull request build. In this case, we simply look up the previous build from the list of pull-request builds as `prev_right`.

6.5 Statistical Analyses

We extract from the structured build and VCS data all previously described measures. Different statistical methods, such as Pearson's chi-square test, or the Mann–Whitney U test, are then used to examine the influence of the different factors on the build outcome. This section presents our findings, and we first give more detailed explanation of our methodology.

6.5.1 Methodology

Determining causal relationships of build failures from our data set is very difficult and would involve elaborate testing with numerous control factors. Because the understanding of such relationships is important for assisting the software quality assurance process, it has been attempted by researches to draw conclusions about causation from tests similar to ours. However, we believe that accurately describing causes of build failures is ultimately not necessary for building a strong prediction system [Shm10]. While our study is motivated by hypotheses proposed in previous research, we examine solely statistical correlation, and draw no conclusions about causality. The goal is to gain a deeper understanding of why certain measures work well as features for build failure prediction models. Hence, when we speak about influence, we do this from a purely statistical point of view.

For each variable, we study the correlation of the developed measures and build outcome. All our experiments require nonparametric statistical test, i.e., tests that make no assumption about the probability distribution of the examined variables. We select the tests based the decision procedure described by Sheskin [She03]. The selected tests are in accord with the approach of Kerzazi et al. [KKA14], who also studied factors that can be linked to build automation failures.

Categorical variables (such as build types or pull request scenarios), are examined using Pearson's chi-square test. Additionally, we calculate Cramér's V (ϕ_c) to determine the effect size of each test. ϕ_c is a measure of association, and is used together with the chi-squared test to interpret the strength of a relationship. For continuous measures (such as the complexity of changes), we use the two-sample Wilcoxon rank sum test, also known as the Mann–Whitney U test (from hereon called the Mann–Whitney test). Unlike the chi-square test, the Mann–Whitney test requires the dependent variable to have exactly two levels, i.e., to be a binary variable. Both of the employed test methods are executed using the R statistical computing system [KM14].

Using these tests, we perform standard significance testing of null hypotheses. A null hypothesis states that there is no relationship between two sets of observations. For every statistical significance test, we determine the p-value of the test, and evaluate the null hypothesis using the common cutoff value $\alpha = 0.05$. If $p < \alpha$, we reject the null hypothesis and conclude that there is a statistical significant relationship between the samples. For the Pearson's chi-square test, we also calculate the effect size $\phi_c \in [0, 1]$ to determine the strength of the relation. Higher ϕ_c values indicate a stronger relationship.

In each of the following sections we test a specific type of factors we have defined in Section 6.2.

6.5.2 Complexity of Changes

The complexity of changes have been used in numerous studies on software defect [MJ15] and build failure analysis [HZ06, CH11] alike. We calculate these metrics from the effective change set (see Section 4.4.1) of a build. Consequently, change complexity metrics can only be calculated on build data that have been successfully linked to change data with topology mapping (see Section 6.1.3).

Our dataset on these metrics is highly skewed, as made evident by Table B.4 in Appendix B. We therefore trim the dataset of extreme outliers by dropping observations where NMF (number of modified files) is above the 99th percentile (individually for each project). This effectively eliminates rare events, such as the first build which contains all changes committed since the repository was initialized. An exception is made for the project RxAndroid, on which very little change data is available. Table B.6 in Appendix B lists mean values of all change complexity metrics per build outcome and project after filtering.

For each metric we run the Mann–Whitney test against the binary build result (failed or passed). Table 6.3 lists the resulting p-values for each metric and project. Highlighted values are $p < 0.05$.

As we have established, previous researchers have linked change complexity metrics to build outcome and software defects. The experiments we perform do not give conclusive evidence that this can be applied in the same way to CI builds. For example, Hassan and Zhang [HZ06] argued that, integrating a large number of files will lead to build

Table 6.3: Results of the Mann–Whitney test of change complexity metrics

	NC	NA	NLA	NLR	NMF	CX
Apache Storm	< **0.001**	< **0.001**	0.437	0.208	0.216	0.743
Crate.IO	0.391	0.222	**0.003**	0.208	0.211	0.209
JabRef	**0.001**	0.360	< **0.001**	< **0.001**	< **0.001**	< **0.001**
Butterknife	**0.004**	0.537	0.162	0.169	0.105	0.460
jcabi-github	< **0.001**	**0.040**	**0.024**	**0.011**	< **0.001**	< **0.001**
Hystrix	0.063	0.696	0.434	0.458	0.445	0.897
Openmicroscopy	**0.016**	**0.035**	0.927	0.475	**0.016**	0.484
Presto	< **0.001**	**0.004**	< **0.001**	< **0.001**	< **0.001**	< **0.001**
RxAndroid	0.491	0.771	0.352	0.663	0.755	0.961
SpongeAPI	**0.019**	0.139	< **0.001**	**0.005**	**0.007**	**0.003**
Spring Boot	0.101	**0.041**	< **0.001**	**0.002**	< **0.001**	< **0.001**
Square OkHttp	0.255	0.988	0.220	**0.009**	**0.012**	0.212
Square Retrofit	0.746	0.485	**0.017**	0.161	0.107	0.316
WordPress Android	0.555	0.172	< **0.001**	< **0.001**	< **0.001**	< **0.001**

failures more often. In eight of the studied projects, the number of modified files has a statistical significant influence on the build result. However, there are cases in which the average number of modified files for passed builds is *larger* than that of failed builds. This unintuitive phenomenon holds for many of our observations.

6.5.3 File Types

It is reasonable to assume that changes to specific types of files will lead to some errors more frequently than others. We test this hypothesis by classifying each file into our defined file types (see Section 6.3.2). In this scenario, there are multiple continuous independent variables, i.e., one for each file type. Testing multiple independent variables against a nominal variable would require advanced statistical methods. Instead, we design the experiments so we can continue to employ our methodological approach. We are interested in what types of changes induce which result types.

The data is prepared as follows: we create categories based on whether a specific file type was changed or not (indicated by the file type name). For example, when a change set includes changes to a system and a test file (and nothing else), the build is labeled with *system+test*. To contain the amount of categories, when a change set contains changes to more than three different file types, we label it *tangled* [HZ13]. The resulting list is sorted by the amount of builds per category, and the top n categories selected. Per category we count the amount of builds in the specific state (errored, failed, or passed). Table 6.4 shows the result of our data preparation approach for the Spring Boot Project with $n = 6$.

The chi-squared test is now conducted on these contingency tables. We select $n = 6$. We observed that for higher n values, the amount of observations would generally drop too far

Table 6.4: Contingency table of file type changes and build results for Spring Boot

	errored	failed	passed
test+system	139	101	608
system	140	86	576
build_config	101	74	530
documentation	90	70	417
tangled	58	52	205
test	45	39	190

to produce representative results. Table B.5 in Appendix B lists all project contingency tables. The cell values correspond to ratio between errored, failed, and passed builds for the respective file type change. Table 6.5 lists the result of the chi-squared test.

Table 6.5: Results of the χ^2 test on file types

	p	ϕ_c
Apache Storm	< **0.001**	0.142
Crate.IO	< **0.001**	0.075
JabRef	< **0.001**	0.155
Butterknife	**0.002**	0.223
jcabi-github	**0.002**	0.182
Hystrix	0.156	0.128
Openmicroscopy	< **0.001**	0.086
Presto	< **0.001**	0.079
RxAndroid	**0.039**	0.272
SpongeAPI	**0.006**	0.088
Spring Boot	0.110	0.047
Square OkHttp	< **0.001**	0.117
Square Retrofit	0.261	0.109
WordPress Android	< **0.001**	0.061

The low amount of data for the projects RxAndroid, and Square Retrofit, make the results for these projects inconclusive. We observe that in 10 out of 14 cases, the p-value is below the cutoff point, with a moderate to low effect size. Although the values indicate a relationship between file type changes and the build outcome, it is unclear whether changes to a specific file type leads to errors more frequently.

Table B.5 in Appendix B shows some paradoxical results. For example, in the Spring Boot project, there are 577 builds in which only documentation files were changed. We can see that these builds have the same error rates as the builds where test and system files were changed. This can be explained, at least partially, by two facts. First, our test does not control for builds that were already previously in a broken state. It is reasonable to assume that a change to a documentation file will not fix a broken build. Therefore, any builds that previously fail, will most likely fail again. Secondly, we observed several

spurious errors in the Spring Boot project. Some tests may randomly fail, even when effectively harmless changes are made. We conducted an experiment where we considered only *original failures*, i.e., failed builds preceded by a successful build. From the 577 builds where only a documentation file was changed, a total of 40 caused original failures. 21 of these failures were caused by failing tests, 18 by test environment crashes, and one because of a dependency error. All of these errors can be, as such, spurious.

6.5.4 Date and Time

The time-of-day and weekday changes are made, have been hypothesized to have a significant influence on software quality [ETL11]. A popular theory is that changes made on Friday evening (before developers leave work for the weekend) are more likely to be of low quality [ŚZZ05]. We calculate the time-of-day and weekday of the build trigger of build b (see Section 6.3.3).

The resulting values are interpreted as nominal values, with time-of-day $\in \{0, ..., 23\}$ and weekday $\in \{0, ..., 6\}$. On these values, we run the chi-squared test against the binary build outcome. Table 6.6 shows the results of the test.

Table 6.6: Results of the χ^2 test on time of day, and weekday measures

	Time of Day		Weekday	
	p	ϕ_c	p	ϕ_c
Apache Storm	0.175	0.087	0.322	0.042
Crate.IO	< **0.001**	0.068	< **0.001**	0.061
JabRef	**0.045**	0.078	< **0.001**	0.069
Butterknife	< **0.001**	0.238	< **0.001**	0.167
jcabi-github	**0.030**	0.190	0.221	0.089
Hystrix	0.065	0.180	0.082	0.104
Openmicroscopy	< **0.001**	0.100	**0.010**	0.035
Presto	0.173	0.047	**0.007**	0.037
RxAndroid	< **0.001**	0.379	< **0.001**	0.232
SpongeAPI	< **0.001**	0.089	< **0.001**	0.075
Spring Boot	**0.045**	0.074	0.117	0.040
Square OkHttp	< **0.001**	0.110	< **0.001**	0.103
Square Retrofit	**0.004**	0.134	< **0.001**	0.167
WordPress Android	< **0.001**	0.074	< **0.001**	0.050

We observe that in 10 out of 14 cases the chi-squared test on the time-of-day value reported a significant influence, with a confidence interval $p \leq 0.05$. However, the results show predominantly a small to medium effect size, indicated by $\phi_c < .21$. Similarly, the weekday value also has a significant influence in 10 out of 14 cases. The effect size is generally smaller than in the time-of-day experiment.

In conclusion, we do not have enough evidence to make a sound claim about the general effect of time-of-day on the build outcome. However, the results indicate that in some

cases (for example the RxAndroid project), the metrics could have a positive effect on the performance of a prediction model.

6.5.5 Author Classification

Contributors in OSS projects have different development habits and experience levels. We classify authors in two dimensions: experience level and commit frequency. We test our author classification approach against the binary build outcome. To that end, the Mann–Whitney test is used on the developer experience, and the chi-squared test on the commit frequency.

We consider only data that contain author information. This is roughly 40% of our dataset, and an average of 58% per project.

Developer Experience

In this first experiment, we test the developer experience value against the binary build result outcome using the Mann–Whitney test. Table 6.7 shows the resulting p-values, and the mean experience values of passed and failed builds, respectively.

Table 6.7: Results of the Mann–Whitney test on developer experience

	p	$\bar{x_p}$	$\bar{x_f}$
Apache Storm	$<$ **0.001**	200	233
Crate.IO	$<$ **0.001**	386	333
JabRef	0.782	467	459
Butterknife	$<$ **0.001**	676	176
jcabi-github	**0.047**	212	263
Hystrix	$<$ **0.001**	255	332
Openmicroscopy	0.861	590	619
Presto	0.661	424	403
RxAndroid	0.341	227	181
SpongeAPI	**0.001**	260	310
Spring Boot	$<$ **0.001**	466	407
Square OkHttp	$<$ **0.001**	720	545
Square Retrofit	$<$ **0.001**	625	284
WordPress Android	$<$ **0.001**	512	422

We observe that in 10 out of 14 cases, the Mann–Whitney test reported a p-value < 0.05. However, we also observe five cases in which the mean experience of passed builds is lower than that of failed builds. Although there appears to be a correlation between developer experience and build failures, we can not provide sufficient evidence for the claim that a high experience level inevitably leads to less failed builds.

Commit Frequency

In this second experiment we test the defined commit frequency intervals (single, daily, weekly, monthly, other) against the binary build outcome using the chi-squared test (see Section 6.3.4). Table B.7 in Appendix B lists failed and passed builds per project and commit frequency intervals. The table shows that some intervals occur rarely in some projects. For example, the Crate.IO project has no authors that commit consistently on a monthly basis. For the purpose of the chi-squared test, intervals with a low amount of data are removed. Table B.8 in Appendix B lists the failure ratio per project and commit frequency interval. Table 6.8 shows the results of the test.

Table 6.8: Results of the χ^2 test on commit frequency

	p	ϕ_c
Apache Storm	$<$ **0.001**	0.109
Crate.IO	$<$ **0.001**	0.073
JabRef	**0.002**	0.073
Butterknife	$<$ **0.001**	0.279
jcabi-github	$<$ **0.001**	0.190
Hystrix	**0.017**	0.149
Openmicroscopy	0.090	0.042
Presto	**0.003**	0.064
RxAndroid	0.519	0.077
SpongeAPI	$<$ **0.001**	0.173
Spring Boot	$<$ **0.001**	0.063
Square OkHttp	$<$ **0.001**	0.083
Square Retrofit	$<$ **0.001**	0.143
WordPress Android	0.186	0.020

We observe that in 11 out of 14 cases the chi-squared test returned a p-value < 0.05. However, the effect size is predominantly small. Overall the commit frequency appears to have only a moderate influence on the build outcome. From Table B.8 in Appendix B, we can also not conclusively determine whether, e.g., daily committers produce less build failures.

6.5.6 Build Types

We categorize builds into: merge, integration, push, pull-request, or pull-request-merge. Figure 6.5 shows the distribution of the different build types among projects.

As we have established earlier, GitHub allows two basic workflows for merging pull requests: the default merge-commit approach, and *merge squashing*. With merge squashing, we cannot detect, from our historic data, when a pull request is merged. Instead, these builds will be counted in the push category. Some projects in our dataset (e.g., Apache Storm, Crate.IO, or Presto) employ the merge-squashing workflow, as made evident by the distribution of build types, i.e., the absence of pull-request-merge builds. The project

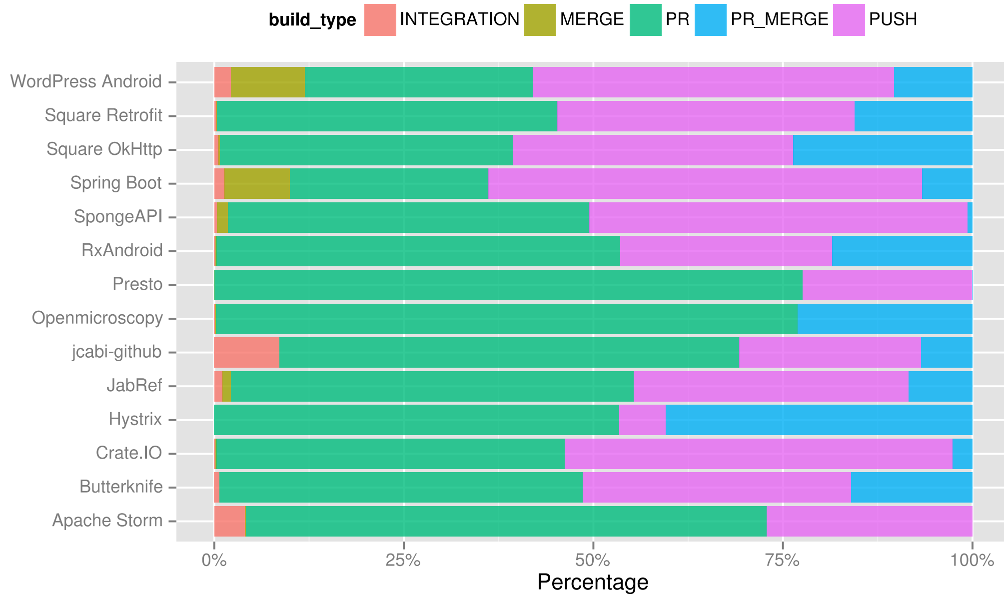


Figure 6.5: Distribution of build types

Openmicroscopy has a very strict contribution policy: changes are integrated almost exclusively through pull requests. This may be the case for other projects, e.g., Presto, but because of the limitation in our build type detection approach, we cannot detect this in our data. The disparity of workflows among projects makes it difficult to compare them.

We now investigate whether different build types have a significant influence on the build outcome. The separation of merge builds is essential because they have a fundamentally different build characteristic and would falsify any conclusions. Consequently, this investigation cannot be done properly on projects that employ merge squashing, because pull-request-merge builds are mixed into the *push* category. For the purpose of the chi-squared test, rare build events are removed. For example, the RxAndroid project has only a single merge build. Per project, we filter those types where the overall amount of builds of that type make up less than 1% of all builds.

We run two experiments using the chi-squared test. The first tests all build types against the binary build outcome. The second test considers only push and pull request builds. Table 6.9 shows the p-values and the effect size of the chi-squared test.

We observe that in all eight cases the build type has a significant influence on the build result, with a moderate effect size.

Next, we examine the effect of individual build types. Table 6.10 list the failure ratio of individual build types. For example, in the JabRef project, only 6% of pull request

Table 6.9: Results of the χ^2 test on build types

	Overall		Push vs. PR	
	p	ϕ_c	p	ϕ_c
JabRef	$<$ **0.001**	0.148	$<$ **0.001**	0.080
Butterknife	$<$ **0.001**	0.214	**0.020**	0.079
jcabi-github	$<$ **0.001**	0.177	**0.022**	0.081
RxAndroid	$<$ **0.001**	0.148	**0.003**	0.129
Spring Boot	$<$ **0.001**	0.075	**0.004**	0.035
Square OkHttp	$<$ **0.001**	0.125	**0.038**	0.029
Square Retrofit	$<$ **0.001**	0.183	0.066	0.037
WordPress Android	$<$ **0.001**	0.139	$<$ **0.001**	0.104

merge builds failed, compared to 27% of pull request builds. This is consistent across all projects. Results on whether or not pull request builds fail more often push builds, or vice versa, across projects, remain inconclusive.

Table 6.10: Failure ratio of different build types

	(1)	(2)	(3)	(4)	(5)
JabRef	0.17	0.28	0.27	0.20	0.06
Butterknife	-	-	0.36	0.44	0.14
jcabi-github	0.32	-	0.50	0.40	0.19
RxAndroid	-	-	0.14	0.24	0.10
Spring Boot	0.32	0.25	0.33	0.30	0.20
Square OkHttp	-	-	0.53	0.56	0.40
Square Retrofit	-	-	0.22	0.26	0.04
WordPress Android	0.14	0.17	0.12	0.19	0.04

(1) INTEGRATION, (2) MERGE, (3) PR, (4) PUSH, (5) PR_MERGE

6.5.7 Pull Request Scenarios

We now examine whether the pull request scenario defined by our taxonomy, or differences in individual predicates that define the taxonomy, have a significant influence on the build outcome. We chose again for our experiment a confidence level of 95% ($p \leq 0.05$). The dependent variable used in the test is the binary failed/passed build outcome.

Analyzing pull request types is highly reliant on data we have gathered through the repository monitoring approach. Because we started monitoring repositories roughly 80 days before the study, we have very little pull request scenario data on projects that run builds less frequently. Specifically, the projects Butterknife, Hystrix, jcabi-github, RxAndroid, and Square Retrofit, have less than 50 records with pull request scenario information. This low amount of data will make any statistical analyses on this categorical variable imprecise, and we therefore omit the listed projects from the analyses. Overall,

only 6.2% of pull request observations contain data about the pull request scenario (see Table B.2). Furthermore, although scenarios H and N are plausible, they practically never occur. The total occurrences in our dataset are 2 and 0 respectively (see Table B.2). In fact, there are only two records in which predicate 2. (the PR has advanced) is not satisfied (meaning that a rebuild of a pull request is almost never manually triggered). We omit the scenarios H and N by completely removing them as values from the categorical variable. Predicate 2. is also omitted in our experiments.

Table 6.11 gives an overview over the test results. Column two and three show the result of testing all scenario types observed within the project. The last six columns show the results of testing the respective predicate (True or False). Highlighted values are $p < 0.05$.

Table 6.11: Results of the χ^2 test on pull request scenarios and their predicates

	Scenarios		Predicates					
			1.		3.		4.	
	p	ϕ_c	p	ϕ_c	p	ϕ_c	p	ϕ_c
Apache Storm	0.456	0.19	0.859	0.01	1.000	0.00	1.000	0.00
Crate.IO	0.513	0.15	0.421	0.04	1.000	0.00	1.000	0.00
JabRef	**0.003**	0.19	0.211	0.05	**0.014**	0.10	0.340	0.04
Openmicroscopy	**0.001**	0.26	**0.013**	0.12	0.094	0.09	0.123	0.08
Presto	**0.012**	0.20	**0.003**	0.13	1.000	0.00	1.000	0.00
SpongeAPI	**0.023**	0.45	**0.043**	0.21	1.000	0.00	0.064	0.22
Spring Boot	0.344	0.37	0.728	0.04	0.761	0.06	0.956	0.01
Square OkHttp	0.403	0.29	0.419	0.08	0.296	0.15	0.588	0.08
WordPress Andr.	**0.002**	0.27	$<$ **0.001**	0.18	0.395	0.05	0.210	0.08

We observe that in five out of nine cases, the pull request scenario has a significant influence on the build outcome, with a medium effect size in most cases. In four cases, the first predicate also has a significant influence, with a lower effect size. We intuitively assumed that pull requests that underwent a rebase (predicate 4) would lead to build failures more often, due to the complexity of the rebase process. Our data does not support this hypothesis, as made evident by the results of predicate 4.

6.5.8 Build History

Measurements of the build history, such as the outcome of the previous build, have been found to be strong predictors for build results [HZ06]. As we will show, our data supports this to a high degree.

Previous Build Result

Through the process of topology mapping, the build history inherits the graph structure of the VCS history (see Section 6.1.1). Consequently, we are no longer able to define a distinct predecessor of a build. In Figure 6.1, build b_5 has two predecessors, namely b_3

and b_4. In terms of the topology, we say that b_3 is the *left*, and b_4 is the *right* predecessor. This distinction allows us to examine the effects of previous errors in the context of merge commits. There are scenarios where a build has more than two predecessors. However, in our dataset, these scenarios make up only 1.5% of observations. In these cases, we search for the left-most and right-most predecessor. We map the results of the previous builds to the categorical variables *prev_left* and *prev_right*. Both can take the values *failed*, *errored*, or *passed*.

We initially run three experiments that test the influence of the previous build outcome on the build outcome. The build outcome is measured as *passed*, *errored*, or *failed*. The first experiment uses observations with one previous build ($n = 1$), the second and third uses those with $n >= 2$ previous builds. Table 6.12 shows an example of three contingency tables on which we executed the chi-square test.

Table 6.12: Contingency tables for the previous-build test from the Spring Boot project

	$n = 1$			$n \geq 2$					
					left			right	
outcome	e	f	p	e	f	p	e	f	p
errored	520	36	252	56	3	53	61	4	47
failed	33	250	264	2	19	31	9	7	36
passed	226	258	3058	17	25	332	111	9	254

(p = passed, e = errored, f = failed)

We observed that, for 12 out of 14 projects in our dataset, the result of the chi-square test was $p < 0.0001$ for every experiment, and average effect size ϕ_c of 0.43 for experiment one, 0.41 for experiment two, and 0.34 for experiment three. The exceptions were projects RxAndroid and SpongeAPI, for which the chi-square test did not give conclusive results. This is due to the low amount of data for these particular projects (see Table B.3 in Appendix B). The results for the other projects indicate that the previous build result, whether one or more previous builds, is a major influence on the build outcome.

By taking a closer look at the individual build outcomes for builds with $n = 1$ predecessors, we can see that build outcomes mostly follow the outcome of their predecessor. For example, a failed build will much more likely follow a failed build than an errored build. Figure 6.6 shows this phenomenon on eight of the projects in our dataset (selected by the amount of available data). Red, orange, and green colors indicate errored, failed, and passed results respectively

Climate

The build climate, or build stability, is a small-windowed moving ratio over the previous build failures. For each build, we calculate the failure ratio of the last k builds. For our experiments, we use $k = 10$. The Mann–Whitney test is used to test the climate against the binary build outcome *failed* or *passed*.

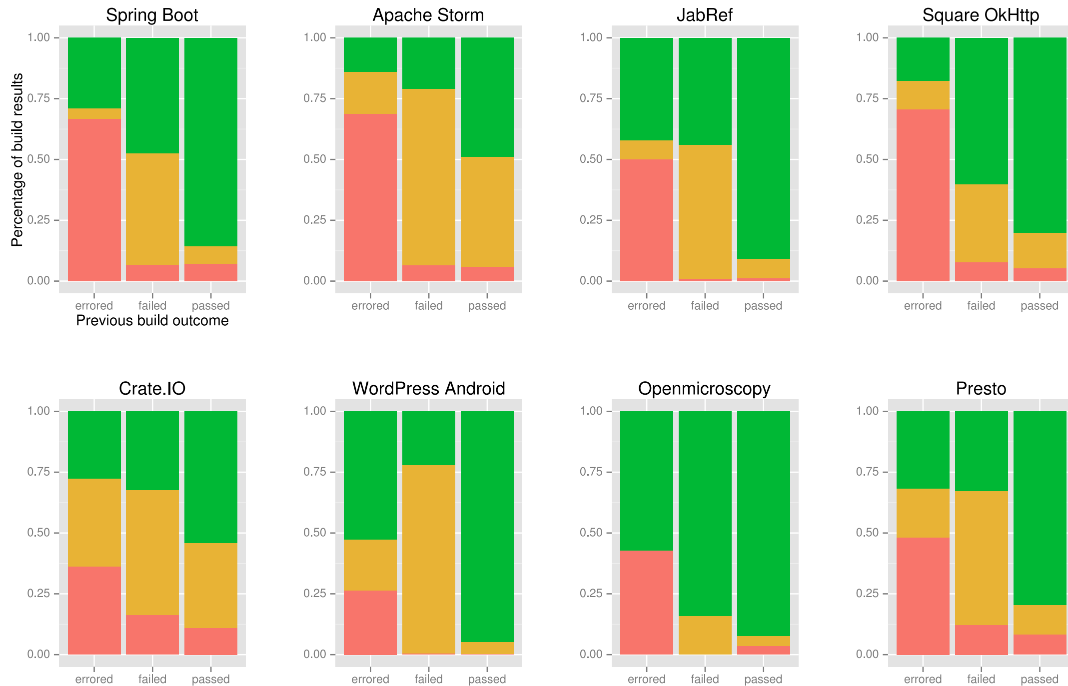


Figure 6.6: Previous build results and the percentage of build outcomes

The Mann–Whitney test revealed a p-value < 0.0001 for every project in our dataset, showing that the build climate of failed and passed builds are non-identical populations. Table 6.13 lists the mean climate values for the entire project, passed, and failed builds. We observe that, in every project, there are periods of high stability or instability. Builds pass more frequently if the previous builds also passed.

It is apparent that build failures often occur consecutively. A high build climate indicates that there is a period of build instability which has to be addressed. Conversely, a low build climate also means significantly more future passed builds. We conclude that the build climate is a major contributor to build failures.

Table 6.13: Mean build climate ($k = 10$) values of projects

	Overall	failed	passed
Apache Storm	0.660	0.904	0.169
Crate.IO	0.704	0.924	0.147
JabRef	0.199	0.716	0.048
Butterknife	0.358	0.965	0.027
jcabi-github	0.415	0.755	0.123
Hystrix	0.491	0.829	0.152
Openmicroscopy	0.161	0.867	0.026
Presto	0.501	0.918	0.069
RxAndroid	0.173	0.880	0.036
SpongeAPI	0.249	0.912	0.021
Spring Boot	0.312	0.715	0.136
Square OkHttp	0.477	0.880	0.107
Square Retrofit	0.207	0.960	0.013
WordPress Android	0.116	0.589	0.031

Days Since Since Last Failure

This experiment is conducted using the chi-squared test. To that end, we create four different time intervals to group the data. The variable t denotes the days since the last failure.

1. $t = 0$: Less than a day ago
2. $t = 1$: A day ago
3. $1 < t \leq 7$: A week ago
4. $t > 7$: More than a week ago

Table B.9 in Appendix B lists, per project, the amount of failed and passed builds for each interval. Table 6.14 lists the result of the chi-squared test.

Table 6.14: Results of the χ^2 test on days since last failure

	p	ϕ_c
Apache Storm	0.374	0.046
Crate.IO	< **0.001**	0.152
JabRef	< **0.001**	0.239
jcabi-github	0.149	0.101
Hystrix	< **0.001**	0.242
Openmicroscopy	< **0.001**	0.259
Presto	< **0.001**	0.223
SpongeAPI	< **0.001**	0.244
Spring Boot	< **0.001**	0.254
Square Retrofit	< **0.001**	0.470
WordPress Android	< **0.001**	0.427

Because of low amounts of data, the results of the chi-squared test were inconclusive for the projects Butterknife, RxAndroid, and Square Retrofit. We observe that in 9 out of 11 cases the p-value is < 0.001 with a medium to large effect size. As the results make apparent, in almost all projects there are phases of instability. During theses phases, the amount of failed builds increase. We conclude that the time since the last failure is a major contributor to build failures.

6.6 Discussion

The purpose of this chapter was to identify factors that influence build failures, and provide evidence about the strength of such relationships. We have examined seven different aspects and a total of 17 different variables of build and VCS data. Some of these variables, such as change complexity, or the date and time changes were committed,

have been linked to both software defects and build failures in the past [HZ06, ETL11]. Additionally, we have elicited several new measures, such as the pull request scenario, to examine the relationship between aspects of the development process and build failures.

In general, our results strongly favor process over change metrics. The analyses how that process metrics, i.e., measurements of the development processes, have a stronger correlation with the build outcome than change metrics. Among all variables, the strongest predictor for build failures is the outcome of the previous build. This shows that build errors often occur consecutively.

We have also observed some paradoxical phenomena. For example, it has been shown that the different experience levels of developers have a significant influence on software quality [ETL11]. It is intuitive to assume that a high degree of experience leads to a higher code quality and therefore less problems. We could not provide conclusive evidence that developers that commit daily or have a high experience level, produce less build failures. A possible interpretation is that daily committers are more careless when pushing changes. It is also possible that the limitations of our approach inhibit the detailed analysis of such hypotheses. We now discuss these limitations.

6.6.1 Limitations

Data Quality

During early experiments in the study, we uncovered the problem of missing historic data (see Section 6.1.2). Table B.1 in Appendix B shows that on average only 28% of build data can be linked to change data. We addressed the problem by creating a monitoring system that fetches the changes from the VCS repository as they are made. This approach does however not allow us to *recover* lost historic data. Up until the point we started the monitoring process, our dataset may have gaps of missing data. How this exactly affects the analyses is however unclear. A different aspect of data quality is *noise*.

Noise

Data obtained through techniques of mining software repositories (MSR) have been found to be inherently noisy [KZWG11]. There are different types of noise in our data that have an influence on the test results. For example, badly written test cases, or tests that cover unstable parts of the system can fail randomly. Change data is associated with these failures regardless. This is particularly problematic, when changes that can not affect the build process are associated with build failures (we given a example below). This introduces noise that distorts our analysis. As we have discussed in Section 6.5.3, some paradoxical results can be explained by looking at these data.

Changes that can not affect the build process should not be associated with build failures. For example, some projects maintain a changelog file, i.e., a text file that lists major changes introduced by a change set. After a successful integration, an additional commit is made to update the changelog. Such a commit should clearly not cause a build failure.

Yet, in the Hystrix project, 34% of builds in which only changelog entries were added, introduced original failures. Conversely, changes made exclusively to test classes only introduced original errors in 25% of cases.

Such spurious data has problematic effects. The performance of machine learning models will ultimately suffer due to skewed statistics. Also, it becomes hard or impossible to draw sound conclusions about causal relationships between calculated metrics and build failures.

Interpretation of Process Metrics

Process metrics can be deceptive, and their meaning heavily dependent on the project's workflow. We give an example of how a project's workflow can impact the meaning of a process measure such as the build type.

We have defined integration builds to be those triggered by a merge commit and where the build change set C_b contains additional commits (see Section 6.4.1). Unlike integration builds, merge builds do not have the risk of additional changes that may not have been tested locally. It is tempting to immediately reason that integration builds fail more often than merge builds. However, such reasoning can be erroneous. For example, in the Apache Storm projects, when a pull request is merged, the changelog is typically updated to reflect the change. Because this is done manually by an integrator, by our approach the build is then considered an integration build.

We conclude that causal reasoning based on interpretations of process metrics should be done with care.

CHAPTER 7

Predicting Build Failures

In the last two chapters we have examined what errors cause continuous integration (CI) builds fail, and what factors can be associated with the build outcome. In this chapter, we present how we use the previously developed methods and gained insights, to predict the outcome of a CI build. Section 7.1 gives an overview of the general approach. Section 7.2 presents details of the experiment design and employed machine learning methods. To determine the effectiveness of our approach, we employ common statistical model validation techniques. The results of this performance evaluation are presented in Section 7.3. Finally, we discuss the results and limitations of our approach in Section 7.4.

7.1 Approach

In order to examine the feasibility of build failure prediction, we conducted an empirical study on the data we have used in previous chapters. The factors we have examined in Chapter 6 are extracted as feature vectors to train different machine learning models. We also devised an approach to update the prediction during the build execution.

7.1.1 Training Features

In the previous chapter we examined a total of 17 different variables of build and version control system (VCS) data. These variables cover two main aspects of build and change data: change and process characteristics. Our analyses suggest that process metrics are stronger predictors for build failures than change metrics. We will substantiate this observation by creating three subsets of features for generating training data. The first set will contain all 17 variables, the second and third will contain variables of change and process metrics, respectively. These features are then extracted from our data to create training input for the selected algorithms.

7.1.2 Machine Learning Algorithms

Different types of machine learning algorithms have been used throughout research for build failure prediction, to a varying degree of success (see Section 3.4.3). There is no conclusive evidence whether a specific algorithm works best for this type of problem. Our approach to update the prediction during the build execution requires non-linear multi-class classification algorithms. We elicited from previous research three well-known training algorithms that satisfy these properties.

- **C4.5 decision trees:** The C4.5 algorithm creates a decision tree where each node checks a specific attribute for a threshold value [FHT01]. The values are selected by the algorithm such that the classified training data are effectively divided by their class.
- **Random Forest:** Random forests are an ensemble learning method that combine multiple decision trees into a single classifier. Given a large enough depth, they can capture complex structures in the data [FHT01].
- **Naive Bayes:** Naive Bayes classifiers are based on the well-known Bayesian theorem. They are often used on data with high dimensionality and have been found to often outperform sophisticated algorithms on such data [FHT01].

We attempted to replicate the decision-tree-construction configuration used by Hassan [HZ06]. The only information we could extract was that they used the C4.5 algorithm with pruning. We set these properties and leave the others at their default values provided by Weka. The same is done for configuring the Random Forest algorithm, previously used by Kerzazi [KKA14]; and the bayesian classifier, used by Wolf et al. [WSDN09]. All algorithms are tested against a 0-R classifier, which predicts the mode of a nominal class.

Our first goal is to examine how well our approach can predict the binary build outcome. We will then test how well the approach can predict the multiplicity of build errors. By predicting individual errors, we can then use our understanding of the temporal dimension of build errors to update the prediction during the build execution.

7.1.3 Runtime-Aware Prediction

A probabilistic classifier, such as Bagging using Random Forest, is capable of predicting the class membership probability of an instance for each class [FHT01]. Rather than computing a unary result (i.e., the predicted class) for an instance, the classifier calculates a probability vector $\mathbf{p} = [p_1 \;\; p_2 \;\; \dots \;\; p_n]$. The number n of components of the vector equals the number of classes. The vector component p_i is the probability of the instance being a member of the i-th class, where $0 \leq p_i \leq 1$. The sum of the vector components is one: $\sum_{i=1}^{n} p_i = 1$. The class with the highest probability value is considered to be the predicted class.

As the build execution progresses, the probability for a specific error to occur declines. For example, if we know that compilation errors occur in the first 20 seconds of the build, and the build has been running for more than 20 seconds, it is unlikely for the build fail due to a compilation error. This temporal aspect of build errors is not considered by the classifier. The predicted class membership probability becomes implausible as the build execution progresses.

We leverage our understanding of the temporal dimension of build errors to update a prediction during the build execution. We use the estimated probability density of the execution duration of individual error categories f_{X_e} (see Section 5.2.3). Given a runtime point t and an error type e, the remaining probability for the error e to occur, is calculated by the tail distribution: $P(X_e > t) = \bar{F}_{X_e}(t) = 1 - \int_0^t f_{X_e}(t)$. The set of predicted classes include the error types and the class *passed*. At a point in time t, we calculate for each class c_i the value of the tail distribution $\bar{F}_{X_{c_i}}(t)$. The result is the vector

$$\mathbf{r}_t = [\bar{F}_{X_{c_1}}(t) \ \dots \ \bar{F}_{X_{c_n}}(t)] \tag{7.1}$$

To that end, we first need to estimate F_{X_e} from our sample data.

Estimating the Cumulative Probability of an Error Type

To construct the cumulative distribution function (CDF) F_{X_c} for the result class c, we first stratify the data by their result class c. Each stratum contains the runtime values of the respective result class, and represents the random variable X_c. For each result class, we estimate F_{X_c} from the stratified sample.

We first construct the empirical distribution function $\hat{F}_{X_c}$, which simply counts the occurrence ratio of samples

$$\hat{F}_{X_c}(t) = \frac{\#\text{samples} < t}{n} \tag{7.2}$$

where $\#\text{samples} < t$ is the number of observations in X_c that are less than t, and n is the total number of samples of X_c. We then construct the piecewise linear function over $\hat{F}_{X_c}$. This creates a nonparametric approximation of the CDF, by linearly interpolating between known values of $\hat{F}_{X_c}$ (known from our sample data). Using the estimated CDF, we can now proceed to update our prediction during the build execution.

Calculating the Runtime-Aware Prediction

To update the prediction $\mathbf{p}$ during the build execution, we proceed as follows. We discretize the runtime into time steps equal to one second. At each second, we update the prediction by computing $\mathbf{r}_t$, and calculating the component-wise product of $\mathbf{p}$ and $\mathbf{r}_t$

$$\mathbf{p} \circ \mathbf{r}_t = [p_1 r_{t_1} \ \dots \ p_n r_{t_n}] \tag{7.3}$$

This continuously reduces membership probabilities of errors that are unlikely to occur from point t. The resulting vector is no longer a probability vector because the sum of its components may be < 1. To compute a probability vector for the updated prediction at point t, we calculate the closure $\mathcal{C}$ [PGETD15] over $\mathbf{p} \circ \mathbf{r}_t$. That is, we divide each component by the vector component sum $s(\mathbf{x}) = \sum_{i=1}^{n} x_i$. This rescales the vector such that the sum of its components is 1.

$$\mathbf{p}_t = \mathcal{C}[\mathbf{p} \circ \mathbf{r}_t] = \left[\frac{p_1 r_{t_1}}{s(\mathbf{p} \circ \mathbf{r}_t)} \dots \frac{p_n r_{t_n}}{s(\mathbf{p} \circ \mathbf{r}_t)} \right] \tag{7.4}$$

The updated prediction $\mathbf{p}_t$ allows us to determine a point in time during the build execution, at which the probability of a build failure is low enough for us to assume that no error will occur. During a build, a developer could decide that a 25% remaining probability for a build failure is low enough to assume that the build will pass. Developers can continue their development as if the build had passed. Given a high enough confidence, the build could even be terminated to save computational resources.

7.2 Experiment Design

We evaluate our approach by conducting several experiments. WEKA[1] is a popular machine learning toolkit that provides a variety of training algorithms for classification task and parameterization of these algorithms. WEKA provides implementations of all algorithms we have selected for our evaluation, and allows the design of experiments using common machine learning model validation techniques.

7.2.1 Testing Features and Algorithms

To test and compare the effectiveness of the features we have developed, we generate multiple datasets for each project. In total, we will have three datasets per project. One for each set of feature groups, i.e, change and process characteristics; and a dataset containing all features.

For each project we run experiments using different datasets and algorithms. Specifically, we test each algorithm we have selected against all datasets. We evaluate each experiment using 10-fold cross validation method and calculate various statistical performance measures, such as the κ-statistic or F_1-score. The result is a matrix that allows us compare the overall performance of features and algorithms. A common approach to test whether the performance scores of two classifiers over multiple datasets are significantly different is to perform a pairwise t-test [Dem06]. This is also the method provided by the WEKA experimenter tool. The t-test allows us to conclude, given the performance scores, whether or not one algorithm outperforms another.

[1] http://www.cs.waikato.ac.nz/~ml/weka/

Using these results, we select the best performing combination of features and algorithms, and examine their classification performance in more detail. Specifically, we will analyze and interpret the confusion matrix of selected experiment results. We are particularly interested in how well failures are detected.

The next step is to test and evaluate our approach to update the prediction during the build execution.

7.2.2 Testing Runtime-Aware Prediction

For this experiment, the K-fold cross-validation approach is impractical. One of the concepts of this evaluation techniques is the randomization of the data within the folds. As we have seen during the examination in Section 5.2.2, the average build execution duration changes during the course of the project. We have also demonstrated that the runtime CDF only produces meaningful results if a specific window over the project lifetime is selected. When sample folds are randomized, the runtime distribution is subsequently distorted, and will not produce meaningful results. We therefore require a sampling and evaluation approach that takes this circumstance into account.

Sampling

As we have discussed, the K-fold cross-validation is impractical, given the nature of our data. Instead, we use the holdout method, where the data are split into two disjoint sets. At a defined point r, the sample data is split. From our observations we know that projects often have a *stabilization* phase when CI is introduced (see Section 5.1.2). In this phase, build errors may occur in an increased amount while the CI configuration is being tuned. To take this into account, we select from our observation dataset with size N, a sample of the last $n = N \cdot q$ data. Training and testing data are taken from this sample, and the size of the training dataset is given by $n \cdot r$. Figure 7.1 illustrates this approach.

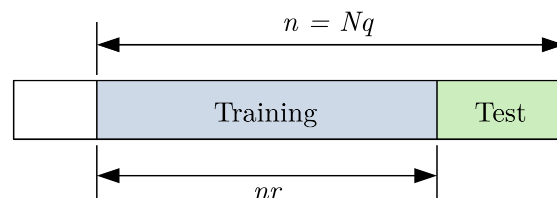


Figure 7.1: Sampling approach for holdout evaluation

We perform stratified sampling as we did during the frequency of occurrence analysis in Section 5.2.3

Evaluation Method

To the best of our knowledge, there are no common validation techniques for the method we propose. Our approach should be able to eliminate false positives, i.e., builds that were actually successful, but were predicted to fail. We examine in detail how the prediction of individual observations changes during the build execution duration, and calculate how the overall prediction performance changes. Also, we calculate how much runtime can be saved by terminating builds that are predicted to be successful with high confidence. The approach is evaluated by examining an exemplary project. We chose Presto because it has high amounts of data, and the test data exhibit a stable runtime distribution.

7.3 Results

We now present the results of our performance evaluation. First, we examined the performance of the binary build outcome classification. We then used similar experiments to test the performance of our multi-class classification approach. Finally, we tested various aspects of the runtime-aware prediction approach.

7.3.1 Binary Result Classification

In this set of experiments we attempt to predict the binary result outcome of a build. Because most previous research also uses the binary build result, this allows us to tentatively compare our approach to others. With these initial results, we can also identify the best performing features and algorithms, to focus our efforts in the later stages.

Our results on binary build outcome prediction show that our approach is generally feasible for predicting the outcome of a build. Table C.1 in Appendix C gives an overview of the results of the pairwise t-test against a 0-R classifier. The compared values are the F_1-scores, calculated from the confusion matrix of the individual prediction results. Except for rare cases, all algorithms perform significantly better than the baseline given by the 0-R classifier. Naive Bayes performed moderately. In over 32 cases, decision trees and random forests performed better than Naive Bayes. The C4.5 decision tree algorithm performed the best overall. However, the performance is only marginally better than that of the random forest algorithm. In three cases, C4.5 performed better than random forest, and in two cases, C4.5 performed worse than random forests. The average F_1-score of both algorithms only differs by 0.01. In terms of feature sets, we see that process metrics consistently outperform change metrics. This substantiates the observations we have made in Chapter 6. Generally, the combination of all features performs about the same as using just process metrics.

We now inspect the results for projects with different levels of classification performance. Specifically, we select three projects to cover a range of medium performing, badly performing, and well performing classifiers. Table 7.1 shows the summary of results from the binary classification for the Spring Boot, Hystrix, and RxAndroid projects. Listed are

the values for precision, recall, the result F_1-score, and the overall amount of observations in the respective class. We observe that in all three cases, the class with less observations

Table 7.1: Summary of binary result classification for different projects

Spring Boot	Precision	Recall	F_1-Score	n
Passed	0.855	0.927	0.890	4 254
Failed	0.757	0.592	0.665	1 636
Weighted Avg.	0.828	0.834	0.827	

Hystrix	Precision	Recall	F_1-Score	n
Passed	0.690	0.683	0.687	287
Failed	0.707	0.714	0.711	308
Weighted Avg.	0.699	0.699	0.699	

RxAndroid	Precision	Recall	F_1-Score	n
Passed	0.945	0.958	0.951	214
Failed	0.559	0.487	0.512	23
Weighted Avg.	0.906	0.911	0.909	

has lower precision and recall values. The weighted average F_1-score of the RxAndroid project is high, because the model has difficulty detecting failed instances, of which very few observations exist. Only 49% of failed builds are detected. With the relatively balanced dataset of the Hystrix project, we observe a better performance in detecting build failures. Although 71% of build failures are detected, the weighted F_1 score is lower because the prediction performance is balanced. These observations indicated that a large factor in the classifier performance is the balance in data of different classes. It is clear that, if there are only few build failures to observe, it is more difficult to fit a model to adequately explain the class of build failures.

7.3.2 Multi-Class Result Classification

Previous research on build failure prediction has focused on the binary outcome of a build: failed or passed. We have uncovered a variety of error types that cause a build to fail (see Section 5.1.2). We have also seen that different errors cause builds to fail at different times during the build execution. Predicting the exact error a build may fail with, allows developers to react faster to possible problems induced by their changes. Using the same three algorithms we used in our previous example, we now present the results of the multi-class classification approach.

The κ-statistic represents a score that can be used to interpret the performance of a

multi-class classifier. However, for the 0-R classifier, the κ-statistic will, as such, always be zero (see Table C.2 in Appendix C). Comparing our prediction approach against the 0-R classifier therefore requires a different metric. For this comparison we chose the root-mean-squared error (RMSE), which is calculated over the probability distribution of each prediction rather than just the confusion matrix results.

Table C.3 in Appendix C shows the results of the pairwise t-test of RMSE values against the 0-R classifier. We observe that in 30 out of 36 cases, random forest outperformed the 0-R classifier. Only a single experiment (change metrics for the Crate.IO project) shows a significantly worse result. The difference in the average error is -0.03, i.e., in favor of random forest. The results for C4.5 decision trees is similar. Naive Bayes exhibits a significantly worse result in 21 cases, with an average error difference of 0.04 in favor of the 0-R classifier. Overall, random forests performed best in this experiment. However, compared to C4.5, the average error difference is marginal at 0.004. In 12 cases random forest show a significant improvement over C4.5, and a significantly worse result in 6 cases.

Table C.2 in Appendix C lists the κ-statistic values of each experiment. We observe that, again, process metrics consistently outperform process metrics in terms of prediction performance. Except for the Apache Storm project, combining all features does not increase the overall prediction performance compared to process metrics.

We next examine the prediction performance for the Spring Boot project. Table 7.2 shows a summary of the prediction performance for individual classes from the Spring Boot project. We observe a similar trend as in the results for the binary build outcome prediction. Classes with less observations are significantly harder to predict. An exception is the *dependency* class, of which 85% of observation were correctly classified. We examined this phenomenon in detail, and found that there is a specific phase during the project, during which this error occurred in a high frequency. Similar observations were made for the *crash* class. The previous-build-result metric is capable of detecting such consecutive occurrences of errors with a high precision.

7.3.3 Runtime-Aware Prediction

We examine how the prediction error rate is affected by updating the prediction during the build execution. To that end, we reduce the multi-class classification result into a binary variable: failed or passed. All builds classified with an error type are counted towards the *failed* category. The class of passed builds remains unchanged. Prediction classes are defined as follows. A *true negative* is a failed build that was detected as such. A *false negative* is an undetected build failure, i.e, failed builds that were labeled as successful. Conversely, a false positive is a successful build that was predicted to fail. We first calculate the number of prediction for each class, given by the classifier. For example, the prediction is a true positive if the actual class is not *passed*, and the predicted class is not *passed*. Using our runtime aware approach, we check whether the prediction changed

Table 7.2: Summary of multi-class prediction performance for the Spring Boot project

Class	Precision	Recall	F_1-score	n
passed	0.841	0.953	0.894	4254
dependency	0.927	0.858	0.891	310
buildconfig	0.796	0.667	0.726	123
itestfailure	0.862	0.802	0.831	101
testfailure	0.572	0.268	0.365	593
crash	0.607	0.443	0.513	345
other	0.000	0.000	0.000	2
git	0.000	0.000	0.000	3
compile	0.400	0.073	0.123	55
checkstyle	0.344	0.116	0.173	95
Weighted Avg.	0.791	0.818	0.794	

to *passed* at some point during the execution runtime. If the actual class is an error, and the prediction was never updated to *passed*, then the observation remains a true positive.

Using the process metric dataset of the Presto project, we train a random forest classifier with roughly 3 085 records. The test set comprises 736 observations. The classifier detected 114 true positives, 399 true negatives, 137 false negatives, and 86 false positives. Our runtime aware approach eliminated 79 false positives. The false positives that were detected were mostly classified with errors that typically occur early during the runtime. There were 37 checkstyle errors, 2 compile errors, 1 dependency error, 1 git error, 3 integration test failures, and 35 test failures. We recorded for each observation the point in time t at which the prediction was updated to *passed*. The ratio between the mean runtime of the builds, and the mean of t, gives us an idea of how much runtime resources can be saved by canceling the build once the predicted outcome is *passed*. The mean runtime of the 79 eliminated false positives was 2 221.9. The mean runtime at which the prediction was updated to *passed* was 953.1, giving us a ratio of 0.43.

Figure 7.2 shows how a concrete false positive was eliminated. The build was predicted to fail with a compile error. Lines indicate the probability of a specific result class during the build runtime. The point $t_{\max}$ indicates the point in time where the prediction was updated to *passed*. The points t_p indicate the points in time where the probability for *passed* reached p.

A total of 43 false negatives were introduced. These are builds correctly identified as build failures by the classifier, but subsequently classified as *passed* by our updating approach. 83% of these misclassified builds were test failures. From our study of build errors we know that test failures occur predominantly in the later stages of the build (see Section 5.2). Also, at the point were test failures occur, most other error types are highly improbable. This explains why the additionally introduced false negatives are mostly test failures. To address this, we also recorded the points in time where the updated probability for *passed* reached 50%, 75%, 95%, and 99% (reflecting the commonly used quartiles in statistical distributions).

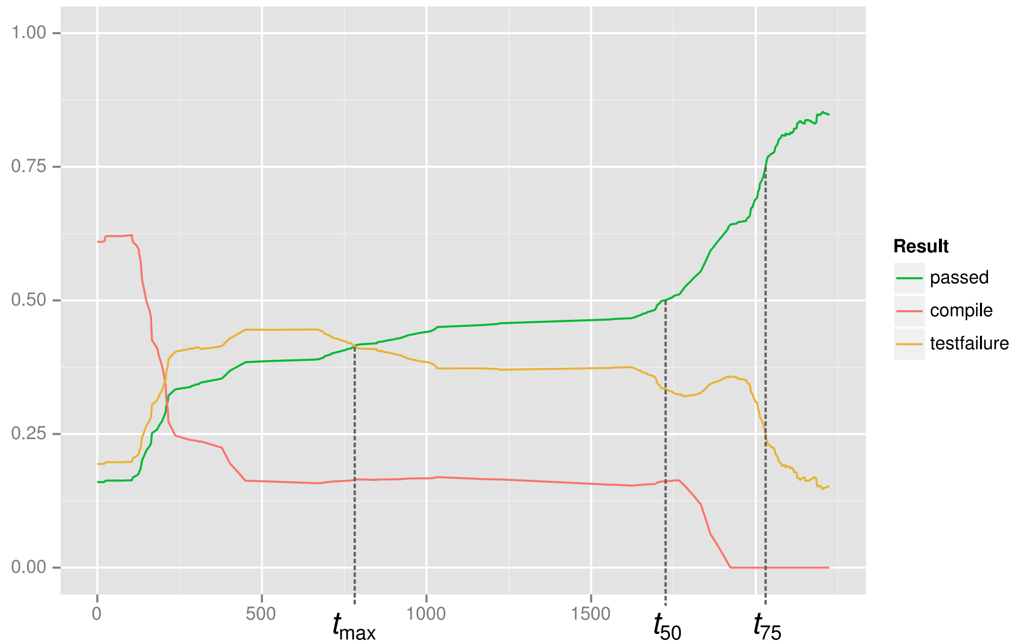


Figure 7.2: Probability of errors during the build execution

As we have seen, the probability for an error declines as the build progresses. It is therefore not surprising that the amount of introduced false positives is reduced if a higher confidence is considered. Table 7.3 lists the amount of introduced false positives that remain after considering the respective probability levels. It also lists the ratio between the mean runtime $\bar{t}$ the respective probability level was reached, and the mean runtime $\bar{r}$ of the builds.

Table 7.3: Number of introduced false positives compared to confidence levels

Probability	False positives	$\bar{t}/\bar{r}$
50%	39	0.46
75%	21	0.71
95%	11	0.92
99%	1	0.93

We observe that the number of false positives is reduced, if one considers a higher probability for a *passed* build. It is clear from the mean runtime ratio that it also takes longer for the probability to be reached. In a practical example a developer could, during the build execution, decide that a 75% probability for a passed build is sufficient to consider the build successful. In this case, the developer could terminate the build to save the remaining 29% of runtime.

7.4 Discussion

In this chapter, we presented our approach for CI build failure prediction. The results from our two-part study on build failures were used to devise methods for a) predicting the binary result outcome of a build, b) predicting the concrete error a build may fail with, and c) updating a prediction during the execution of a build. We performed evaluation experiments using three different algorithms and three different feature sets. The purpose was to determine whether CI build failure prediction is feasible, and how well classification models perform. Additionally, we examined how the temporal aspect of build errors can be used to reason about predictions during the build execution.

Our results indicate that predicting CI build failures from change and process metrics is feasible. We have shown that our approach is significantly better than using a 0-R classifier that predicts the mode (or average) of a class. Decision trees and random forest training algorithms performed similarly well. In terms of feature sets, process metrics consistently outperformed change metrics. Using a combination of all features did not increase the prediction performance compared to process metrics.

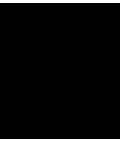
By analyzing the runtime CDF of individual error categories, it becomes possible to reason about a prediction during the build execution. We devised an approach to update the initial probability distribution given by the classifier. We estimated the CDF for individual error categories from our sample data. At every second during the build execution, we update the probability distribution of the prediction using values of the CDF. The updated prediction also includes the likelihood of error occurrences at a specific point in time. With this approach, we were able to reduce the amount of false positives, i.e., successful builds initially predicted to fail. As a trade-off, false negatives, i.e. failed builds that were eventually predicted to be successful, were introduced.

7.4.1 Limitations

Although overall results of the binary classification indicate a high prediction accuracy (average F_1-scores of 0.80), it is clear that an imbalanced data set may distort these performance measurements. Another consequence of imbalanced data is the low classification performance of the class with less observations. Learning on imbalanced data is a challenging task that requires advanced machine learning methods [HG09] to be properly addressed.

The average runtime distribution of builds changes significantly during the evolution of the project (see Section 5.2.2). Consequently, an estimation of the CDF of errors categories from previous observations will become increasingly inaccurate as the project evolves. Accurately estimating robust CDFs, requires advanced techniques of time-series analysis and transformation. For the purpose of our study, we estimated the CDFs from the data set the classifier was tested with.

CHAPTER 8



Conclusion

Build automation is a key aspect of continuous integration (CI), and build failures have a direct negative impact on the development process. Yet, there has been little research on the different aspects and causes of build failures. In this thesis we presented qualitative and quantitative evidence on CI build failures based on publicly available data gathered from 14 different open source software (OSS) project. We have demonstrated that the prediction of CI build failures is feasible. We have also shown that incorporating the runtime behavior of errors allows a prediction to be updated during the execution of a build. Although there are some limitations to our approach, we believe that practitioners can benefit from using our CI build failure prediction method. By using a prediction system, practitioners can react faster to possible problems and save time and resources by making decisions before a build has completed.

8.1 Discussion of Research Questions

In Chapter 1, we introduced four concrete research questions that are concerned with addressing the existing research gaps in the context of CI build failures. We now present concrete answers to these research questions based on the evidence we have gathered throughout this work.

RQ1: Why do CI builds fail? In Chapter 5, we presented our systematic study of build errors that cause builds to fail. A total of 54 248 logfiles of executed CI builds were analyzed to elicit 14 different error kinds. We observed that, on average across all projects, the three most frequent causes of builds failures are 1) failing unit tests (40%), 2) code quality measures being below a certain threshold (11%), and 3) compilation errors (10%). We also found that many builds fail because of faulty interaction between the version control system (VCS) and the CI system. For example, when pull requests are merged before the build worker can check out the change, the build will fail in the

first 5-20 seconds. Another common error is a crash or the exceeding of some time limit of the build worker.

RQ2: What factors can be associated with CI build failures? In Chapter 6, we examined different aspects of build and VCS data. Using statistical methods, we studied the strength of the relationship between these aspects and the build outcome. A statistical significance in almost all variables was found, to a varying degree of effect. In general, our results strongly suggest that process metrics outperform change metrics in terms of build failure prediction. We found that measurements of the development process (e.g., build type or pull request scenario) have a higher correlation with the build outcome than measurements of change data (e.g., size and complexity of the introduced changes). Our results show that errors often occur consecutively, and that, in the majority of cases, the strongest predictor for future build failures is the outcome of the previous builds.

RQ3: How well can statistical models predict the outcome of a CI build? In Chapter 7, we presented our approach to build statistical models for the purpose of predicting build failures. We evaluated our approach using three different multi-class non-linear classification models, and comparing them using common statistical model validation techniques. First, the performance of the binary build result outcome was tested. We found that overall, C4.5 (decision trees) was the best performing algorithm. We also found that process metrics consistently outperformed change metrics. Using C4.5 and training data with process metrics, the generated classifiers showed an average F_1-scores ranging from 0.71 up to 0.98 in extreme cases, with an overall average of 0.84. The performance evaluation of multi-class classification showed that we could determine the concrete error that might occur, reasonably well. Using C4.5 and process metrics, we recorded an average root-mean-squared error (RMSE) of 0.17, compared to 0.21 produced by the 0-R classifier.

RQ4: Can the temporal aspect of build errors be used for prediction? In Chapter 5 we examined not only the multiplicity of build errors, but also the frequency of occurrence of errors during the build execution. We found that we could determine accumulation points of certain errors, and reason about the probability of errors occurring at a given point in time during the build execution. This allowed us to update error probabilities determined by a classifier, and address the issue of declining plausibility of predictions as the build progresses (see Chapter 7). By updating the prediction at each second during the runtime, we found that we could eliminate false positives, i.e., builds that were predicted to fail but were, in fact, successful. However, a trade-off of introducing additional false negatives was also observed.

8.2 Future Work

Although research in the area of build failure detection has begun over a decade ago [HZ06], we believe that there are many more problems to be solved in order to create a robust and accurate build failure prediction system for modern CI environments. This section lists some of the issues this thesis leaves open for future work.

8.2.1 Further Investigation of Factors Influencing Build Results

There are a multitude of factors that influence build results. Even among the factors we have explored, there are additional aspects of causality to be considered. For example, a build can be interpreted in different ways depending on the context it is executed in. Push builds and pull request builds are fundamentally different in their purpose and in the way they are executed (see Section 6.4.1). It is reasonable to assume that the factors influencing the outcome of such builds are also different. Another example are the intents of a change in reaction to a build. Changes made to fix a defect are different from those that cause defects [ŚZZ05]. It is reasonable to assume that this also holds true for build failures. Furthermore, incorporating personalized prediction models, as suggested by Jiang et al. [JTK13], could help capture the uneven distribution of coding habits and experience levels among developers. To fully understand the causal relation between development practices and build failures, these aspects need to be investigated in more detail.

Investigating Developer Behavior

It is reasonable to assume that build results on CI servers are strongly dependent on habitual behavior of individual developers. This has been investigated in the past, and researchers have found that few developers test their software locally, or are aware of CI build failures [KKA14, BGPZ15]. However, some developers may run a full build locally before pushing their changes, more often than others. If a build fails locally, and the developer applies a fix before pushing, information about the failure and subsequent fix is lost. Because the previous build result is a very strong predictor for future failures (see Section 6.5.8), it is clear that a prediction model could greatly benefit from this local build information. Further investigating the influence of developer behavior on CI build failures is therefore a promising research effort.

8.2.2 Improving Data Quality

Transient Failure Detection and Noise

Transient build failures occur in many projects during phases where the build environment is unstable, badly configured or when badly written tests exist with non-deterministic behavior (see Section 6.5.8). Variables related to such failures, when included in the training set, can skew statistics and have a negative impact on the prediction accuracy. For example, it is unlikely that changing a line in a README file leads to test failures.

However such observations exist, and should be considered as noise (see Section 6.5.3). Detecting and cleaning the data set of such noise could increase the performance of a prediction model. How to deal with such noise in the context of software defect prediction has already been investigated [KZWG11], however, to the best of our knowledge, no such efforts have been made in the context of build failure prediction.

Inclusion of Project Meta-Data

We omitted information from project management tools, such as issue trackers, in our approach, because the additional dimension would have increased the overall system complexity beyond the scope of this work. However, many effective approaches in software defect prediction leverage data from issue and bug trackers [RKBD14]. Linking bug and issue data with build and VCS data opens many possibilities for additional metrics to be included in the feature engineering process.

8.2.3 Improving Learning Methods

Imbalance of Data

Our datasets are highly imbalanced in terms of the dependent variable, i.e., the build outcome. Imbalanced data can significantly compromise the performance of standard machine learning algorithms [HG09]. We have not studied to which extent the imbalance in data affects the performance of our models, but it is clear that there is an opportunity to improve overall performance by considering the imbalance of data.

Bootstrapping

The presented approach requires the presence of a large amount of historical data to train a model for each project from ground up. Predicting builds may also be interesting for projects that are just starting off, which do not have such data. There has been research in the area of cross-project defect prediction [ZLXS15], which could be a useful starting point to devise a bootstrapping mechanism. This way, a classifier for a new project could be created based on existing data from projects we know to be similar.

APPENDIX A

Statistical Data on Build Errors

Table A.1: Summary of builds by project and build state category

	errored	failed	passed	Sum
Apache Storm	514	2249	1469	4232
Butterknife	238	132	673	1043
Crate.IO	4584	7612	5232	17428
Hystrix	67	484	567	1118
JabRef	201	1235	4824	6260
jcabi-github	118	384	628	1130
Openmicroscopy	1600	936	12143	14679
Presto	4208	3281	7192	14681
RxAndroid	23	81	560	664
SpongeAPI	235	1795	6140	8170
Spring Boot	1337	1014	5656	8007
Square OkHttp	1909	1536	3393	6838
Square Retrofit	405	189	2288	2882
WordPress Android	151	1721	10620	12492
Sum	15590	22649	61385	99624

Table A.2: Summary of builds by project and build error category

	testfailure	itestfailure	git	crash	quality	compile	dependency
Apache Storm	2008	0	16	59	0	139	176
Butterknife	55	0	25	9	85	21	8
Crate.IO	1697	5333	3293	1077	563	467	154
Hystrix	453	0	35	4	0	17	8
JabRef	685	78	49	12	213	206	4
jcabi-github	263	0	3	9	115	41	17
Presto	3036	298	1619	1008	658	281	324
RxAndroid	19	0	11	0	5	20	0
SpongeAPI	379	0	171	7	31	694	149
Spring Boot	918	118	31	447	210	95	335
Square OkHttp	2430	0	302	2	413	159	96
Square Retrofit	105	0	86	26	211	103	6
Sum	12048	5827	5641	2660	2504	2243	1277

	buildconfig	license	other	doc	androidsdk	incompatibility	buildout	Sum
Apache Storm	106	114	156	0	0	0	0	2774
Butterknife	5	0	0	5	163	0	0	376
Crate.IO	16	0	249	144	0	0	91	13084
Hystrix	10	0	0	0	0	0	0	527
JabRef	53	0	33	0	0	0	0	1333
jcabi-github	0	0	20	0	0	87	0	555
Presto	217	75	50	61	0	0	0	7627
RxAndroid	46	0	0	0	0	0	0	101
SpongeAPI	75	499	4	123	0	0	0	2132
Spring Boot	165	0	18	0	0	0	0	2337
Square OkHttp	11	0	3	0	0	30	0	3446
Square Retrofit	2	0	57	0	0	0	0	596
Sum	706	688	590	333	163	117	91	34888

APPENDIX B

Statistical Data on Factors Influencing Build Results

Table B.1: Result of filtering observations that have linked change data

Project	Total	Filtered	Ratio
Apache Storm	4233	1620	0.28
Crate.IO	17787	4748	0.21
JabRef	6261	3296	0.34
Butterknife	1043	501	0.32
jcabi-github	1130	379	0.25
Hystrix	1118	595	0.35
Openmicroscopy	14726	3874	0.21
Presto	14882	4198	0.22
RxAndroid	665	237	0.26
SpongeAPI	8177	1896	0.19
Spring Boot	8013	5890	0.42
Square OkHttp	6843	3049	0.31
Square Retrofit	2882	1019	0.26
WordPress Android	12509	8555	0.41

Table B.2: Summary of pull request scenarios per project

	A	B	C	D	H	I	J	K	L	N	S1	S2	NONE	Sum
Apache Storm	8	5	13	23	0	14	15	10	11	0	97	40	2496	2732
Butterknife	3	0	6	1	0	0	0	4	0	0	21	3	418	456
Crate.IO	25	0	72	58	0	6	13	10	15	0	122	21	7036	7378
Hystrix	0	0	5	8	0	0	2	0	8	0	21	0	469	513
JabRef	106	21	105	176	0	23	76	13	56	0	107	39	2456	3178
jcabi-github	0	0	0	6	0	0	0	0	0	0	2	0	470	478
Openmicroscopy	17	3	39	65	2	3	72	20	118	0	68	24	10211	10642
Presto	103	1	76	14	0	57	28	48	14	0	112	74	10166	10693
RxAndroid	0	0	2	2	0	0	0	0	0	0	5	0	270	279
SpongeAPI	1	1	13	22	0	4	12	3	18	0	17	3	3598	3692
Spring Boot	1	1	7	6	0	3	6	1	3	0	37	10	1901	1976
Square OkHttp	6	0	22	2	0	10	1	9	1	0	34	14	2379	2478
Square Retrofit	0	0	1	3	0	0	0	1	1	0	11	6	1211	1234
WordPress Andr.	0	53	5	104	0	1	53	1	41	0	108	48	3138	3552
Sum	270	85	366	490	2	121	278	120	286	0	762	282	46219	49281

(S1 = SIMPLE_1, S2 = SIMPLE_2, implausible scenarios were omitted)

Table B.3: Summary of previous build results per project

	p	e	f	p/p	p/e	p/f	e/p	f/p	e/e	e/f	f/f	f/e	Sum
Apache Storm	200	64	352	175	23	131	17	156	34	36	227	23	1438
Crate.IO	1686	355	1714	80	59	95	7	63	34	14	133	76	4316
JabRef	1526	38	221	853	3	264	7	42	1	1	53	2	3011
Butterknife	237	44	5	98	2	4	1	0	5	1	1	0	398
jcabi-github	266	39	54	67	2	11	2	38	1	0	24	1	505
Hystrix	192	4	165	43	0	29	0	26	0	1	41	3	504
Openmicroscopy	408	56	19	2577	172	98	126	92	81	2	11	11	3653
Presto	1909	476	697	210	124	80	6	37	4	4	27	20	3594
RxAndroid	112	0	13	63	0	0	0	3	0	0	2	0	193
SpongeAPI	1352	1	153	208	4	27	0	13	0	0	2	0	1760
Spring Boot	3574	779	544	277	126	13	26	34	47	2	5	8	5435
Square OkHttp	924	248	246	734	54	127	44	164	162	23	80	29	2835
Square Retrofit	534	9	5	315	11	2	2	2	1	0	0	0	881
WordPress Andr.	4149	19	920	2499	24	75	17	226	0	1	97	5	8032
Sum	17069	2132	5108	8199	604	956	255	896	370	85	703	178	36555

(p = passed, e = errored, f = failed. p/e = left passed, right errored)

Table B.5: Contingency tables of common file type changes

Square Retrofit	e	f	p	Builds
test+ system	0.02	0.02	0.96	186
system	0.05	0.05	0.89	152
build_config	0.02	0.03	0.95	60
documentation	0.02	0.00	0.98	51
test+ build_config+ system	0.00	0.04	0.96	48
build_config+ documentation	0.00	0.00	1.00	26

Butterknife	e	f	p	Builds
test+ system	0.07	0.06	0.87	71
documentation	0.35	0.02	0.63	65
system	0.24	0.05	0.71	59
build_config	0.42	0.00	0.58	48
build_config+ documentation	0.38	0.00	0.62	24
test+ build_config+ system	0.33	0.00	0.67	15

Apache Storm	e	f	p	Builds
system+ documentation	0.10	0.54	0.35	345
documentation	0.06	0.60	0.34	225
system	0.15	0.65	0.20	143
test+ system+ documentation	0.09	0.55	0.35	127
build_config	0.16	0.75	0.09	75
test+ system	0.14	0.47	0.40	43

Crate.IO	e	f	p	Builds
test+ system+ documentation	0.18	0.41	0.41	900
test+ system	0.17	0.47	0.37	800
system	0.17	0.42	0.41	497
documentation	0.12	0.46	0.42	411
system+ documentation	0.08	0.47	0.45	383
test	0.11	0.51	0.38	291

JabRef	e	f	p	Builds
system	0.01	0.15	0.84	774
test+ system	0.02	0.26	0.72	436
system_resources	0.00	0.07	0.93	181
test	0.00	0.34	0.66	176
tangled	0.01	0.26	0.72	155
documentation	0.02	0.07	0.92	121

Hystrix	e	f	p	Builds
system	0.01	0.58	0.41	124
test+ system	0.02	0.50	0.48	104
test	0.01	0.53	0.45	77
documentation	0.02	0.53	0.45	66
build_config	0.07	0.43	0.50	44
webapp	0.00	0.32	0.68	25

jcabi-github	e	f	p	Builds
documentation	0.14	0.35	0.51	165
test+ system	0.04	0.24	0.73	102
build_config	0.10	0.30	0.59	69
system	0.13	0.45	0.43	47
test	0.04	0.14	0.82	28
test+ build_config+ system	0.10	0.10	0.80	10

Presto	e	f	p	Builds
system	0.15	0.22	0.63	2267
documentation	0.10	0.23	0.67	359
system+ documentation	0.10	0.28	0.62	268
build_config	0.15	0.31	0.54	264
build_config+ system	0.22	0.25	0.53	237
system+ itest	0.05	0.17	0.78	58

Openmicroscopy	e	f	p	Builds
system	0.07	0.03	0.90	1060
webapp	0.11	0.04	0.85	489
test+ system	0.06	0.03	0.91	176
documentation	0.02	0.05	0.93	126
test	0.10	0.05	0.85	110
itest	0.18	0.02	0.80	90

RxAndroid	e	f	p	Builds
documentation	0.00	0.00	1.00	35
test+ system	0.06	0.00	0.94	35
build_config	0.00	0.16	0.84	25
system	0.00	0.07	0.93	15
properties	0.00	0.00	1.00	13
test	0.00	0.00	1.00	6

Square OkHttp	e	f	p	Builds
test+ system	0.13	0.20	0.66	766
system	0.18	0.16	0.66	348
documentation	0.11	0.17	0.71	167
build_config+ itest	0.00	0.10	0.90	83
build_config	0.26	0.22	0.52	82
test	0.14	0.20	0.66	80

SpongeAPI	e	f	p	Builds
system	0.01	0.12	0.87	1318
test+ system	0.02	0.20	0.78	101
build_config	0.00	0.06	0.94	83
build_config+ system	0.00	0.26	0.74	31
documentation	0.00	0.03	0.97	31
test	0.04	0.21	0.75	28

WordPress Android				
	e	f	p	Builds
system	0.01	0.16	0.84	3119
system_resources+ system	0.00	0.22	0.78	1025
system_resources	0.00	0.20	0.79	550
build_config	0.01	0.15	0.84	330
test+ system	0.03	0.14	0.83	59
build_config+ system	0.00	0.20	0.80	50

Spring Boot				
	e	f	p	Builds
test+ system	0.16	0.12	0.72	848
system	0.17	0.11	0.72	802
build_config	0.14	0.10	0.75	705
documentation	0.16	0.12	0.72	577
tangled	0.18	0.17	0.65	315
test	0.16	0.14	0.69	274

Table B.4: Mean and max values of change complexity metrics per project (before filtering)

	NC		NA		NLA	
	$\bar{x}$	max	$\bar{x}$	max	$\bar{x}$	max
Apache Storm	35.80	3153	3.40	181	4535.84	858497
Crate.IO	2.70	1719	1.08	11	722.81	801973
JabRef	3.32	818	1.05	17	288.15	46663
Butterknife	1.13	6	1.00	2	99.73	3412
jcabi-github	19.54	914	1.52	28	1413.96	70458
Hystrix	2.96	540	1.08	29	342.97	79910
Openmicroscopy	42.62	23333	1.24	59	9724.36	5884058
Presto	2.16	74	1.05	7	443.27	236067
RxAndroid	1.13	7	1.01	2	121.11	1450
SpongeAPI	1.43	27	1.04	4	170.80	7968
Spring Boot	1.35	106	1.06	9	103.94	35607
Square OkHttp	1.09	106	1.01	17	119.20	73756
Square Retrofit	1.19	13	1.02	6	59.34	2518
WordPress Andr.	2.30	4933	1.02	46	278.52	397718

	NLR		NMF		CX	
	$\bar{x}$	max	$\bar{x}$	max	$\bar{x}$	max
Apache Storm	1949.23	857814	94.23	7931	0.14	6.80
Crate.IO	562.72	678499	14.92	8712	0.13	0.95
JabRef	282.87	51947	20.65	4445	0.14	0.83
Butterknife	46.68	1821	3.79	39	0.15	0.77
jcabi-github	259.45	12503	59.95	2862	0.07	0.99
Hystrix	176.48	30328	7.54	883	0.10	0.72
Openmicroscopy	6025.97	4376524	147.02	84668	0.09	0.95
Presto	114.21	19050	12.96	2100	0.15	0.81
RxAndroid	111.58	6257	1.91	44	0.13	1.12
SpongeAPI	77.67	7235	7.19	1533	0.12	0.99
Spring Boot	43.58	35492	8.48	893	0.10	0.78
Square OkHttp	77.22	40094	3.27	527	0.10	0.99
Square Retrofit	38.32	3214	3.53	73	0.13	0.91
WordPress Andr.	120.42	288973	7.31	16119	0.05	0.87

Table B.6: Mean values of change complexity metrics per build outcome and project (after filtering)

	NC			NA			NLA		
	all	f	p	all	f	p	all	f	p
Apache Storm	4.36	5.20	2.76	1.58	1.58	1.58	1649.23	2291.15	427.39
Crate.IO	1.33	1.36	1.30	1.06	1.06	1.05	130.68	143.62	111.93
JabRef	1.73	1.89	1.69	1.02	1.01	1.02	181.64	252.00	163.33
Butterknife	1.12	1.31	1.09	1.00	1.00	1.01	88.54	81.45	89.91
jcabi-github	8.98	1.50	14.31	1.22	1.00	1.37	599.68	34.19	1002.78
Hystrix	1.71	1.69	1.74	1.02	1.02	1.02	135.61	152.90	117.39
Openmicroscopy	5.50	5.12	5.55	1.14	1.15	1.14	223.19	188.85	228.04
Presto	2.00	2.22	1.86	1.05	1.07	1.04	400.10	594.24	273.78
RxAndroid	1.13	1.12	1.13	1.01	1.00	1.01	121.11	173.06	116.15
SpongeAPI	1.34	1.74	1.28	1.04	1.05	1.04	147.45	191.73	141.09
Spring Boot	1.30	1.33	1.29	1.06	1.07	1.05	85.51	108.32	76.74
Square OkHttp	1.04	1.03	1.04	1.00	1.00	1.00	76.47	66.34	81.79
Square Retrofit	1.17	1.11	1.17	1.02	1.00	1.02	53.43	95.59	52.10
WordPress Andr.	1.27	1.28	1.27	0.99	1.00	0.99	52.45	58.73	51.24

	NLR			NMF			CX		
	all	f	p	all	f	p	all	f	p
Apache Storm	1188.33	1689.39	234.61	15.27	18.97	8.25	0.14	0.14	0.12
Crate.IO	74.73	85.85	58.60	6.67	7.05	6.12	0.13	0.13	0.12
JabRef	165.94	234.32	148.14	10.87	13.35	10.22	0.14	0.16	0.13
Butterknife	39.42	39.69	39.36	3.43	3.20	3.47	0.15	0.16	0.15
jcabi-github	115.88	9.18	191.95	26.93	1.99	44.71	0.06	0.03	0.08
Hystrix	87.17	78.08	96.75	4.22	4.07	4.37	0.09	0.09	0.09
Openmicroscopy	340.43	636.32	298.69	9.62	9.26	9.67	0.09	0.09	0.09
Presto	91.77	112.24	78.46	10.12	11.88	8.98	0.14	0.16	0.13
RxAndroid	111.58	127.35	110.08	1.91	1.71	1.93	0.13	0.16	0.13
SpongeAPI	64.16	85.30	61.12	4.47	6.04	4.25	0.12	0.14	0.11
Spring Boot	29.84	30.69	29.51	5.77	6.95	5.31	0.09	0.10	0.09
Square OkHttp	46.94	25.36	58.27	2.69	2.31	2.90	0.10	0.09	0.10
Square Retrofit	32.19	30.67	32.24	3.17	3.22	3.17	0.13	0.15	0.13
WordPress Andr.	31.58	41.63	29.64	2.35	2.88	2.25	0.05	0.06	0.05

Table B.7: Summary of builds per project and author commit frequency

	Daily		Weekly		Monthly		Single		Other		
	f	p	f	p	f	p	f	p	f	p	Sum
Apache Storm	398	137	333	154	1	0	49	41	258	133	1504
Crate.IO	59	230	0	0	0	0	16	56	47	44	452
JabRef	2356	1443	113	101	0	0	23	14	207	198	4455
Butterknife	190	192	6	1	6	2	31	32	54	28	542
jcabi-github	480	1986	10	31	3	16	4	86	109	373	3098
Hystrix	53	125	91	81	0	0	8	11	68	97	534
Openmicroscopy	406	2843	28	212	2	5	0	12	11	151	3670
Presto	879	1324	253	488	26	68	71	142	241	448	3940
RxAndroid	8	80	0	0	0	0	2	40	10	82	222
SpongeAPI	109	656	18	157	13	8	8	107	77	599	1752
Spring Boot	1247	3131	55	134	55	59	96	236	138	375	5526
Square OkHttp	817	1563	34	32	0	0	51	126	100	127	2850
Square Retrofit	14	536	9	69	0	3	9	98	12	158	908
WordPress Android	1256	6400	0	5	0	0	8	30	44	294	8037
Sum	8272	20646	950	1465	106	161	376	1031	1376	3107	37490

Table B.8: Summary of failure ratio per project and author commit frequency

	Daily	Weekly	Monthly	Single	Other
Apache Storm	0.74	0.68	1.00	0.54	0.66
Crate.IO	0.20			0.22	0.52
JabRef	0.62	0.53		0.62	0.51
Butterknife	0.50	0.86	0.75	0.49	0.66
jcabi-github	0.19	0.24	0.16	0.04	0.23
Hystrix	0.30	0.53		0.42	0.41
Openmicroscopy	0.12	0.12	0.29	0.00	0.07
Presto	0.40	0.34	0.28	0.33	0.35
RxAndroid	0.09			0.05	0.11
SpongeAPI	0.14	0.10	0.62	0.07	0.11
Spring Boot	0.28	0.29	0.48	0.29	0.27
Square OkHttp	0.34	0.52		0.29	0.44
Square Retrofit	0.03	0.12	0.00	0.08	0.07
WordPress Android	0.16	0.00		0.21	0.13

Table B.9: Summary of builds in days-since-last-failure intervals

	$t = 0$		$t = 1$		$1 < t \leq 7$		$t > 7$	
	f	p	f	p	f	p	f	p
Apache Storm	637	267	130	69	216	109	50	23
Crate.IO	2038	1091	186	164	382	377	54	91
JabRef	389	846	43	256	129	862	38	388
Butterknife	79	7	3	3	14	14	15	184
jcabi-github	56	56	14	18	52	70	97	163
Hystrix	164	90	26	20	66	78	34	67
Openmicroscopy	203	499	34	185	91	863	117	1677
Presto	1066	1114	108	259	140	441	22	79
RxAndroid	12	12	1	1	0	10	5	96
SpongeAPI	109	287	13	112	24	421	80	718
Spring Boot	1254	1958	110	466	171	852	36	393
Square OkHttp	691	824	108	205	123	407	76	398
Square Retrofit	25	47	0	17	3	82	5	363
WordPress Android	901	1232	87	757	197	2799	115	1959

(p = passed, f = failed)

APPENDIX C

Statistical Data on Build Failure Prediction

Table C.1: Weighted average F_1-score results of binary build outcome prediction

Project	Dataset	(1)	(2)	(3)	(4)
Apache Storm	All	0.54	0.68 ○	0.72 ○	0.71 ○
	Change	0.54	0.48 ●	0.59 ○	0.60 ○
	Process	0.54	0.71 ○	0.73 ○	0.73 ○
Crate.IO	All	0.44	0.44	0.70 ○	0.70 ○
	Change	0.44	0.27 ●	0.59 ○	0.59 ○
	Process	0.44	0.67 ○	0.73 ○	0.71 ○
JabRef	All	0.71	0.82 ○	0.86 ○	0.84 ○
	Change	0.71	0.73	0.74 ○	0.77 ○
	Process	0.71	0.82 ○	0.87 ○	0.87 ○
Butterknife	All	0.62	0.96 ○	0.98 ○	0.97 ○
	Change	0.62	0.69 ○	0.88 ○	0.85 ○
	Process	0.62	0.96 ○	0.98 ○	0.97 ○
jcabi-github	All	0.50	0.69 ○	0.75 ○	0.75 ○
	Change	0.50	0.50	0.66 ○	0.66 ○
	Process	0.50	0.74 ○	0.79 ○	0.77 ○
Hystrix	All	0.35	0.47 ○	0.72 ○	0.69 ○
	Change	0.35	0.38	0.64 ○	0.61 ○
	Process	0.35	0.70 ○	0.71 ○	0.69 ○
Openmicroscopy	All	0.80	0.84 ○	0.86 ○	0.86 ○
	Change	0.80	0.80	0.84 ○	0.83 ○
	Process	0.80	0.85 ○	0.87 ○	0.87 ○
Presto	All	0.47	0.75 ○	0.79 ○	0.78 ○
	Change	0.47	0.57 ○	0.64 ○	0.63 ○
	Process	0.47	0.74 ○	0.80 ○	0.80 ○

RxAndroid	All	0.86	0.85	0.89 ○	0.91 ○
	Change	0.86	0.85	0.86	0.88
	Process	0.86	0.91 ○	0.90 ○	0.92 ○
SpongeAPI	All	0.82	0.86 ○	0.90 ○	0.87 ○
	Change	0.82	0.81	0.83 ○	0.85 ○
	Process	0.82	0.86 ○	0.90 ○	0.90 ○
Spring Boot	All	0.61	0.78 ○	0.81 ○	0.81 ○
	Change	0.61	0.61 ○	0.67 ○	0.67 ○
	Process	0.61	0.79 ○	0.82 ○	0.83 ○
Square OkHttp	All	0.47	0.76 ○	0.80 ○	0.81 ○
	Change	0.47	0.50	0.70 ○	0.69 ○
	Process	0.47	0.77 ○	0.80 ○	0.81 ○
Square Retrofit	All	0.91	0.95 ○	0.96 ○	0.95 ○
	Change	0.91	0.91	0.94 ○	0.93 ○
	Process	0.91	0.95 ○	0.97 ○	0.97 ○
WordPress Android	All	0.77	0.85 ○	0.94 ○	0.94 ○
	Change	0.77	0.77	0.81 ○	0.80 ○
	Process	0.77	0.91 ○	0.94 ○	0.94 ○
Average		0.63	0.74	0.81	0.80

○, ● statistically significant improvement or degradation

(1) 0-R
(2) Naive Bayes
(3) C4.5 Decision Tree
(4) Random Forest

Table C.2: κ-statistic results of multi-class build outcome prediction

Project	Dataset	(1)	(2)	(3)	(4)
Apache Storm	All	0.00	0.13 ○	0.40 ○	0.38 ○
	Change	0.00	0.02 ○	0.13 ○	0.15 ○
	Process	0.00	0.37 ○	0.41 ○	0.37 ○
Crate.IO	All	0.00	0.25 ○	0.31 ○	0.27 ○
	Change	0.00	0.10 ○	0.19 ○	0.14 ○
	Process	0.00	0.27 ○	0.31 ○	0.29 ○
JabRef	All	0.00	0.14 ○	0.48 ○	0.33 ○
	Change	0.00	0.01	0.01	0.13 ○
	Process	0.00	0.35 ○	0.50 ○	0.47 ○
Butterknife	All	0.00	0.79 ○	0.84 ○	0.89 ○
	Change	0.00	0.25 ○	0.62 ○	0.59 ○
	Process	0.00	0.81 ○	0.84 ○	0.89 ○
jcabi-github	All	0.00	0.17 ○	0.44 ○	0.39 ○
	Change	0.00	0.02	0.22 ○	0.22 ○
	Process	0.00	0.42 ○	0.49 ○	0.46 ○
Hystrix	All	0.00	0.11 ○	0.38 ○	0.35 ○
	Change	0.00	0.06 ○	0.27 ○	0.21 ○

	Process	0.00	0.33 ∘	0.38 ∘	0.36 ∘
Presto	All	0.00	0.20 ∘	0.46 ∘	0.40 ∘
	Change	0.00	0.01	0.21 ∘	0.10 ∘
	Process	0.00	0.36 ∘	0.45 ∘	0.45 ∘
RxAndroid	All	0.00	0.54 ∘	0.56 ∘	0.54 ∘
	Change	0.00	0.14	0.02	0.25
	Process	0.00	0.49 ∘	0.62 ∘	0.54 ∘
SpongeAPI	All	0.00	0.22 ∘	0.33 ∘	0.29 ∘
	Change	0.00	-0.01	0.01	0.18 ∘
	Process	0.00	0.28 ∘	0.35 ∘	0.36 ∘
Spring Boot	All	0.00	0.38 ∘	0.53 ∘	0.49 ∘
	Change	0.00	0.03 ∘	0.11 ∘	0.10 ∘
	Process	0.00	0.50 ∘	0.53 ∘	0.52 ∘
Square OkHttp	All	0.00	0.41 ∘	0.55 ∘	0.58 ∘
	Change	0.00	0.13 ∘	0.36 ∘	0.31 ∘
	Process	0.00	0.47 ∘	0.56 ∘	0.58 ∘
Square Retrofit	All	0.00	0.43 ∘	0.59 ∘	0.52 ∘
	Change	0.00	-0.02	0.42 ∘	0.22 ∘
	Process	0.00	0.52 ∘	0.58 ∘	0.61 ∘
	Average	0.00	0.27	0.40	0.39

∘, • statistically significant improvement or degradation

(1) 0-R
(2) Naive Bayes
(3) C4.5 Decision Tree
(4) Random Forest

Table C.3: Root-mean-squared error (RMSE) results of multi-class build outcome prediction

Project	Dataset	(1)	(2)	(3)	(4)
Apache Storm	All	0.27	0.35 •	0.26 ∘	0.24 ∘
	Change	0.27	0.38 •	0.31 •	0.28
	Process	0.27	0.26 ∘	0.24 ∘	0.25 ∘
Crate.IO	All	0.24	0.25 •	0.23	0.23 ∘
	Change	0.24	0.25 •	0.25 •	0.25 •
	Process	0.24	0.24	0.22 ∘	0.22 ∘
JabRef	All	0.17	0.27 •	0.15 ∘	0.15 ∘
	Change	0.17	0.35 •	0.17	0.17 ∘
	Process	0.17	0.19 •	0.15 ∘	0.15 ∘
Butterknife	All	0.24	0.15 ∘	0.13 ∘	0.11 ∘
	Change	0.24	0.26	0.19 ∘	0.18 ∘
	Process	0.24	0.13 ∘	0.13 ∘	0.10 ∘
jcabi-github	All	0.26	0.36 •	0.23 ∘	0.22 ∘
	Change	0.26	0.44 •	0.25	0.25 ∘
	Process	0.26	0.23 ∘	0.22 ∘	0.23 ∘

Hystrix	All	0.28	0.37 •	0.26	0.25 ◦
	Change	0.28	0.39 •	0.27	0.27
	Process	0.28	0.28	0.25 ◦	0.25 ◦
Presto	All	0.20	0.25 •	0.17 ◦	0.17 ◦
	Change	0.20	0.32 •	0.20 ◦	0.20
	Process	0.20	0.21 •	0.17 ◦	0.17 ◦
RxAndroid	All	0.20	0.19	0.15 ◦	0.15 ◦
	Change	0.20	0.28 •	0.20	0.17 ◦
	Process	0.20	0.19	0.14 ◦	0.14 ◦
SpongeAPI	All	0.16	0.19 •	0.14 ◦	0.14 ◦
	Change	0.16	0.28 •	0.16	0.15
	Process	0.16	0.17 •	0.14 ◦	0.14 ◦
Spring Boot	All	0.20	0.20 ◦	0.17 ◦	0.16 ◦
	Change	0.20	0.23 •	0.20 ◦	0.20
	Process	0.20	0.19 ◦	0.16 ◦	0.16 ◦
Square OkHttp	All	0.21	0.22	0.17 ◦	0.16 ◦
	Change	0.21	0.26 •	0.20 ◦	0.20 ◦
	Process	0.21	0.19 ◦	0.17 ◦	0.16 ◦
Square Retrofit	All	0.12	0.13	0.09 ◦	0.09 ◦
	Change	0.12	0.22 •	0.10 ◦	0.11 ◦
	Process	0.12	0.11	0.09 ◦	0.09 ◦
	Average	0.21	0.25	0.19	0.18

◦, • statistically significant improvement or degradation

(1) 0-R
(2) Naive Bayes
(3) C4.5 Decision Tree
(4) Random Forest

Acronyms

BFS breadth first search. 72

CDF cumulative distribution function. 55, 91, 93, 99

CI continuous integration. xi, 1–4, 7, 14–17, 21–24, 26–28, 31, 33, 35–37, 39, 41, 46, 55, 57–59, 63, 66–68, 74, 89, 93, 99, 101–103

CVCS centralized version control system. 11, 21

DAG directed acyclic graph. 8, 36

DVCS distributed version control system. 8, 11, 12, 21, 22, 35, 57, 59, 60, 62

IDE integrated development environment. 14

KDE kernel density estimation. 54

MSR mining software repositories. 4, 11, 21, 24, 32, 87

OSS open source software. 1, 4, 16, 31, 33, 41, 66, 78, 101

PDF probability density function. 54

RMSE root-mean-squared error. 95, 96, 102, 117

SCM software configuration management. 7, 57, 62, 68

VCS version control system. 1, 7–9, 11, 14–16, 21–24, 26, 28, 32, 33, 35–37, 39, 41, 57, 60, 61, 68, 71–73, 82, 86, 87, 89, 101, 102

Bibliography

[ABJ10] Erik Arisholm, Lionel C. Briand, and Eivind B. Johannessen. A systematic and comprehensive investigation of methods to build and evaluate fault prediction models. *Journal of Systems and Software*, 83(1):2–17, 2010.

[Alp14] Ethem Alpaydin. *Introduction to machine learning.* MIT press, second edition, 2014.

[BA02] Stephen P Berczuk and Brad Appleton. *Software Configuration Management Patterns: Effective Teamwork, Practical Integration.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[BBM96] V R Basili, L C Briand, and W L Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering*, 22(10):751–761, oct 1996.

[BCSD14] Caius Brindescu, Mihai Codoban, Sergii Shmarkatiuk, and Danny Dig. How do centralized and distributed version control systems impact software changes? *Proceedings of the 36th International Conference on Software Engineering*, undefined(undefined):322–333, 2014.

[BDL10] Alberto Bacchelli, Marco D'Ambros, and Michele Lanza. Are popular classes more defect prone? *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 6013 LNCS:59–73, 2010.

[BGPZ15] Moritz Beller, Georgios Gousios, Annibale Panichella, and Andy Zaidman. When, how, and why developers (do not) test in their IDEs. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2015*, ESEC/FSE 2015, pages 179–190, New York, NY, USA, 2015. ACM.

[BRB+09] Christian Bird, Peter C. Rigby, Earl T. Barr, David J. Hamilton, Daniel M. German, and Prem Devanbu. The promises and perils of mining git. *Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories, MSR 2009*, pages 1–10, 2009.

[BZ14] Chrstian Bird and Thomas Zimmermann. Predicting software build errors, 2014.

[CH11] Marcelo Cataldo and James D Herbsleb. Factors leading to integration failures in global feature-oriented development. In *Proceeding of the 33rd international conference on Software engineering - ICSE '11*, page 161, New York, New York, USA, 2011. ACM Press.

[CM09] Paul S P Cowpertwait and Andrew V Metcalfe. *Introductory time series with R*. Springer Science & Business Media, 2009.

[CS14] Juliet Corbin and Anselm Strauss. *Basics of qualitative research: Techniques and procedures for developing grounded theory*. Sage publications, 2014.

[CS15] Scott Chacon and Ben Straub. *Pro git*. Apress, 2015.

[Dem06] Janez Demšar. Statistical Comparisons of Classifiers over Multiple Data Sets. *The Journal of Machine Learning Research*, 7(Jan):1–30, 2006.

[DLR12] Marco D'Ambros, Michele Lanza, and Romain Robbes. *Evaluating defect prediction approaches: a benchmark and an extensive comparison*, volume 17. 2012.

[DMG07] Paul M Duvall, Steve Matyas, and Andrew Glover. *Continuous integration: improving software quality and reducing risk*. Pearson Education, 2007.

[DSTH12] Laura Dabbish, Colleen Stuart, Jason Tsay, and Jim Herbsleb. Social Coding in GitHub: Transparency and Collaboration in an Open Software Repository. In *Proceedings of the ACM 2012 Conference on Computer Supported Cooperative Work*, CSCW '12, pages 1277–1286, New York, NY, USA, 2012. ACM.

[ETL11] Jon Eyolfson, Lin Tan, and Patrick Lam. Do time of day and developer experience affect commit bugginess. *8th Working Conference on Mining Software Repositories, MSR 2011, Co-located with ICSE 2011, May 21, 2011 - May 22, 2011*, pages 153–162, 2011.

[FF06] Martin Fowler and Matthew Foemmel. Continuous integration. *ThoughtWorks http://www.thoughtworks.com/Continuous Integration.pdf*, page 122, 2006.

[FHT01] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. *The elements of statistical learning*, volume 1. Springer series in statistics Springer, Berlin, 2001.

[GKMS00] Todd L. Graves, Alan F. Karr, U. S. Marron, and Harvey Siy. Predicting fault incidence using software change history. *IEEE Transactions on Software Engineering*, 26(7):653–661, 2000.

[GZSD15] Georgios Gousios, Andy Zaidman, Margaret-anne Storey, and Arie Van Deursen. Work Practices and Challenges in Pull-Based Development: The Integrator's Perspective. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, pages 358–368, 2015.

[Has08] Ahmed E Hassan. Automated Classification of Change Messages in Open Source Projects. *Applied Computing 2008, Vols 1-3*, pages 837–841, 2008.

[Has09] Ahmed E. Hassan. Predicting faults using the complexity of code changes. *Proceedings - International Conference on Software Engineering*, pages 78–88, 2009.

[HBB+11] Tracy Hall, Sarah Beecham, David Bowes, David Gray, and Steve Counsell. A Systematic Review of Fault Prediction Performance in Software Engineering. *IEEE Transactions on Software Engineering*, 38(6):1276–1304, 2011.

[HG09] H He and E A Garcia. Learning from Imbalanced Data. *IEEE Transactions on Knowledge and Data Engineering*, 21(9):1263–1284, sep 2009.

[HGH08] Abram Hindle, Daniel M. German, and Ric Holt. What do large commits tell us?: a taxonomical study of large commits. *Proc. MSR*, pages 99–108, 2008.

[HL08] Lile P. Hattori and Michele Lanza. On the nature of commits. *Aramis 2008 - 1st International Workshop on Automated engineeRing of Autonomous and runtiMe evolvIng Systems, and ASE2008 the 23rd IEEE/ACM Int. Conf. Automated Software Engineering*, pages 63–71, 2008.

[HZ06] Ahmed Hassan and Ken Zhang. Using Decision Trees to Predict the Certification Result of a Build. *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*, pages 189–198, 2006.

[HZ13] Kim Herzig and Andreas Zeller. The Impact of Tangled Code Changes. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 121–130, Piscataway, NJ, USA, 2013. IEEE Press.

[JTK13] Tian Jiang, Lin Tan, and Sunghun Kim. Personalized defect prediction. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 279–289, nov 2013.

[KCM07] Huzefa Kagdi, Michael L Collard, and Jonathan I Maletic. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *Journal of Software Maintenance and Evolution: Research and Practice*, 19(2):77–131, 2007.

[KGB+14] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M German, and Daniela Damian. The Promises and Perils of Mining GitHub. In

Proceedings of the 11th Working Conference on Mining Software Repositories, MSR 2014, pages 92–101, New York, NY, USA, 2014. ACM.

[KKA14] Noureddine Kerzazi, Foutse Khomh, and Bram Adams. Why Do Automated Builds Break? An Empirical Study. *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 41–50, 2014.

[KM14] John Kloke and Joseph W McKean. *Nonparametric statistical methods using R.* CRC Press, 2014.

[KSD11] Irwin Kwan, Adrian Schröter, and Daniela 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):307–324, 2011.

[KZWG11] Sunghun Kim, Hongyu Zhang, Rongxin Wu, and Liang Gong. Dealing with noise in defect prediction. *2011 33rd International Conference on Software Engineering (ICSE)*, pages 481–490, 2011.

[KZWZ07] Sunghun Kim, Thomas Zimmermann, E. James Whitehead, and Andreas Zeller. Predicting faults from cached history. *Proceedings - International Conference on Software Engineering*, pages 489–498, 2007.

[McK12] Wes McKinney. *Python for data analysis: Data wrangling with Pandas, NumPy, and IPython.* O'Reilly Media, Inc., 2012.

[MJ15] Lech Madeyski and Marian Jureczko. Which process metrics can significantly improve defect prediction models? An empirical study. *Software Quality Journal*, 23(3):393–422, 2015.

[MPS08] Raimund Moser, Witold Pedrycz, and Giancarlo Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. *Proceedings of the 13th international conference on Software engineering - ICSE '08*, page 181, 2008.

[NB07] Nachiappan Nagappan and Thomas Ball. Using software dependencies and churn metrics to predict field failures: An empirical case study. *Proceedings - 1st International Symposium on Empirical Software Engineering and Measurement, ESEM 2007*, pages 364–373, 2007.

[NBZ06] Nachiappan Nagappan, Thomas Ball, and Andreas Zeller. Mining metrics to predict component failures. *Proceedings of the 28th international conference on Software engineering*, pages 452–461, 2006.

[NZZ+10] Nachiappan Nagappan, Andreas Zeller, Thomas Zimmermann, Kim Herzig, and Brendan Murphy. Change bursts as defect predictors. *Proceedings - International Symposium on Software Reliability Engineering, ISSRE*, pages 309–318, 2010.

[PGETD15] Vera Pawlowsky-Glahn, Juan José Egozcue, and Raimon Tolosana-Delgado. *Modeling and analysis of compositional data.* John Wiley & Sons, 2015.

[RD13] Foyzur Rahman and Premkumar Devanbu. How, and why, process metrics are better. *Proceedings - International Conference on Software Engineering*, pages 432–441, 2013.

[RHT+13] Danijel Radjenovic, Marjan Hericko, Richard Torkar, Aleš Zivkovic, Danijel Radjenović, Marjan Heričko, Richard Torkar, and Aleš Živkovič. Software fault prediction metrics: A systematic literature review. *Information and Software Technology*, 55(8):1397–1418, 2013.

[RKBD14] Foyzur Rahman, Sameer Khatri, Earl T. Barr, and Premkumar Devanbu. Comparing static bug finders and statistical prediction. *Proceedings of the 36th International Conference on Software Engineering - ICSE 2014*, pages 424–434, 2014.

[SBZ12] Emad Shihab, Christian Bird, and Thomas Zimmermann. The effect of branching strategies on software quality. *Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement - ESEM '12*, page 301, 2012.

[Sch10] Adrian Schroter. Predicting build outcome with developer interaction in Jazz. *2010 ACM/IEEE 32nd International Conference on Software Engineering*, 2:511–512, 2010.

[She03] David J Sheskin. *Handbook of Parametric and Nonparametric Statistical Procedures.* crc Press, 2003.

[Shm10] Galit Shmueli. To Explain or to Predict? *Statistical Science*, 25(3):289–310, 2010.

[SO13] Rachel Schutt and Cathy O'Neil. *Doing data science: Straight talk from the frontline.* O'Reilly Media, Inc., 2013.

[SSE+14] Hyunmin Seo, Caitlin Sadowski, Sebastian Elbaum, Edward Aftandilian, and Robert Bowdidge. Programmers' build errors: a case study (at google). *Proceedings of the 36th International Conference on Software Engineering - ICSE 2014*, (Section 2):724–734, 2014.

[Swa76] E. Burton Swanson. The Dimensions of Maintenance. In *2nd international conference on Software engineering*, volume XXXIII of *ICSE '76*, pages 81–87, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[ŚZZ05] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? *ACM SIGSOFT Software Engineering Notes*, 30(4):1, may 2005.

[VYW+15] Bogdan Vasilescu, Yue Yu, Huaimin Wang, Premkumar Devanbu, and Vladimir Filkov. Quality and productivity outcomes relating to continuous integration in GitHub. *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2015*, pages 805–816, 2015.

[WP12] Hyrum K. Wright and Dewayne E. Perry. Release engineering practices and pitfalls. *Proceedings - International Conference on Software Engineering*, pages 1281–1284, 2012.

[WSDN09] Timo Wolf, Adrian Schröter, Daniela Damian, and Thanh Nguyen. Predicting build failures using social network analysis on developer communication. *Proceedings - International Conference on Software Engineering*, pages 1–11, 2009.

[YFZ+16] Meng Yan, Ying Fu, Xiaohong Zhang, Dan Yang, Ling Xu, and Jeffrey D. Kymer. Automatically classifying software changes via discriminative topic model: Supporting multi-category and cross-project. *Journal of Systems and Software*, 113:296–308, 2016.

[YWF+15] Yue Yu, Huaimin Wang, Vladimir Filkov, Premkumar Devanbu, and Bogdan Vasilescu. Wait For It: Determinants of Pull Request Evaluation Latency on GitHub. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 367–371, 2015.

[ZLXS15] Yun Zhang, David Lo, Xin Xia, and Jianling Sun. An Empirical Study of Classifier Combination for Cross-Project Defect Prediction. *2015 IEEE 39th Annual Computer Software and Applications Conference*, pages 264–269, 2015.