An Empirical Analysis of Build Failures in the Continuous Integration Workflows of Java-Based Open-Source Software

Thomas Rausch*, Waldemar Hummer*, Philipp Leitner†, Stefan Schulte*

* Distributed Systems Group
Vienna University of Technology, Austria
{rausch, hummer, schulte}@dsg.tuwien.ac.at

† Software Evolution and Architecture Lab
University of Zurich, Switzerland
leitner@ifi.uzh.ch

***Abstract*—Continuous Integration (CI) has become a common practice in both industrial and open-source software development. While CI has evidently improved aspects of the software development process, errors during CI builds pose a threat to development efficiency. As an increasing amount of time goes into fixing such errors, failing builds can significantly impair the development process and become very costly. We perform an in-depth analysis of build failures in CI environments. Our approach links repository commits to data of corresponding CI builds. Using data from 14 open-source Java projects, we first identify 14 common error categories. Besides test failures, which are by far the most common error category (up to >80% per project), we also identify noisy build data, e.g., induced by transient Git interaction errors, or general infrastructure flakiness. Second, we analyze which factors impact the build results, taking into account general process and specific CI metrics. Our results indicate that process metrics have a significant impact on the build outcome in 8 of the 14 projects on average, but the strongest influencing factor across all projects is overall stability in the recent build history. For 10 projects, more than 50% (up to 80%) of all failed builds follow a previous build failure. Moreover, the fail ratio of the last k=10 builds has a significant impact on build results for all projects in our dataset.**

***Keywords*-mining software repositories; continuous integration; build errors; correlation analysis**

I. Introduction

The social coding ecosystem has dramatically changed the way open-source software (OSS) is developed [1]–[3]. In particular, hosted version control systems (VCS), such as Bitbucket or GitHub, and hosted build automation platforms, such as Bamboo or Travis-CI, have made continuous integration (CI) widely available for OSS projects of every size. While CI has reportedly improved the quality of processes and the developed software itself [4], there are also potential drawbacks. Failing builds can lead to inefficiencies and significantly impair the development process [5]. A comprehensive understanding of both, build errors and factors that impact build outcomes, is necessary to uncover problems in the development process and subsequently increase development efficiency.

Despite the widespread use of CI, little is known about the variety and frequency of errors that cause builds to fail, or development practices that impact the outcome of builds. Yet, during development, a large amount of time and focus goes into finding such errors and fixing broken builds [5]. Previous research on build errors has focused on compiler errors [6] or test failures [7], and has not considered other aspects of CI builds. Further studies of CI build failures either have a narrow scope of explored metrics (e.g., socio-technical factors) [8], [9], or consider only a single closed-source project [5], [10]. Also, the pull-based development model, as facilitated by Git-based code hosting platforms, has introduced new aspects to the development process [3] that have not been considered in prior work on build failure analysis.

In this paper, we present an empirical study of CI build failures in 14 Java-based OSS projects. We extract and analyze data from publicly available GitHub repositories and Travis-CI build logs to address the following research questions:

RQ1. *Which types of errors occur during CI builds of Java-based open-source systems?*
We explore CI build data to find which types of errors occur in CI builds, and to determine the frequency of occurrence of these errors. We use a semi-automated procedure to systematically classify build logfiles and map them to different error types. This data is then aggregated to gather quantitative evidence about the frequency of different error types. We observe that the most common reasons for build failures are failing integration tests, code quality measures being below a required threshold, and compilation errors. These findings help developers prioritize efforts in reducing build failures.

RQ2. *Which development practices can be associated with build failures in such systems?*
Based on existing research on software defect and build failure analysis, combined with observations from the pull-based development model, we formulate a set of 16 process and CI metrics that we hypothesize to influence build results. After extracting and linking data from different sources, we perform statistical correlation tests to examine the strength of such influence. Our results show that change complexity and author experience can describe build outcomes reasonably well. The strongest influencing factor for failing builds, however, is overall stability of the build system in the recent build history.

Our research has important implications for both, practitioners and researchers. Practitioners should closely monitor the stability of their CI system to avoid the perpetuation of build failures. An unstable build environment or tests that cause builds to fail randomly should be quickly identified and neutralized. This will reduce the noise in historical build data and allow for more precise data analytics to identify causes of build errors. Our metrics provide means of identifying such builds in historical data. Software developers should develop a habit of running full CI builds locally before pushing complex changes. Developers of CI systems should address common problems in the interaction between build servers and VCS repositories, as this is a considerable source of errors. Researchers should be aware that, through the proliferation of the PR-based development model, CI build data has inherited the non-linear nature of VCS data. Without explicitly addressing this structural mismatch, empirical analyses on these data may provide incorrect or biased results.

II. Related Work

CI and the related fields of continuous delivery/deployment (CD) have received considerable attention in recent years [11]. CI comes in various flavors. In their analysis of CI practices in industry, Ståhl et al. [12] show that CI usage is not homogeneous, but rather has a multitude of variation points, including reliability aspects such as fault handling, fault duration, and integration. Despite its increasing adoption, CI also comes with challenges. Vassallo et al. [13] found in a study at ING that test automation accounts for a significant portion of the total development effort, with a third of developers allocating more than 50% of their time for testing activities. To make matters worse, automation scripts are often not reliable in provisioning the required test infrastructure [14]. Schermann et al. [15] conclude that architectural issues are one of the main barriers for CI/CD adoption in the broader software industry. Furthermore, advanced CI practices such as partial rollouts are highly complex and require better abstractions. Neely et al. [16] emphasize the need to remove manual steps from the deployment process, to heavily invest in fast automated tests, and to create test environments that mimic production. Other research has focused on software engineering processes such as the *pull-based development* model in combination with CI. Yu et al. [17] study determinants of pull request (PR) evaluation latency of GitHub projects using Travis-CI. Among the main impact factors of PR latency are process-related factors as well as the presence of CI. Gousios et al. [18] determine that 75% of projects that use PRs also use CI. Beller et al. [7] study CI builds specifically with regard to the impact of testing. A core finding is that automated tests are the single most important reason why builds fail.

Analysis of *build errors* is facilitated by standardized project structures and build lifecycle models, e.g., Maven/Gradle [19]. In an empirical study, Seo et al. [6] examined over 26 million builds triggered in Google's centralized build environment. Error types were elicited from the build system's log output, based on expert assessments. Kerzazi et al. [5] conducted a study based on a closed-source project to identify the impact factors of build failures. They identified missing referenced files, mistakenly committed work-in-progress code, and transitive dependency errors as main culprits. Cataldo and Herbsleb [20] found that cross-feature dependencies increase the likelihood of integration failures. They also found that organizational attributes can have an even higher impact on integration failures than purely technical attributes. There are also ongoing research efforts that focus on prediction of build failures. The pioneering work by Hassan and Zhang [10] uses decision trees to predict build outcomes. The majority of existing prediction approaches are based mostly on socio-technical factors [8], [9] which can have strong influence in certain team settings. Work on build error prediction also appears in the recent patent literature [21] and with its cost savings potential it will likely remain a hot topic in the future.

To the best of our knowledge, no existing work has analyzed the types of and reasons for CI build failures across a heterogeneous set of Java-based OSS. We argue that this research is important to augment and validate existing work in the area, which has so far focused on closed source software and smaller sample sizes.

III. Study Setup

In this section, we present the setup of our study. We focus on project selection and data collection. Detailed approaches for RQ1 and RQ2 are discussed in the respective sections.

A. Research Subjects

The study is conducted using real-world data gathered from 14 OSS projects that employ CI using GitHub and Travis-CI. To contain the complexity of the data analysis process, we restrict the study to systems written in Java, and projects that use Maven or Gradle as their build automation tools. This allows us to make reasonable assumptions about the structure of the codebase and build log formats.

To find highly active projects fitting these requirements, the GitHub repository list was queried and the results were sorted by popularity. The initial set of repositories was first filtered by their use of Travis-CI. Furthermore, we defined a set of criteria to ensure that the selected projects provide sufficient data for our analyses. Our acceptance criteria were: a) number of builds ≥ 400 (this also ensures a similar amount of commits), b) number of committers ≥ 50, and c) at least one commit or build in the last month (baseline March 2016). This resulted in the selection of projects listed in Table I. The column *Build freq.* denotes the mean time (in days) between builds. The *Fail ratio* describes the fraction of builds that failed.

B. Data Collection

Our research questions require two kinds of data: build data from CI servers, and change data from VCS repositories. In particular, we require metadata of build executions (outcome, execution duration, etc.), log output of the build systems, revision deltas, and metadata about commits (author, date, etc.). Using GHTorrent [22] and TravisTorrent [23] as data sources is impractical as they do not contain the information required for the detailed analysis of build errors, job execution durations, or revision deltas. Instead, we implement a custom data crawler and continuous repository mining.

TABLE I
NAME, DESCRIPTION AND METADATA OF PROJECTS USED AS RESEARCH SUBJECTS

Name	Description	Age VCS*	Commits	Committers	Age CI*	Builds	Build freq.*	Fail ratio
Apache Storm	Distributed computation framework	1961	11656	253	647	5472	8.5	0.69
Crate.IO	Scalable SQL database	1390	13722	69	1024	21864	21.4	0.63
JabRef	Application for managing BibTeX databases	1407	16851	112	1046	9615	9.2	0.29
Butterknife	Android dependency injection library	1426	894	114	1426	1220	0.9	0.34
jcabi-github	Object-oriented wrapper of Github API	1179	2543	60	1024	1316	1.3	0.45
Hystrix	Fault tolerance library for distributed systems	1532	2321	103	880	1228	1.4	0.48
Openmicroscopy	Microscopy data environment	4364	55571	59	1501	16383	10.9	0.19
Presto	Distributed SQL query engine for big data	1635	23500	308	1180	19112	16.2	0.49
RxAndroid	RxJava bindings for Android	1264	495	74	880	728	0.8	0.16
SpongeAPI	Minecraft plugin API	874	3692	213	874	8835	10.1	0.24
Spring Boot	Java application framework	1561	11300	566	1276	10051	7.9	0.28
Square OkHttp	HTTP+HTTP/2 client for Android and Java	1651	3832	263	1576	7439	4.7	0.49
Square Retrofit	HTTP client for Android and Java	2336	1640	231	1576	3040	1.9	0.20
WordPress-Android	WordPress for Android	2697	21548	66	1191	15025	12.6	0.14

* in days

1) Data Crawler: The build metadata are gathered by querying the Travis-CI REST API[1] using a custom-built crawler, and are stored locally for ad hoc analysis. Logfiles of builds are crawled from Amazon S3, the storage backend used by Travis-CI. Code change data is extracted from the projects' Git repository using a combination of Git commands and Python scripts to process the output of these commands.

2) Continuous Repository Mining: A well-known problem in mining VCS repositories is the potential loss of historical data through history manipulation operations [24] and the PR-based development model [25]. For example, when a PR is updated, GitHub creates a transient merge commit that simulates the merge, which is then checked out and built by Travis-CI. Unless explicitly pulled, these commits are not part of the local history, and are discarded after some days in the remote repository. This prevents any detailed analysis of PRs from historical data. Furthermore, a common practice in code reviews of PRs is to amend the suggested changes to the respective commit (by squash rebasing), and hence to overwrite the remote history. The updated commits are then built by the CI server, but the amended commits are lost on the remote site. To retain all historical data, we implemented continuous repository mining [26]. Each time a change is made to a branch or a PR, a bot fetches these changes, retaining all commits that may have been lost through subsequent manipulations of the Git history. While this method allows us to precisely analyze the changes that were built, it also limits the analysis to the data collected by the bot. We continuously mined the repositories for a period of eight months (from May 2016 to January 2017). In our final dataset, on average, 47% of builds have associated change data, varying from 27% (SpongeAPI) to 72% (Spring Boot).

IV. TYPES OF BUILD ERRORS

In this section, we answer RQ1. We discuss qualitative and quantitative data on build errors, gathered from the 14 OSS projects introduced in Section III.

[1]https://docs.travis-ci.com/api

A. Approach

Build tools, such as Maven or Gradle, report errors in the form of plaintext log output. Travis-CI records the output for each build and stores it into logfiles. These logfiles are the basis for this part of our study.

To elicit error categories, logfiles have to be systematically analyzed. A well-known method in qualitative research, employed for the exploration and labeling of data, is open coding [27]. Seo et al. [6] also used this technique to elicit error categories from compiler messages. Based on the concept of open coding, we developed *LogCat*, a tool to assist the process of exploring and analyzing build log data. The two main parts of LogCat are a parser to find and label logfiles based on message patterns, and an internal data structure to organize logfiles and labels. It also provides a command-line interface to view, search, and label logfiles.

To create a taxonomy of error categories, we performed several coding iterations. A coding iteration for an individual project was performed as follows. An unlabeled logfile was selected at random and the error message was analyzed. A search pattern was generalized from the given error message. The set of logfiles matching the pattern was examined to determine whether the pattern indeed captured a specific error type. If so, the pattern was stored and assigned a label, along with all matching logfiles. To account for variations in error messages, LogCat allows a one-to-many mapping between labels and message patterns. The labeling process was repeated until at least 90% of logfiles were matched by a search pattern. Each logfile was assigned at most one label. We then refined search patterns and labels in subsequent iterations.

We were unable to conduct the error categorization for two of the 14 projects. Openmicroscopy uses a very heterogeneous build environment that prevented us from making sound decisions about error types. Similarly, the builds for Wordpress-Android reported vague or incomprehensible error messages. Although we excluded these projects from the error categorization process, their build data was still used for the statistical analyses presented in Section V.

TABLE II
DESCRIPTION AND FREQUENCY OF ERROR CATEGORIES

Label	Projects	Description
testfailure	12	An automated test did not pass
compile	12	Compilation error
git	12	VCS interaction error, e.g., worker fails to fetch code
buildconfig	11	Faulty build config, e.g., syntax error in pom.xml
crash	11	Build environment crashed or exceeded time limit
dependency	11	Dependency error, e.g., invalid version number
quality	10	Coding-rule violation during code inspection
unknown	9	Errors without a clearly identifiable cause
itestfailure	4	An automated integration test failed
doc	3	Documentation issue, e.g., undocumented methods
license	3	License criteria not met, e.g., missing license headers
compatibility	2	API incompatibility, e.g., due to version conflict
androidsdk	1	Android SDK-related error, e.g., download failed
buildout	1	Specific build error of Python submodule in Crate.IO

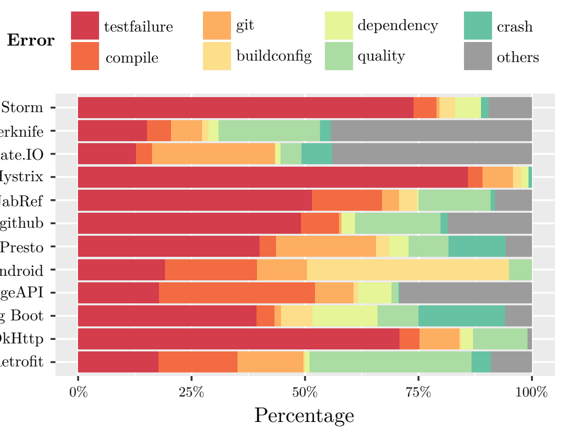

Fig. 1. Distribution of common error categories

B. Results

We analyzed a total of 54 248 logfiles, 92% of which were successfully assigned an error category. Overall, the analyzed projects exhibit a mean failure ratio of 37%, and only four projects have a failure ratio above 50% (see also Table I). In terms of Travis-CI build status (errored, failed, or passed), most failed builds have the status *failed*. This affirms the results of [7].

1) Error Categories: A total of 21 different labels were created and then grouped into categories. For example, some projects use multiple static code analysis tools to measure code quality. We labeled errors generated by each tool separately (e.g., *checkstyle* or *findbugs*), and later grouped these labels into the category *quality*. This resulted in a total of 14 and an average of 9 categories per project. Table II lists the 14 categories we identified, the amount of projects they occurred at least once in, and a short description of each error type.

From 14 error categories, test failures, compile errors and VCS interactions errors are the only ones common across all projects. 10 out of 12 projects use automatic code inspection, which poses a potential threat to build stability.

2) Error Frequency: To determine the frequency of errors, we counted all logfiles that matched an error message pattern of the respective error category. A build may comprise several build *jobs* (e.g., to execute builds in different runtime environments) and each job may fail independently. If multiple jobs of a build fail, we only count each error type once. Figure 1 shows the frequency of error categories common to 10 or more projects.

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 *testfailure*, 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 appears infeasible without deep knowledge of the systems' test configurations, and it would also increase the number of defined categories, which may be undesirable for analysis purposes.

Test failures are by far the most common reason for build failures (up to >80%). There is also a non-negligible amount of noise in the build data, e.g., caused by Git interaction errors, or general infrastructure flakiness.

3) Build Outcome By Execution Phase: A main goal of CI is to provide rapid feedback on the state of code changes. Waiting a long time on such feedback can have a negative impact on the development process [5]. We therefore consider it useful to analyze *when* errors occur during the build execution, in order to identify errors that have the highest potential to delay feedback.

To measure build runtime, we cannot rely on the build duration reported by Travis-CI, because it may not be accurate for builds that comprise multiple jobs. The reported value is the cumulative build time of all jobs, which is impractical for our purpose of detecting time values of specific build errors. Instead, we determine the build execution duration based on Travis-CI's job metadata (start/end timestamps).

Early analyses showed that the average build duration may change dramatically during the project's lifetime, and also vastly differs across projects. Hence, we normalize the runtime data for each project as follows: we compute the rolling build duration median of *passed* builds, with window size $k = \frac{n}{100}$, where n is the number of builds. We then divide the duration of each build by the respective rolling median value. This yields a representative distribution, where the relative build duration 1.0 denotes the average runtime of a passed build.

Figure 2 shows a box plot of the build execution duration for the 8 most common build result categories. To account for imbalances in the dataset (see Table I), we sampled 500 observations from each project. Our results indicate that *git* errors entirely and *dependency*/*buildconfig*/*compile* errors predominantly occur in the first half of an average build

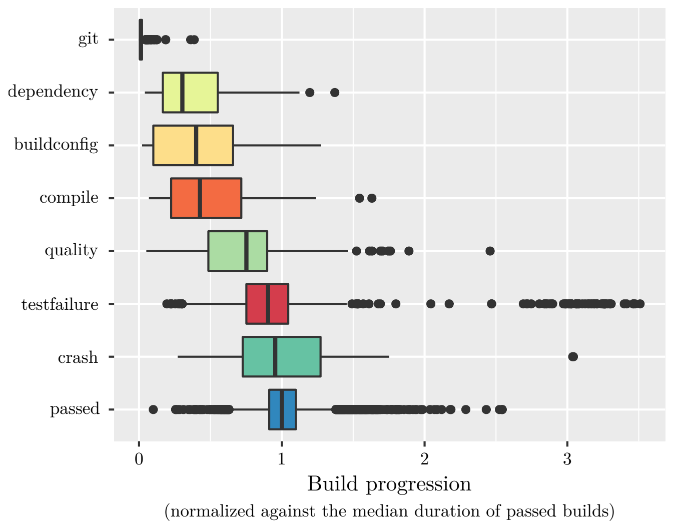

Fig. 2. Occurrence of different result categories during the build progression

duration. The distribution of build *crashes* is relatively uniform around the average mark for successful build durations, with a few outliers, typically caused by deadlocked builds that have stopped producing log output for a certain amount of time and are subsequently terminated by Travis-CI. We also observe that *testfailure* has the highest outlier rate. In our investigations, we found that many projects use the *retry* feature, which forces the build worker to re-initiate a failing build phase for a specific number of times before terminating. This is often used to alleviate external factors, e.g., a dependency server that is transiently unavailable. However, if the build fails towards the end of a test suite, and the entire build is re-initiated several times, the build duration will be correspondingly high.

On average, 30% of errors occur in the first half of the build runtime. The later half is dominated by 70% testfailures. Together with a build-retry approach, testfailures can cause long delays in the feedback loop.

C. Discussion

Using LogCat, we elicited 14 different error types by analyzing a total of 54 248 logfiles. We found that, on average, the most common reasons for build failures are failing unit tests, code quality measures being below a certain threshold, and compilation errors. A surprisingly high number (on average 9% and up to 27%) of builds fail because the build worker cannot fetch the change data from GitHub. We observed that PRs are often updated and immediately merged. The PR update causes a build to be triggered, but when Travis-CI initiates the build worker, the PR data on GitHub is already gone, causing the build to fail immediately.

The impact of a build failure on the development process depends on the error that caused the failure [6]. Whereas compile errors are usually straight-forward to fix, errors during the test phase can result from, e.g., race conditions in integration tests, which are notoriously difficult to debug and fix. Although not in scope for this paper, in future work it will be interesting to analyze the *mean time to repair* for different error types.

Interpreting the temporal dimension of build results allows us to reason 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, if a build is already halfway through the average build duration, it becomes increasingly unlikely that the build will terminate with a *buildconfig* error (see Figure 2). Such data might be suitable to serve as the basis for build error prediction, which is part of our future work.

V. Impact of Development and Deployment Practices on Build Results

We now turn towards RQ2, and study the factors that influence whether CI builds succeed or fail.

A. Approach

First, we define metrics (see Section V-B) to characterize individual changes (complexity of commits, file types, and experience of committers) and the overall CI process (build histories, build triggers, pull request scenarios). We proceed to extract these metrics from the build and VCS repositories using the approach described in Section III. Finally, we use statistical hypothesis testing to determine the strength of correlation between these metrics and build failures.

1) Data Processing: Data in VCSs is graph-structured, and the used source code management (SCM) workflow has an impact on the structure of the version history and change deltas [2]. In contrast, build data from CI servers consist of unstructured or semi-structured logfiles, and a traditional relational data model for build metadata. For data gathering, this structural mismatch between change and build data has to be overcome. We call this *process topology mapping*, and employ an approach similar to the one presented in [7]. We leverage the metadata provided by Travis-CI (e.g., commit that triggered a build) to create a graph homomorphism. Figure 3 illustrates how build data is mapped to commits in a common branching scenario. Circles represent individual commits, and green and red elements indicate successful and failed builds, respectively. Note that arrows indicate predecessor relationships, pointing from children to parents.

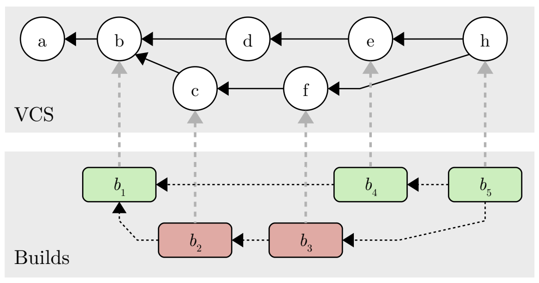

Fig. 3. Linking build and change data

Through topology mapping, the linked build data inherits the graph structure of the Git history. This allows us to a) define an ordering and determine predecessors of builds based on the VCS history topology, and b) precisely determine the change set C_b of a build b. For example, the change set C_{b_4} contains the commits $\{d, e\}$. The last commit in C_b is the trigger of build b. If the trigger is a merge commit, we consider the effective changes to be all commits in C_b, except the merge commit. In case the change set includes only a merge commit, the effective changes are made up of that commit.

Additionally, we stratify the data based on the previous build outcome. When a build fails, the immediately following commit is likely to be an attempt to fix the underlying problem. Such changes have a different intent than changes following a successful build. As we want to study changes that lead to build failures, it is important to make a distinction between changes that a) introduce failures (*break*), b) perpetuate failures (*broken*), c) fix failures (*fix*), and d) are unrelated to failures (*passing*). In Figure 3, examples for these categories are a) b_2, b) b_3, c) b_5, and d) b_1, b_4. In previous research, this distinction was typically not taken into consideration. This is particularly concerning as our study shows that the previous build outcome is by far the strongest predictor for the build outcome (see also Section V-C). That is, if the build failed, the next build is overproportionally more likely to fail as well. To account for this in our statistical analyses, we first stratify our data and consider only builds where changes introduced failures, or are unrelated to failures. We also filter extreme outliers to account for data anomalies such as the first build in the project (which has linked to it all previously made commits in the repository). We do this by removing data that are outside the 99^{th} percentile of respective metrics, such as the number of commits or the number of lines added (see Section V-B). Table III lists the amount of data for each project before and after stratification and cleaning.

TABLE III
OBSERVATIONS IN DATASET BEFORE AND AFTER PROCESSING

	Unstratified		Stratified and cleaned			
	Failed	Passed	Break	Broken	Fix	Passing
Apache Storm	3761	1706	246	1512	458	260
Butterknife	410	805	22	113	26	423
Crate.IO	14649	7210	1170	2515	1284	2072
Hystrix	584	639	91	236	136	206
JabRef	2783	6827	561	1087	637	3714
jcabi-github	584	727	138	149	150	255
Openmicroscopy	3191	13187	326	536	911	3178
Presto	9728	9379	843	1723	1183	2775
RxAndroid	118	605	11	19	23	223
SpongeAPI	2122	6708	170	121	138	1923
Spring Boot	2804	7242	708	1176	841	4382
Square OkHttp	3610	3824	403	812	528	1578
Square Retrofit	608	2427	22	32	32	993
WordPress Android	2161	12859	390	1124	514	8347

2) Statistical Analyses: For each extracted metric, we study its correlation with the build outcome. As we cannot make assumptions about the underlying probability distribution, we rely on nonparametric statistical tests. We select the tests based on the decision procedure described by Sheskin [28].

Categorical variables, such as build type (BT) or pull request scenario (PRS), are examined using Pearson's χ^2-test. We report the effect size with Cramér's V, $\phi_c \in [0, 1]$. Higher ϕ_c values indicate a stronger 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 (in the following called Mann–Whitney test). We report the effect size with the rank-biserial correlation r from the U statistic. Using these tests, we perform standard significance testing of null hypotheses, i.e., that there is no relationship between the respective metric and the build outcome. For every statistical significance test, we determine the p-value of the test, and evaluate the null hypothesis using the common level of significance $\alpha = 0.05$. If $p < \alpha$, we reject the null hypothesis and conclude that there is a statistical significant relationship between the samples.

B. Candidate Metrics

A core aspect of identifying causes of build failures is deciding which factors to test, and which metrics to use as proxies for these factors. We follow a comprehensive approach, and test a combination of novel metrics and ones which have been suggested as good proxies for potential causes of build failures in existing literature, amounting to 16 different metrics.

1) Process Metrics: Process metrics categorize and quantify characteristics of commits that trigger automated builds. We extracted 11 popular process metrics from existing research. Table IV lists for each metric a brief definition. Following, we describe the motivation and intuition behind each of the selected metrics in more detail.

Complexity of changes: This category includes size and complexity measurements of the change set. The underlying rationale is that complex changes are more error-prone. Complexity of changes (e.g., size, churn, or entropy) has already been found to impact software quality [5], [30]. In software defect prediction, the most widely used process metrics are number of revision (NR), number of distinct committers (NDC), number of modified lines (NML), and number of defects in previous versions (NDPV) [31]. Complexity measures using entropy functions from information theory have also proven suitable to predict faults [30] and build failures [10].

File types: We hypothesize that changes to specific types of files will more frequently lead to certain types of errors. For example, changes to a README file are unlikely to cause a compilation error. Similarly, *buildconfig* errors will typically be caused by changes to a build configuration file. To study the relationship between changes to file types and build failures, we implement a simple file type classification scheme similar to [32]. We leverage the project structure convention dictated by Maven or Gradle to categorize files based on their names or location in the file tree. Specifically, we define the following categories: system code, system resources, test code, test resources, properties, documentation, git (e.g, `.gitignore` file), build config, CI config (e.g., the Travis-CI config file), webapp (e.g., JSP files), benchmark (e.g., JMH benchmark classes), and lib (binary dependencies).

TABLE IV
NAME AND DESCRIPTION OF PROCESS METRICS

Metric	Short	Description
Complexity of changes		
Number of commits	NC	The size of the change set C_b, i.e., the amount of commits that were pushed since the last build.
Number of authors	NA	The number of distinct authors involved in the change set.
Number of modified files	NMF	The total number of distinct files modified across all commits of the change set.
Lines added	NLA	The number of lines added across all commits of the change set.
Lines removed	NLR	The number of lines removed across all commits of the change set.
Change scattering	CX	We adapt the Shannon entropy to measure how changes (in terms of number of modified files) are scattered across files. We normalize the value with the total number of files in the system at the observed time [10].
File types		
File types	FT	The types of files that were changed in C_b as a concatenation of labels. For example, if a system and a test file were changed, the value for FT will be *system+test*. To contain the amount of categories, if C_b contains changes to more than three different file types, we label it *tangled* [29].
Date and time		
Time of Day	TD	The time-zone adjusted time of day $[0, 23]$ of the commit that triggered the build.
Weekday	WD	The time-zone adjusted weekday $[0, 6]$ (0 being Monday) of the commit that triggered the build.
Author		
Author experience	AX	The time difference in days between a developer's first commit and the current observation. If the effective change set contains multiple authors, we define the main author as the person with the most commits in the change set. We break ties by choosing the author of the latest commit.
Author commit frequency	ACF	The author's most occurring time difference between consecutive commits. To calculate ACF, we categorize the commit frequency into: daily, weekly, monthly, other (<20 commits), and single (1 commit).

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 [33]. We test whether this hypothesis holds for CI builds by analyzing the time-of-day (TD) and day-of-the-week (WD) that changes were made. We follow the approach by Eyolfsson et al. [34] to account for different time zones of developers.

Author: Eyolfsson et al. [34] suggested that authors' experience (AX) and commit frequency (ACF) influence how likely a commit is going to be buggy. We adapt the classification approaches suggested in [34] to fit our data. We define the author of a build to be the main author of the effective change set of that build, i.e., the developer with the highest number of commits in the effective change set.

2) CI Metrics: Modern SCM tools allow a high degree of freedom when it comes to implementing workflows [18]. These workflows, particularly branching and pull-based workflows, have been found to impact both software quality and work productivity [4], [35]. Another goal of our study is to investigate different workflow characteristics, and how they are related to build results. Based on existing research, we formulated five CI metrics. Table V again lists for each metric a brief definition, with a more detailed discussion of the intuition and rationale behind each metric following.

TABLE V
NAME AND DESCRIPTION OF CI METRICS

Metric	Short	Description
Build type		
Build type	BT	The build type BT $\in$ {PUSH, PR, MERGE, INTEGRATION, PR_MERGE, UNKNOWN}. First, we determine the event type (*push* or *pull request*) from the Travis-CI build metadata. Then, we determine from the effective change set whether the build is a regular, merge, or integration commit. Pull request merges are determined from GitHub's commit message scheme "*Merge pull request #pr* [..]".
Pull request scenario		
Pull request scenario	PRS$_x$	Information about the circumstance under which a pull request was updated. PRS$_1$: Whether the branch being merged into has changed since the last build. PRS$_2$: Whether the commit history of the PR was changed using Git rebase.
Build history		
Previous build result	PB	The result of the previous build(s). 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.
Build climate	BC	The build climate, or build stability is the fail ratio of the last k builds.
Days since last fail	DLF	The time that has passed since the last build failure [10]. We discretize the variable by creating four intervals: Let t denote 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.

Build type: Builds are executed during different stages of the workflow [5]. This context can give additional meaning to the intent of CI builds. For example, Travis-CI distinguishes between *push* builds (direct commits to a master branch) and *pull request* builds. Furthermore, builds that are triggered by merge commits may also have different characteristics. When a conflict-free PR is merged, the merge commit triggers a new build that contains no changes. In contrast, if integrators manually merge branches locally and then push these changes, the change set may include additional changes to fix conflicts. We extend Travis-CI's build type definition to capture these characteristics, and classify builds as follows: *push* (builds triggered by a regular commit pushed directly into a branch), *merge* (builds triggered by manually pushing a merge commit, where C_b contains only that commit), *integration* (builds triggered by a merge commit, where C_b contains additional commits), *pull request* (builds triggered by creating or updating a PR), and *pull request merge* (builds that are triggered when a PR is merged).

Pull request scenario: Git and GitHub allow a high degree of freedom when implementing the PR workflow. It has been suggested that the employed branching workflow has an impact on software quality [35]. A consequence of isolating development into branches are potential merge conflicts that can subsequently cause build failures. When a PR is updated, parallel changes made in the branch being pulled into may not have been taken into account by the author. Some developers use Git's history manipulation operations to update pull requests. We examine how different characteristics of PRs affect build results. To that end, we define two key characteristics of PRs based on observations in our data. PRS_1) since the last build, changes were made in the branch being merged into, PRS_2) the commit history of the PR was manipulated either via rebase, amend or similar commands.

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 [10]. We extend the approach of Hassan and Zhang [10], to account for the effects of topology mapping, i.e., that a build may have more than one previous build. Specifically, we calculate the result of the previous build (PB), and the days since the last failure (DLF). Additionally, we examine periods of build instability, using the notion of build climate (BC), which is a small moving window over the fail ratio.

C. Results

The remainder of this section presents the results of our statistical analyses for each metric type. Table VI shows the results of the significance tests for each metric and project. The table contains the calculated effect sizes (see Section V-A2), and all cells with a value of $p < 0.05$ are highlighted in the table. In the following paragraphs we examine the results for each group of metrics in more detail.

Caveat: Although our pre-processing accounts for biases in the data, it also has a detrimental effect on the dataset size (see Table III). As we consider only builds in the group *break*, and *passing* (see Section V-A), the resulting small sample sizes for the projects Butterknife, RxAndroid, and Square Retrofit in the category make the results of some tests inconclusive. In particular, results of the χ^2-test may yield incorrect results due to the low amounts of observations [28].

Complexity of changes: In general, we observe that the Mann–Whitney test reveals a significant effect for most variables in many projects. We investigate whether higher complexity relates to an increase in build failures by examining the median or mean values for each metric.

NC: The mean values indicate that most change sets comprise one or two commits. In five of the eight projects where NC correlates with build results, failed builds have higher mean NC values. For the projects Apache Storm, jcabi-github and Openmicroscopy, the opposite holds. *NA:* The median NA are 1 for all projects for both, failed an passed builds. The difference of mean NA values between failed and passed builds are negligible, with a maximum of .03. *NLA/NLR:* The mean values for NLA and NLR are significantly higher for failed builds in every correlating case, with the exception of jcabi-github. The median values support that both, higher NLA and NLR, lead to an increase in build failures. *NMF:* We observe eight correlating cases, in seven of which the mean NMF is higher for failed than for passed builds. For the project Square OkHttp, the mean difference of .1 is negligible. For all other cases, failed builds have, on average, 1.8 more modified files than passed builds. The median values, ranging from 1 to 3, support this only in five cases. *CX:* In all eight correlating cases, we observe that mean CX values are significantly higher for failed builds, with the exception of jcabi-github.

Previous research has suggested that high change complexity leads to an increase in software defects [30] and build failures [10]. Our results provide moderate evidence to support this claim for CI builds.

File types: To perform the χ^2-test and avoid small sample sizes, we select for each project the six FT categories containing the most changes. The most common categories are *system* and *system+test*. The χ^2-test reveals five cases where FT has a significant effect on the build outcome. Although the values indicate a relationship between file type changes and the build outcome, we cannot find evidence whether changes to a specific file type lead to errors more frequently. We even observe some non-intuitive results. For example, in the Spring Boot project, there exist 511 builds where C_b contains only changes to documentation files (e.g., an update of the README file). In 44 cases, these changes introduced build failures, 25 of which were caused by failing tests, 18 by test environment crashes, one because of a dependency error. We observe similar behavior in other projects.

Even objectively harmless changes sometimes break builds. This often indicates unwanted flakiness of tests or the build environment.

Date and time: We perform the χ^2-test on contingency tables of hours $[0 - 23]$ and weekday $[0 - 6]$, respectively.

TD: We observe four cases where TD has significant effect on the build outcome. However, because the data is split into 24 groups (one for each hour), the results are generally inconclusive. For jcabi-github, the results should be interpreted with caution because some groups contain zero values. In two cases, we observe that the late night hours (23-5), have a significantly higher fail ratio. Conversely, for Openmicroscopy these late night commits actually have the *lowest* failure ratio. *WD:* The χ^2-test reveals no significant influence of the weekday on the build outcome.

There is little evidence in our data that the date or time of a change has a positive or negative influence on the build results.

Author classification: *AX:* The Mann–Whitney test reveals eight cases where AX has a significant effect on the build outcome. In five of these cases, *less* experienced developers produced fewer build failures. A possible explanation is that daily committers are more careless when pushing changes. *ACF:* The χ^2-test reveals six case where ACF has a significant effect on the build outcome. We observe only two cases where daily committers cause significantly fewer failures than others.

TABLE VI
SIGNIFICANCE TEST RESULTS FOR ALL METRICS

	NC	NA	NLA	NLR	NMF	CX	FT	TD	WD	AX	ACF	BT	PRS_1	PRS_2	PB	BC	DLF
Apache Storm	.15	.07	−.06	.01	−.01	−.05	.12	.21	.13	.00	.07		.31	.16	.25	−.95	.09
Crate.IO	.01	.00	−.04	.02	−.02	−.02	.04	.07	.04	.11	.04		.13	.07	.31	−.97	.12
JabRef	−.10	−.01	−.30	−.23	−.21	−.21	.14	.09	.05	−.09	.07	.14	.14	.00	.51	−.94	.10
Butterknife	−.02	.00	−.22	−.27	−.26	−.17	.19	.25	.12	−.15	.00	.12			.76	−.99	.31
jcabi-github	.18	.03	.18	.17	.29	.23	.35	.33	.17	−.38	.29				.14	−.92	.07
Hystrix	.04	−.02	.08	.04	.06	.09	.11	.32	.17	−.05	.26				.31	−.90	.12
Openmicroscopy	.03	.00	.03	.00	.03	−.03	.05	.10	.03	−.05	.03		.13	.08	.32	−.98	.12
Presto	−.02	.00	−.09	−.06	−.03	−.05	.05	.13	.03	−.01	.03		.11	.00	.37	−.98	.03
RxAndroid	.09	.01	−.19	−.26	−.18	−.32	.20	.32	.14	−.31	.06	.07			.49	−.98	.24
SpongeAPI	−.07	−.03	−.31	−.24	−.19	−.20	.14	.12	.07	−.11	.08		.30	.00	.36	−.99	.06
Spring Boot	−.04	−.03	−.17	−.13	−.14	−.14	.09	.07	.04	.01	.06	.10	.24	.06	.45	−.85	.08
Square OkHttp	.00	.00	−.02	−.03	.00	−.01	.09	.15	.07	.15	.07	.06	.17	.01	.41	−.96	.16
Square Retrofit	−.02	.01	−.43	−.22	−.33	−.31	.11	.15	.05	.20	.04	.19			.53	−.99	.09
WordPress Android	−.11	.00	−.21	−.21	−.17	−.19	.08	.05	.03	.07	.04	.06	.09	.00	.67	−.94	.08

Highlighted values are $p < 0.05$. Inconclusive results were omitted

In our study, authors that commit less frequently tend to cause fewer build failures.

Build types: We perform the χ^2-test on 8 of the 14 projects. Some projects (e.g., Apache Storm, Crate.IO, Presto) employ the merge-squashing workflow[2]. Due to limitations in our build type classification approach (see Section V-A), we cannot definitively determine the build type in these projects.

The χ^2-test reveals four cases where the build type has a significant influence on the build outcome. We observe three cases where PR builds have a significantly higher failure ratio than push builds. A possible explanation is that, because PRs are isolated from the main development branch, users are less careful when committing changes. Unsurprisingly, merge builds, in particular PR merge builds, exhibit the lowest failure rate. Yet, although PR merge builds are essentially redundant builds of an already tested integration, we observe that these builds can still introduce failures. Such failures may be caused by an unstable build environment or randomly failing tests.

PRs tend to cause failures more frequently than changes pushed directly into a branch. Successfully building a PR does not guarantee the success of a subsequent PR merge.

Pull request scenarios: We perform the χ^2-test on each PRS criterion. Only 9 of the 14 projects contain sufficient information about the PR scenario to provide conclusive results. To our surprise, none of the criteria have a significant effect on the build outcome. In some cases, however, the results are inconclusive due to a lack of data. We intuitively assumed that PRs whose base branch has changed since the last build (PRS_1) would lead to build failures more often. However, our data does not support this hypothesis.

There is no supporting evidence that either history manipulation operations or development parallel to a PR have an effect on the PR's build outcome.

[2]https://help.github.com/articles/about-pull-request-merges (2017-02-10)

Build history: Our data processing approach stratifies data based on the previous build result. Consequently, analyses for build history metrics are performed on the unstratified data.

PB: Because a build can have more than one predecessor in the build topology, we consider the previous build *failed* if at least one of the previous builds failed. We observe that the χ^2-test reveals a significant effect of the previous build result in every project. In ten cases, at least 50% (up to 80%) of all failed builds follow a previous build failure. Also, builds clearly pass more frequently if the previous builds also passed.

BC: The Mann–Whitney test for BC $k = 10$ reveals a significant effect for every project in our dataset. We observe that, in every project, there are periods of high stability or instability. It is apparent that build failures often occur consecutively. A bad build climate indicates that there is a period of build instability which has to be addressed.

DLF: The χ^2-test reveals eight cases where DLF has a significant influence on the build outcome. This result is closely related to that of PB, and we observe that most failures happen consecutively on the same or the next day.

Build failures mostly occur consecutively. Phases of build instability perpetuate failures.

D. Discussion

Our analysis of impact factors focuses on two main categories: process metrics and CI metrics. Using the comprehensive set of extracted metrics, we are able to confirm some results of previous research (e.g., that complex changes tend to be more error prone, or that previous build results are a strong predictor for build failures [10]), but also provide empirical evidence to generate some rather unexpected results. For example, we could not find evidence for the intuitive notion that experienced developers produce fewer errors. In fact, our results suggest that developers with less activity in a repository cause fewer build failures. Furthermore, the analysis of some metrics (e.g., FT or BT) also provided insights beyond statistical significance levels. With these, we have uncovered that a considerable amount of builds fail without apparent external influence. Such build failures cause noise in data, thereby skewing statistics and minimizing the confidence of empirical analyses. A key takeaway for practitioners here is to maintain a stable build environment and to identify and rapidly stabilize tests which are subject to erratic behavior.

Based on the observation that there is often a relationship between two subsequent builds b_1/b_2, we differentiate between *breaking* (passed/failed), *broken* (failed/failed), *fixing* (failed/passed), and *passing* (passed/passed) builds. Since our analyses show that the outcome of previous builds is one of the strongest predictors for the next build result, we argue that it is imperative to consider historical data and the progression of build results in future research. Approaches in the past have mostly worked under the assumption of a linear build history. However, with the proliferation of the PR-based development model, CI build data has inherited the non-linear nature of VCS data. The linking of CI-build and VCS data is at the core of our approach, and is, by itself, a non-trivial task. Topology mapping addresses the challenges arising from local versus remote repository clones, GitHub's PR workflow, loss of historical data due to Git's history manipulation operations, etc. Although efforts have been made to provide comprehensive and open CI and VCS datasets (e.g., GHTorrent or TravisTorrent), some analyses still require more detailed data not attainable via these datasets.

VI. Threats to Validity

In this section, we discuss the most important threats to the validity of our study. We have identified threats to the construct, internal, and external validity.

Construct Validity: refers to the degree to which a test measures what it claims to be measuring. Our study of RQ1 required the semi-automated classification of builds into error categories. It is possible that this process has resulted in some misclassifications. To mitigate this threat, the first author has manually inspected a sampling of classifications and iterated until no further misclassifications could be found. Furthermore, our analysis of when in the timeline of a build specific errors tend to happen is threatened by the fact that absolute build times tend to evolve over the lifetime of a project. That is, an error occurring after five minutes may be "late in the build" at some point during a project's lifetime, and "early" at another. RQ2 required us to select a finite set of process and CI metrics from the vast body of previously published and used metrics. We have strived to choose a comprehensive set of existing metrics, and used definitions that followed previously published research. However, there is still the construct threat that, e.g., change scattering as defined by Hassan et al. [30] is not a valid measure for measuring how changes are scattered across files.

Internal Validity: describes the extent to which the conclusions of the study are justified by the data. A major threat to the internal validity of our results for RQ2 is that our research methodology does not allow us to establish causal relationships between individual process and CI factors and build errors. Instead, we were only able to establish correlations between metrics and build outcomes. Our conclusions should be interpreted in this light. Second, readers should note that we did not completely stratify variables before correlation analysis, as doing so would have resulted in very small sample sizes and limited statistical power. To mitigate this threat, we have manually inspected the data and selectively stratified the data across dimensions with unusually high influence on the outcomes. For instance, an initial analysis has shown that the previous build outcome is by far the most accurate predictor of build outcomes (i.e., if the build is currently broken, the next build is overproportionally more likely to also break), we have stratified the data based on the current build status.

External Validity: refers to the validity of generalized results. In this regard, it is important to note that we have only considered Java-based projects using Travis-CI and either the Maven or Gradle build systems. Further, we have only investigated projects using Git and hosted on GitHub. Our results should not be generalized to other version control workflows, particularly to ones that are centralized. Finally, we have explicitly chosen high-profile projects with many contributors for our study. This raises the question whether our results are generalizable for smaller OSS, specifically those that have less well-established CI processes and less builds.

VII. Conclusion

Despite the wide-spread adoption of CI, little is known about the variety and frequency of build errors, or different factors that influence build results. In this paper we presented qualitative and quantitative evidence on CI build failures based on publicly available data gathered from 14 Java OSS projects.

Our results are aligned along two main research questions. First, we systematically analyzed build log data from which we elicited 14 different error categories. Analysis of the temporal dimension showed that roughly 30% errors occur in the first half of an average build execution, and that testfailures are a major threat to the CI feedback process. Second, we performed a comprehensive analysis of impact factors based on a set of 11 process metrics and 5 CI metrics that we defined based on existing literature. Among the most influential factors are build stability, change complexity, and author experience.

Our results indicate some important directions for the future of CI. Most importantly, software developers should eliminate flaky tests and keep build systems healthy. Developers of CI systems should address common issues, such as broken interactions with VCS repositories. We also envision that future research will focus more on topological aspects of build histories. Furthermore, we believe that our metrics can be used not only for retrospective analysis, but for actual prediction of build results. We actively pursue this in our ongoing work.

Acknowledgments

We want to thank Florian Rosenberg for providing valuable input in the early stage of this research. This paper is supported by TU Wien research funds and by the Commission of the European Union within the CREMA H2020-RIA project (Grant agreement no. 637066).

References

[1] L. Dabbish, C. Stuart, J. Tsay, and J. Herbsleb, "Social Coding in GitHub: Transparency and Collaboration in an Open Software Repository," in *Proceedings of the ACM Conference on Computer Supported Cooperative Work (CSCW'12)*. ACM, 2012, pp. 1277–1286.

[2] C. Brindescu, M. Codoban, S. Shmarkatiuk, and D. Dig, "How Do Centralized and Distributed Version Control Systems Impact Software Changes?" in *Proceedings of the 36th International Conference on Software Engineering (ICSE'14)*. ACM, 2014, pp. 322–333.

[3] G. Gousios, M. Pinzger, and A. v. Deursen, "An Exploratory Study of the Pull-based Software Development Model," in *Proceedings of the 36th International Conference on Software Engineering (ICSE'14)*. ACM, 2014, pp. 345–355.

[4] B. Vasilescu, Y. Yu, H. Wang, P. Devanbu, and V. Filkov, "Quality and Productivity Outcomes Relating to Continuous Integration in GitHub," in *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE'15)*. ACM, 2015, pp. 805–816.

[5] N. Kerzazi, F. Khomh, and B. Adams, "Why Do Automated Builds Break? An Empirical Study," in *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Sept 2014, pp. 41–50.

[6] H. Seo, C. Sadowski, S. Elbaum, E. Aftandilian, and R. Bowdidge, "Programmers' Build Errors: A Case Study (at Google)," in *Proceedings of the 36th International Conference on Software Engineering (ICSE'14)*, 2014, pp. 724–734.

[7] M. Beller, G. Gousios, and A. Zaidman, "Oops, My Tests Broke the Build: An Explorative Analysis of Travis CI with GitHub," in *Proceedings of the 14th Working Conference on Mining Software Repositories (MSR)*, 2017.

[8] T. Wolf, A. Schroter, D. Damian, and T. Nguyen, "Predicting Build Failures Using Social Network Analysis on Developer Communication," in *Proceedings of the International Conference on Software Engineering (ICSE'09)*, 2009, pp. 1–11.

[9] I. Kwan, A. Schroter, and D. Damian, "Does Socio-Technical Congruence Have an Effect on Software Build Success? A Study of Coordination in a Software Project," *IEEE Transactions on Software Engineering*, vol. 37, no. 3, pp. 307–324, May 2011.

[10] A. E. Hassan and K. Zhang, "Using Decision Trees to Predict the Certification Result of a Build," in *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*, Sept 2006, pp. 189–198.

[11] P. Rodríguez, A. Haghighatkhah, L. Lwakatare, S. Teppola, T. Suomalainen, J. Eskeli, T. Karvonen, P. Kuvaja, J. Verner, and M. Oivo, "Continuous Deployment of Software Intensive Products and Services: A Systematic Mapping Study," *Journal of Systems and Software*, 2016.

[12] D. Ståhl and J. Bosch, "Modeling Continuous Integration Practice Differences in Industry Software Development," *Journal of Systems and Software*, vol. 87, pp. 48–59, Jan. 2014.

[13] C. Vassallo, F. Zampetti, D. Romano, M. Beller, A. Panichella, M. D. Penta, and A. Zaidman, "Continuous Delivery Practices in a Large Financial Organization," in *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*, 2016.

[14] O. Hanappi, W. Hummer, and S. Dustdar, "Asserting Reliable Convergence for Configuration Management Scripts," in *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'16)*, 2016.

[15] G. Schermann, J. Cito, P. Leitner, U. Zdun, and H. Gall. (2015) An Empirical Study on Principles and Practices of Continuous Delivery and Deployment. PeerJ Preprints 4:e1889v1 https://doi.org/10.7287/peerj.preprints.1889v1.

[16] S. Neely and S. Stolt, "Continuous Delivery? Easy! Just Change Everything (Well, Maybe It Is Not That Easy)," in *Agile Conference (AGILE'13)*, Aug 2013, pp. 121–128.

[17] Y. Yu, H. Wang, V. Filkov, P. Devanbu, and B. Vasilescu, "Wait for It: Determinants of Pull Request Evaluation Latency on GitHub," in *Proceedings of the 2015 IEEE/ACM 12th Working Conference on Mining Software Repositories (MSR)*, May 2015, pp. 367–371.

[18] G. Gousios, A. Zaidman, M.-A. Storey, and A. van Deursen, "Work Practices and Challenges in Pull-based Development: The Integrator's Perspective," in *Proceedings of the 37th International Conference on Software Engineering (ICSE'15)*, 2015, pp. 358–368.

[19] S. McIntosh, B. Adams, and A. Hassan, "The Evolution of Java Build Systems," *Empirical Software Engineering*, vol. 17, no. 4, pp. 578–608, 2012.

[20] M. Cataldo and J. D. Herbsleb, "Factors Leading to Integration Failures in Global Feature-oriented Development: An Empirical Analysis," in *Proceedings of the 33rd International Conference on Software Engineering (ICSE'11)*, 2011, pp. 161–170.

[21] C. Bird and T. Zimmermann, "Predicting Software Build Errors," 2017, US Patent Grant US9542176 B2.

[22] G. Gousios and D. Spinellis, "GHTorrent: Github's Data From a Firehose," in *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories (MSR)*, June 2012, pp. 12–21.

[23] M. Beller, G. Gousios, and A. Zaidman, "TravisTorrent: Synthesizing Travis CI and GitHub for Full-Stack Research on Continuous Integration," in *Proceedings of the 14th Working Conference on Mining Software Repositories (MSR)*, 2017.

[24] C. Bird, P. C. Rigby, E. T. Barr, D. J. Hamilton, D. M. German, and P. Devanbu, "The Promises and Perils of Mining Git," in *Proceedings of the 6th IEEE International Working Conference on Mining Software Repositories (MSR)*, 2009, pp. 1–10.

[25] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. German, and D. Damian, "The Promises and Perils of Mining GitHub," in *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR)*. ACM, 2014, pp. 92–101.

[26] D. German, B. Adams, and A. Hassan, "Continuously Mining Distributed Version Control Systems: an Empirical Study of How Linux Uses Git," *Empirical Software Engineering*, vol. 21, no. 1, pp. 260–299, 2016.

[27] A. Strauss, J. Corbin *et al.*, *Basics of Qualitative Research*. Newbury Park, CA: Sage, 1990, vol. 15.

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

[29] K. Herzig and A. Zeller, "The Impact of Tangled Code Changes," in *Proceedings of the 10th Working Conference on Mining Software Repositories (MSR'13)*, May 2013, pp. 121–130.

[30] A. E. Hassan, "Predicting Faults Using the Complexity of Code Changes," in *Proceedings of the International Conference on Software Engineering (ICSE'09)*, 2009, pp. 78–88.

[31] L. Madeyski and M. Jureczko, "Which Process Metrics Can Significantly Improve Defect Prediction Models? An Empirical Study," *Software Quality Journal*, vol. 23, no. 3, pp. 393–422, 2015.

[32] C. Macho, S. McIntosh, and M. Pinzger, "Predicting Build Co-changes with Source Code Change and Commit Categories," in *Proceedings of the 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER'16)*, vol. 1, March 2016, pp. 541–551.

[33] J. Śliwerski, T. Zimmermann, and A. Zeller, "When Do Changes Induce Fixes?" in *Proceedings of the International Workshop on Mining Software Repositories (MSR'05)*, 2005, pp. 1–5.

[34] J. Eyolfson, L. Tan, and P. Lam, "Do Time of Day and Developer Experience Affect Commit Bugginess?" in *Proceedings of the 8th Working Conference on Mining Software Repositories (MSR'11)*, 2011, pp. 153–162.

[35] E. Shihab, C. Bird, and T. Zimmermann, "The Effect of Branching Strategies on Software Quality," in *Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM'12)*, 2012, pp. 301–310.