Studying the Impact of Noises in Build Breakage Data

Article *in* IEEE Transactions on Software Engineering · August 2019

CITATIONS
0

READS
30

4 authors:

Taher Ahmed Ghaleb
Queen's University
19 PUBLICATIONS **36** CITATIONS
SEE PROFILE

Daniel Alencar da Costa
University of Otago
27 PUBLICATIONS **120** CITATIONS
SEE PROFILE

Ying Zou
Queen's University
154 PUBLICATIONS **1,947** CITATIONS
SEE PROFILE

Ahmed E. Hassan
Queen's University
375 PUBLICATIONS **8,906** CITATIONS
SEE PROFILE

Studying the Impact of Noises in Build Breakage Data

Taher Ahmed Ghaleb, *Member, IEEE*, Daniel Alencar da Costa, *Member, IEEE*, Ying Zou, *Senior Member, IEEE*, Ahmed E. Hassan, *Fellow, IEEE*

Abstract—Much research has investigated the common reasons for build breakages. However, prior research has paid little attention to builds that may break due to reasons that are unlikely to be related to development activities. For example, Continuous Integration (CI) builds may break due to timeout or connection errors while generating the build. Such kinds of build breakages potentially introduce noises to build breakage data. Not considering such noises may lead to misleading results when studying CI builds. In this paper, we propose three criteria to identify build breakages that can potentially introduce noises to build breakage data. We apply these criteria to a dataset of $350,246$ builds from 153 GitHub projects that are linked with Travis CI. Our results reveal that 33% of the build breakages are due to environmental factors (e.g., errors in CI servers), 29% are due to (unfixed) errors in previous builds, and 9% are due to build jobs that were later deemed by developers as noisy (there is an overlap of 17% between these three types of breakages). We measure the impact of noises in build breakage data on modeling build breakages. We observe that models that use uncleaned build breakage data can lead to misleading associations between build breakages and development activities (e.g., the role of developer). However, such associations could not be observed after eliminating noisy build breakages. Moreover, we replicate a prior study that investigates the association between build breakages and development activities using data from 14 GitHub projects. We observe that some observations reported by the prior study (e.g., pull requests cause more breakages) do not hold after eliminating the noises from build breakage data.

Index Terms—Continuous Integration; CI build breakages; Noisy data; Mining software repositories; Empirical software engineering

1 Introduction

Continuous Integration (CI) is a software development practice that allows software development teams to generate software builds more quickly and periodically (e.g., daily or hourly) [1]. CI brings many advantages, such as the earlier identification of code integration errors [2, 3]. CI builds consist of multiple jobs, each of which runs on a different runtime environment. A CI build can break if any of its jobs breaks. A build breakage may occur due to several reasons, such as configuration errors, installation errors, compilation errors, or test failures [4, 5]. Much research has been devoted to studying breakages of CI builds. Rausch *et al.* [4] explored the common reasons for build breakages in CI and how frequent they break CI builds. Vassallo *et al.* [5] studied how build breakages in CI differ in open source and industrial projects. Researchers also studied the possibility of predicting build breakages [4, 6–10]. Other studies investigated the factors that share a strong association with CI build breakages [4, 7, 9, 11, 12]. These studies found that process metrics (e.g., commit complexity and file types), developer roles (i.e., core or peripheral), and build breakage history are among the factors with the strongest association with the likelihood of a build breakage.

- *Taher Ahmed Ghaleb is with the School of Computing, Queen's University, Canada. E-mail: taher.ghaleb@queensu.ca*
- *D. da Costa is with Department of Information Science, University of Otago, Dunedin, New Zealand. E-mail: danielcalencar@otago.ac.nz*
- *Y. Zou is with Department of Electrical and Computer Engineering, Queen's University, Canada. E-mail: ying.zou@queensu.ca*
- *Ahmed E. Hassan is with the Software Analysis and Intelligence Lab (SAIL), School of Computing, Queen's University, Canada. E-mail: ahmed@cs.queensu.ca*

However, prior research has paid little attention to builds that may break due to reasons that are unlikely to be related to the activities of developers. For example, a build may break due to environmental factors, such as timeouts, connection resets, or memory allocation errors. Zhao *et al.* [13] studied the types of build breakages in 250 open source projects spanning a 10-month period. They observed a decreasing trend over time in the number of build breakages that are caused due to timeouts or missing dependencies. Rausch *et al.* [4] studied build breakages in 14 open source Java projects and observed a non-negligible amount of noise in build breakage data. Gallaba *et al.* [12] found that builds may have ignored breakages (i.e., do not impact the build status) or breakages that are not caused by the building tool (e.g., Maven[1]). Other studies have investigated flaky tests, where tests are broken non-deterministically [14–17].

Despite the valuable insights reported by existing studies, there was no discussion as to whether build breakages are really caused by the developers who triggered the builds. In particular, build breakages that are caused due to environmental errors should not be used to study the association of build breakages with development activities. Additionally, prior studies did not consider that builds can break due to (unfixed) errors introduced in previous builds or due to noisy build jobs that were excluded from the build later. Although prior studies discussed noises that may exist in build breakage data, the impact of modeling build breakages using clean data (i.e., after removing the noises) has not been investigated. Studying build breakages

[1]http://maven.apache.org

without considering the aforementioned factors may lead to misleading observations. For example, development activities (e.g., being a core developer) may be deemed as the culprits for breaking CI builds when it is not the case.

In this paper, we (i) study breakages that can potentially introduce noises in build breakage data and (ii) measure the impact of such noises on the observations reported in the literature. Noisy build breakages may occur due to (a) environmental factors (e.g., errors in CI servers), (b) previous (unfixed) build errors, or (c) build jobs that are later deemed by developers as noisy. We conduct an empirical study on data collected from TravisTorrent [18], a commonly used dataset to study CI build breakages [5, 9–12, 19]. Our dataset contains $350,246$ CI builds from 153 GITHUB projects linked to TRAVIS CI, a cloud-based CI service. In our study, we address the following research questions:

RQ$_1$: What is the proportion of build breakages that can introduce noises in build breakage data? It is important for researchers and practitioners to identify the proportion of build breakages data that can negatively affect their analyses. We define three criteria for identifying CI build breakages that can taint the results of the prior research. Our results show that 33% of broken builds are impacted by environmental factors that are unlikely to be related to development activities. We identify 60 (sub)categories of environmental build breakages. In addition, we observe that 29% of broken builds are potentially cascading (unfixed) errors from previous builds. Finally, we observe that 9% of broken builds are primarily broken by jobs that were later deemed by developers as noisy. We identify an overlap of 17% between the identified types of breakages. For example, jobs with environmental breakages, which are allowed to fail later, may generate cascading breakages (in addition to allowed breakages).

RQ$_2$: What is the impact of using uncleaned build data on modeling build breakages? We measure the impact of using uncleaned build breakage data (i.e., containing noisy build breakages) on modeling CI build breakages and the subsequent analyses. We construct two mixed-effects logistic models using both the original (uncleaned) data and our cleaned data. We use the build status (i.e., *passed* or *broken*) as the dependent variable in both models. We also use the same metrics as independent variables in both models. We observe that using uncleaned build data to model CI build breakages has a considerable impact on the performance of models (i.e., an AUC reduction of 6%). In addition, we observe that models that use uncleaned build breakage data can lead to misleading conclusions. For example, in the model that uses uncleaned data, build breakages are significantly associated with core developers, whereas the model fitted using the cleaned data shows no evidence for such an association. Furthermore, we observe that observations reported by prior research (e.g., pull requests cause more breakages) may not hold if noisy build breakages are filtered out from build breakage data.

In summary, this paper makes the following contributions:

- Three criteria to identify build breakages that can potentially corrupt the data used by the prior research.
- A catalogue of environmental build breakages and their frequency of occurrences in 153 GITHUB projects that are linked with TRAVIS CI.
- An empirical study of the impact of using uncleaned build breakage data on the observations reported by prior research studying build breakages.

Paper organization. The rest of this paper is organized as follows. Section 2 discusses build breakage examples that motivate our work. Section 3 introduces the experimental setup of our study. Section 4 describes the criteria to identify build breakages that can potentially introduce noises to build breakage data. Section 5 presents the results and findings of our study. Section 6 discusses the implications of our findings for researchers and practitioners. Section 7 describes the threats to the validity of our results. Section 8 outlines the prior research related to our study. Finally, Section 9 concludes the paper and outlines avenues for future work.

2 MOTIVATING EXAMPLES

Prior studies have relied heavily on the build statuses generated by TRAVIS CI to perform the analyses [4, 5, 8, 9, 19–21]. For example, researchers have proposed models to study the relationship between the build status (i.e., *passed* or *failed*) and several metrics that are related to the development process (e.g., committing code changes by core developers or at night) [4, 10, 12]. However, in this section, we illustrate three types of build breakages that could potentially impact the results reported by these models.

Environmental breakages. *Environmental* breakages are caused by the environment that generates the builds (e.g., the TRAVIS CI servers). Examples of *Environmental* breakages are connection timeouts or exceeded time limits when running commands or tests. Such breakages are unlikely to be caused by developers. A real example from the `Diaspora` project[2] is illustrated in Table 1. As shown in the table, a sequence of builds were generated from *September 7th, 2011* to *September 8th, 2011*. At a first glance, one might assume that *developer A* broke the build (i.e., build `192`) and, after several failed attempts to fix the breakage (i.e., at builds `193`-`204`), *developer E* finally fixed the breakage (i.e., at build `208`). Nevertheless, by taking a deeper look at the logs of the builds listed in Table 1, we uncover a different story. In reality, *developer A* did not break the build, since the real reason for the breakage of build `192` was a server connection timeout (see the "Detailed Status" column of Table 1). In fact, *developer B* broke the build when generating build `194`. Moreover, *developer E* did not fix the breakage. In fact, *developer A* fixed the breakage when generating build `202`. However, build `202` was broken due to another connection timeout. Interestingly, the two consecutive builds (builds `203` and `204`) were not really broken. Both builds received a broken status due to a bug on the TRAVIS CI service. As a result, all the jobs of builds `203` and `204` have passed, however, TRAVIS CI wrongly generated broken statuses.

Cascading breakages. *Cascading* breakages occur because of inherited mistakes from previous builds. For example, considering builds ranging from `194` to `201`, one might assume that they all represent different breakages. However, by analyzing their logs, we observe that only build `194` has a new error when compared to its predecessor (see "Detailed status", in Table 1). The remaining builds (i.e.,

[2]https://github.com/diaspora/diaspora

TABLE 1: An excerpt of CI builds in the Diaspora project

Build No.	Developer*	Timestamp	Duration	Status	Broken jobs	Detailed status	Breakage reason
191	C	Sept. 7, 2011 - 19:46	24 *mins*	passed	0	Successful	
192	A	Sept. 7, 2011 - 20:05	24 *mins*	errored	1	Timeout	No actual breakage
193	A	Sept. 7, 2011 - 20:27	23 *mins*	errored	1	Timeout	No actual breakage
194	B	Sept. 7, 2011 - 20:49	11 *mins*	errored	6	***Failure/Error: Resque.should_receive(:enqueue)***	First occurrence of the breakage
195	A	Sept. 7, 2011 - 21:19	11 *mins*	errored	6	***Failure/Error: Resque.should_receive(:enqueue)***	Not related to the breakage
196	C	Sept. 7, 2011 - 21:40	12 *mins*	errored	6	***Failure/Error: Resque.should_receive(:enqueue)***	Not related to the breakage
198	B	Sept. 7, 2011 - 23:01	11 *mins*	errored	6	***Failure/Error: Resque.should_receive(:enqueue)***	Failed Fix attempt
199	D	Sept. 8, 2011 - 00:03	10 *mins*	errored	6	***Failure/Error: Resque.should_receive(:enqueue)***	Not related to the breakage
200	B	Sept. 8, 2011 - 17:26	11 *mins*	errored	6	***Failure/Error: Resque.should_receive(:enqueue)***	Failed Fix attempt
201	B	Sept. 8, 2011 - 17:48	11 *mins*	errored	6	***Failure/Error: Resque.should_receive(:enqueue)***	Failed Fix attempt
202	A	Sept. 8, 2011 - 18:01	29 *mins*	errored	1	Timeout	Successful Fix attempt
203	B	Sept. 8, 2011 - 19:01	23 *mins*	errored	0	Errored, but all jobs have passed	Not related to the breakage
204	E	Sept. 8, 2011 - 19:03	23 *mins*	errored	0	Errored, but all jobs have passed	Not related to the breakage
208	E	Sept. 8, 2011 - 22:00	22 *mins*	passed	0	Successful	

* Developer names are encoded

195-204) share the same error as build 194, meaning that these builds are unlikely to represent new breakages. In other words, the newly pushed commits are unlikely to be the cause of the breakage. Therefore, researchers should be careful when including *Cascading* breakages in their models if the main goal is to predict new build breakages.

Allowed breakages. *Allowed* breakages occur when builds are broken by jobs that are disregarded by developers and later deemed as noisy. Builds can have jobs with integration environments that are under experimentation. If these jobs recurrently break the builds, developers may decide later to exclude such jobs in one of the three forms: (a) marking the job as *allow_failures*, (b) completely removing the job from the build, or (c) changing the job configuration to a different environment. Therefore, builds that are broken only because of such jobs could have passed if such jobs were excluded earlier by developers. To illustrate this type of breakage, we show in Table 2 an example of the Puma[3] project. Table 2 shows a sequence of 17 builds that were generated from *December 3rd, 2013* to *January 25th, 2014*. As shown in Table 2, there are 14 builds that are broken due to different job breakages. We observe that job C and job D are recurrently broken and, as a result, 14 builds are broken. However, in build 473, both jobs are marked as *allow_failures*, which allowed build 473 to pass. As a consequence, build 456 and builds 458-472 could also pass if either jobs or both of them were excluded earlier. Build 471 could also pass, since its status was partially impacted by both jobs and partially by jobs E and F, which were removed in build 472. However, builds 453 and 454 would not passed, since they were broken due to two other broken jobs (i.e., A and B).

TABLE 2: An excerpt of CI builds in the Puma project

Build No.	Status	Broken jobs*	Build configuration action
450	passed		
453	failed	A & B & C & D	
454	failed	A & B & C & D	
456	failed	C & D	
457	passed		
458	errored	D	
459	failed	D	
461	failed	C & D	
462	failed	C & D	
466	errored	C & D	
467	failed	C & D	
468	failed	C & D	
469	failed	C & D	
470	failed	D	
471	errored	C & D & E & F	*Jobs added*: E & F & G
472	errored	C & D	*Jobs removed*: E & F
473	passed		*Jobs marked as allow_failures*: C & D

* A: Ruby: 1.9.3, B: Ruby: 2.0.0, C: Ruby: jruby-19mode, D: Ruby: rbx, E: Ruby: jruby-18mode, F: Ruby: 1.8.7, G: Ruby: 2.1.0

3 Experimental Setup

This section presents the experimental setup and the steps of collecting and processing the data for our studied RQs.

3.1 Data collection

Fig. 1 shows an overview of our empirical study. We collected data from TravisTorrent [18]. TravisTorrent, in its 11.1.2017 release, stores CI build breakage data of 1,283 projects. Prior research regarding CI builds relies heavily on the TravisTorrent data when conducting empirical studies on CI [4, 8, 9, 19]. TravisTorrent contains projects that are written in three programming languages: Ruby, Java, and JavaScript. We filter the studied projects using two criteria to ensure that we have sufficient data for our analyses. Some projects in TravisTorrent do not actively generate CI builds (e.g., toy projects), which leads to a small number of builds for such projects. Therefore, we select projects with at least 1,000 unique builds in TravisTorrent. Using this criterion, we obtain 154 projects (66 Java, 87 Ruby, and one Javascript). We exclude the Javascript project, since it cannot be used as a representative of the Javascript language. Our dataset contains builds that are triggered by different events: git pushes (83.69%), pull requests (15.98%), API requests (0.32%) and scheduled Cron jobs (0.01%).

Due to space restrictions, we show a complete overview of the 153 studied projects in our online appendix [22]. These projects are of a range of domains, including, but not limited to, applications, programming languages, and tools. The number of CI builds of our selected projects is 350,246, which represents about 52% of the total builds in TravisTorrent. The total number of build jobs in our dataset is 1,927,239 with an average of 5 jobs per build. Each build job runs (and may pass or break) independently from each other. We use the TravisPy[4] API to collect more metrics about build jobs (e.g., the job status and configuration). We also use the boto3[5] API to download the plain-text build logs from Amazon S3, the storage backend of Travis CI. . Moreover, we compute additional metrics for builds (e.g., the number of configuration files that are modified) using the commits that trigger the builds in our dataset.

[3]https://github.com/puma/puma

[4]https://pypi.python.org/pypi/TravisPy/0.1.1

[5]https://aws.amazon.com/sdk-for-python

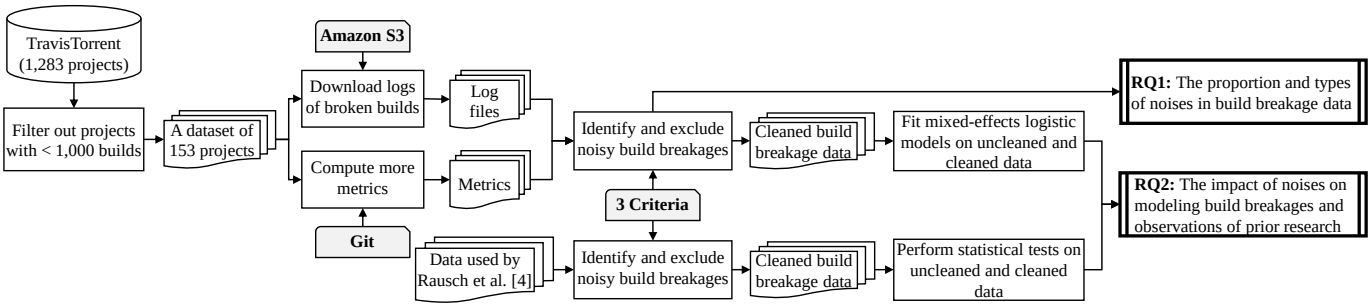


Fig. 1: Overview of our empirical study of the noises in build breakage data

3.2 Data processing

We sort builds of our studied projects in a chronological order according to the triggering time of each build. We also exclude *Canceled* builds because they are incomplete and do not give a clear picture about whether they would pass or fail if they had not been interrupted. Considering that a build may contain multiple jobs, the breakage could be partial (i.e., some jobs are broken) or complete (i.e., all jobs are broken). In our analyses, we only consider the broken jobs of broken builds. We use the build logs for such jobs to investigate the categories of environmental build breakages. We include *allow_failures* jobs in our analyses at the job level. However, we disregard the *allow_failures* jobs when we label a breakage as *Allowed* breakage (i.e., at the build level).

4 Criteria to Identify Noisy Build Breakages

In this section, we explain how we apply three criteria to identify (1) *Environmental* breakages, (2) *Cascading* breakages, and (3) *Allowed* breakages in the studied projects. After we apply the three criteria, we clean our studied dataset by excluding builds that have the three types of breakages. A replication package of the heuristics used in this Section is available in our online appendix [22].

4.1 Identifying *Environmental* breakages

To identify *Environmental* build breakages, we analyze the raw build log files for all broken jobs that belong to broken builds in our dataset. The total number of broken jobs is $321,855$. We label build logs of broken jobs based on the error or failure messages found. Our log labeling process is semi-automated as it involves manual and automated analyses of build logs. In the manual analysis, we scan build logs to find any error messages that correspond to the build breakage. TRAVIS CI may recover from errors but keeps a record of such errors in the build log. Therefore, we label build logs according to the last logged error message. For example, a build may experience a dependency installation error but, because failed commands may rerun several times, the command that installs the dependency may succeed later. After a successful dependency installation, however, the build may experience an environmental error (e.g., a timeout error). In such a case, we ignore the message of the recovered error and label the build log according to the last error message (i.e., the timeout error).

Build logs contain heterogeneous and inconsistent error messages. A certain build error may be reported using different forms of error messages. For example, "`The command xyz exited with 8`", "`ERROR 404: Not Found`", and "`Error: 404 Client Error`" are different forms of error messages to report a server connection error. Therefore, we employ a thematic analysis [23] to manually identify themes (i.e., categories) of *Environmental* build breakages. Two of the co-authors perform the manual analysis using a statistically significant sample of 384 out of $321,855$ build logs (a confidence level of 95% and a confidence interval of $\pm 5\%$). We use open coding [24] to produce an initial set of categories of *Environmental* breakages. In addition, we tag each build log with one of three labels: *Developer*, *Environmental*, and *Suspicious*. We assign the *Developer* label to the jobs in which the build breakage is most likely caused by errors introduced by developers. We assign the *Environmental* label to the jobs in which the build breakage is most likely caused by environmental factors. We assign the *Suspicious* label to the jobs in which we could not identify the underlying cause of breakage (e.g., empty logs, accidentally trimmed logs, or terminated logs with a successful exit code). Each build log is assigned a single label, which represents the label of the broken job. We conduct consensus meetings to resolve all labeling and categorization disagreements.

After the manual analysis, we use the identified labels and categories to generate heuristics (i.e., python scripts [22]) that automate the process of identifying error messages, categorizing, and labeling the build logs in our dataset. After the automated labeling, the two co-authors perform a manual analysis of another statistically significant sample (i.e., additional 384 build logs) to validate the labels and categories generated by the automated labeling. We use the Cohen's $kappa$ inter-rater agreement statistic [25] to measure how reliable is the manual validation of the automatically generated labels. To compute the agreement level, we used two codes in which the raters indicate whether there is a *match* or *mismatch* between the automatically generated and manually assigned labels We discuss the cases in which there is (i) a disagreement between the raters or (ii) a mismatch between the manual and automated labeling. We resolve the disagreements using consensus meetings, refine our heuristics, and repeat the same process for the set of unlabeled build logs. We provide more details about the obtained level of agreement in Section $5{-}RQ_1$: Findings.

We approach online resources to identify whether an error messages is related to *Environmental* issues. For example, we study issues related to these error messages on different issue reporting websites (e.g., TRAVIS CI,[6] rvm,[7]

[6]https://github.com/travis-ci/travis-ci/issues
[7]https://github.com/rvm/rvm/issues

and rubygems[8]). Some build breakages require a deeper investigation prior to classifying them as *Environmental*. For example, in the cases where errors are caused due to missing objects (e.g., files, configuration, or gems), we analyze the commits that trigger the broken builds to check whether such objects exist in the repository or not. To do this, we automatically perform a `git diff` command for the commits that triggered the build. Then, we check whether the reported objects are truly missing in that revision. If a missing object is found in the repository, we consider the build breakage as *Environmental*. Otherwise, we consider the build breakage as *Developer*.

Considering that a build may have more than one job, we label builds using the labels of their jobs. We exclude the jobs that are marked as *allow_failures* when we label builds, since such jobs do not break the build if they are broken. We assign the *Developer* label to builds that have at least one broken job with the *Developer* label. We assign the *Environmental* label to builds that have all the broken jobs labeled as *Environmental* or *Suspicious*. Builds that have the *Developer* label are the only builds that we should associate with developers' activities. However, environmentally broken builds cannot be considered as *passed*, since *Developer* errors may potentially occur if such builds are restarted. Therefore, a clean build breakage dataset should contain only build breakages that have the *Developer* label.

4.2 Identifying *Cascading* breakages

To identify *Cascading* build breakages, we sort the builds of each branch of the studied projects in a chronological order based on the build triggering timestamp. Then, we analyze every pair of consecutively broken builds to identify which builds are simply broken because of a former (unfixed) breakage. First, we compare the number of broken jobs between each pair of consecutively broken builds. If both builds have a matching number of broken jobs, we compare the number of errors in all broken jobs. If both builds have a matching number of errors, we compare their error messages. If both builds have identical error messages, we consider that the current build breakage is broken due to existing (unfixed) errors introduced by the commits that triggered by the previously broken build. However, we cannot assume that the commits of the currently broken build are free of errors, since errors of former commits may hinder showing the errors that might be introduced by current commits.

4.3 Identifying *Allowed* breakages

To identify *Allowed* build breakages, we use the chronologically sorted builds to identify the broken jobs that are later removed from the builds or marked as *allow_failures*. The excluded jobs may contain errors that are deemed by developers to be noisy or unimportant at a particular point of development. We identify unique build jobs by two attributes: the language runtime version (e.g., `Ruby: 2.1`) and the integration environment (e.g., `DB=mysql2`). If a build has multiple broken jobs, we consider the build to have an *Allowed* breakage if all the broken jobs are later excluded from the build. Builds in which only a subset of the broken jobs are excluded (i.e., other jobs are fixed by developers) are considered to have *Developer* breakages.

[8]https://github.com/rubygems/rubygems/issues

5 Experimental Results

This section discusses the results of our research questions.

5.1 RQ_1: What is the proportion of build breakages that can introduce noises in build breakage data?

Motivation: Despite the research invested on build breakages [4, 5, 26–28], there is a lack of awareness about *Environmental*, *Cascading*, and *Allowed* breakages. These types of build breakages can potentially taint the current conclusions about build breakages that exist in the literature. Therefore, it is important to study the proportion and frequency of build breakages that can potentially introduce noises in build breakage data.

Approach: To address this RQ, we analyze the results obtained from the build breakage categorization and labeling using the three proposed criteria proposed in Section 4. We use the breakage labels at both the job and build levels. We report statistics about the proportion of *Environmental*, *Cascading*, and *Allowed* breakages in our studied projects. In addition, we report the ratio and frequency of each (sub)category of *Environmental* build breakages. We also perform additional manual analyses to gain more insights about the identified noisy build breakages.

Findings: ***About*** 55% ***of build breakages are environmental, cascading, and/or allowed.*** Fig. 2 shows the ratios of *Environmental*, *Cascading*, and *Allowed* build breakages in our dataset. As Fig. 2 indicates, there is an overlap of 17% between the three types of breakages. For example, ignoring environmentally broken builds may cascade the breakage to the next builds. Likewise, jobs with *Environmental* breakages, which are allowed to fail later, may generate *Cascading* breakages (in addition to *Allowed* breakages).

Environmental build breakages occur in 39% ***of the broken jobs.*** We observe that 39% of the analyzed broken jobs (i.e., $126,673$ out of $321,855$) are impacted by environmental factors. At the build level, we observe that 33% of broken builds in the studied projects (a median of 30%) are broken due to environmental factors. Approximately, one-third of environmentally broken builds experienced test failures, suggesting that such builds may contain flaky tests.

The statuses of 29% ***of builds are likely due to cascading breakages.*** Our results show that 29% of builds in our dataset are unlikely to be caused by the developers who triggered those particular builds. In addition, 76% of these builds are triggered by different developers from the developers who first introduced the breakage.

About 9% ***of broken builds are allowed breakages.*** We identify $2,022$ (i.e., 8%) of the unique jobs in the studied projects that are later excluded from broken builds (15% marked as *allow_failures* and 85% completely removed from the builds). Although we find that 33% of broken builds in our dataset contain excluded jobs, only 9% of such builds are primarily broken by those jobs. To understand the reasons behind job exclusion, we manually analyze a statistical sample of 62 commits (a confidence level of 95% and a confidence interval of $\pm 10\%$) in which developers mark jobs as *allow_failures*. We find that developers identify

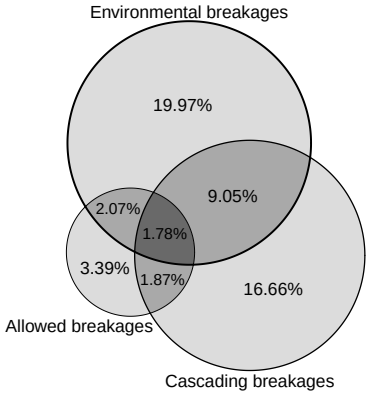


Fig. 2: Ratios of *Environmental*, *Cascading*, and *Allowed* build breakages in our dataset

the excluded jobs to be noisy (or flaky) in 68% of the cases (e.g., "*CI: allow JRuby build to fail, too flaky to be useful*"[9]). In 16% of the cases, developers express their inability to fix the breakage (e.g., "*Sick of jruby breaking travis*"[10]) or indicate that the breakage is unimportant (e.g., "*allow jruby 1.9 mode to fail, cuz I don't care*"[11]). In 16% of the cases, developers do not provide clear reasons why they exclude jobs (e.g., "*[travis-ci] allow failures for truffle for a green build*"[12]). Therefore, we advise researchers to filter builds affected by those jobs out, since the breakages could be due to the abnormal behavior of the excluded jobs.

There are* 60 *(sub)categories of Environmental build breakages. Table 3 presents the ratio and frequency of each *Environmental* (and *Suspicious*) breakage identified in our dataset. Each breakage subcategory represents the potential cause of the *Environmental* breakage. The last column of Table 3 shows the number of projects in which a breakage (sub)category exists. We obtain a strong inter-rater agreement (i.e., $k = 0.82$) with a strong observed agreement of 0.93 in manually validating the automated labeling and categorization of build breakages. We observe that, in 40% of the cases in which there was a disagreement with the automated breakage categorization, build logs are unexpectedly terminated (e.g., *logging stopped progressing*). In such cases, build logs are automatically categorized according to the last logged error messages. However, since the termination of such build logs is abnormal (i.e., the underlying cause of breakage is unknown), we consider them as *Suspicious*. In addition, 32% of disagreement was related to error messages that are associated with other preceding error messages (e.g., '`bundle: command not found`' should precede '`Command failed with status (127)`').

The majority (i.e.,* 78%*) of Environmental build breakages are caused by internal CI errors or issues related to external connections and exceeding limits. Internal CI errors introduce about 39% of the total *Environmental* breakages. We identify 18 reasons that may influence internal CI errors and, consequently, break CI builds. We observe that both the *Connection issues* and *Exceeding limits* categories form around 39% of *Environmental* build breakages. Connection-related breakages mostly occur due to requesting data from external servers. Limits are usually set by TRAVIS CI to prevent builds, commands, and tests from running forever.

TABLE 3: Categories of *Environmental* build breakages

Category	Subcategory	#	%	*Freq.*
Internal CI issues	**Unidentified branch/tree/commit**	29,297	23.10%	132
	Failure to fetch resources	7,658	6.00%	106
	Error building gems	7,107	5.60%	58
	Logging stopped progressing*	1,632	1.30%	90
	Error fetching CI configuration	1,092	0.90%	49
	Error finding gems	753	0.60%	27
	Cannot execute git command	506	0.40%	24
	Multithreading issues	405	0.30%	3
	Unknown TRAVIS CI error	241	0.20%	34
	Build script compilation error	142	0.10%	8
	Cannot access GITHUB	883	0.70%	87
	Empty log*	65	0.00%	20
	Caching problems	44	0.00%	5
	Writing errors	32	0.00%	2
	Remote repository corruption	20	0.00%	2
	Cannot allocate resources	15	0.00%	3
	Storage server offline	9	0.00%	2
	Path issues	7	0.00%	3
	TOTAL	**49,908**	**39.40%**	**152**
Exceeding limits	Stalled build (not output received)	16,798	13.30%	136
	Command execution time limit	3,816	3.00%	78
	Log size limit	3,613	2.80%	52
	Test running limit	2,691	2.10%	59
	Time limit waiting for response	259	0.20%	12
	Job runtime limit	234	0.20%	7
	API rate limit	67	0.00%	3
	TOTAL	**27,478**	**21.70%**	**144**
Connection issues	**Connection timeout**	4,763	3.80%	95
	Connection refused, reset, closed	4,716	3.70%	111
	Server or service unavailable	3,370	2.70%	89
	Broken connection/pipes	3,072	2.40%	23
	Unknown host	2,927	2.30%	16
	Connection credentials error	2,242	1.80%	11
	Remote end hung up unexpectedly	377	0.30%	36
	Network transmission error	177	0.10%	26
	SSL connection error	152	0.10%	17
	Connection, proxy, & sync errors	79	0.10%	9
	SSL certificate error	18	0.00%	9
	TOTAL	**21,893**	**17.30%**	**140**
Ruby & bundler issues	No compatible gem versions	5,521	4.40%	36
	Cannot find, parse, execute gems	2,477	2.00%	51
	Command loading failure	2,839	2.20%	52
	Bad file descriptor	1,515	1.20%	5
	Dependency request error	1,009	0.80%	36
	Bundler not installed	873	0.70%	35
	TOTAL	**14,234**	**11.20%**	**81**
Memory & disk issues	**Out of memory/disk space**	3,395	2.70%	45
	Segmentation fault	1,924	1.50%	41
	Core dump problems	1,301	1.00%	33
	Memory stack error	540	0.40%	18
	Corrupted memory references	7	0.00%	2
	TOTAL	**7,167**	**5.70%**	**84**
Platform issues	**Language installation issues**	1,785	1.40%	59
	Invalid platform	13	0.00%	2
	Unexpected failure	4	0.00%	1
	TOTAL	**1,802**	**1.40%**	**60**
Virtual Machine issues	**Improper VM shut down**	1,114	0.90%	51
	VM creation error	82	0.10%	10
	VM connection problem	17	0.00%	7
	Invalid VM state	12	0.00%	4
	Stalled VM	11	0.00%	6
	TOTAL	**1,236**	**1.00%**	**61**
Accidental abruption	**Build crashes unexpectedly**	2,182	1.70%	39
	TOTAL	**2,182**	**1.70%**	**39**
Buggy build status‡	**Build exited successfully***	425	0.30%	28
	Jobs passing but build broken*	114	0.10%	20
	TOTAL	**539**	**0.40%**	**44**
Database (DB) issues	DB creation quota	159	0.10%	1
	DB connection error	37	0.00%	2
	TOTAL	**196**	**0.20%**	**2**
External bugs	**E.g., interpreter bugs**	38	0.00%	8
	TOTAL	**38**	**0.00%**	**8**
TOTAL ENVIRONMENTAL BREAKAGES		**126,673**	**100%**	**152**

∗ Suspicious build breakages, i.e., the cause of the breakage is unidentified
‡ Issues reported to TRAVIS CI [13,14,15]
– **Bold** subcategories represent breakages that are likely to be exclusively environmental

[9]https://travis-ci.org/rails/rails/builds/125719232
[10]https://travis-ci.org/middleman/middleman/builds/1641894
[11]https://travis-ci.org/teamcapybara/capybara/builds/1923320
[12]https://travis-ci.org/jruby/jruby/builds/94375071

[13]https://github.com/travis-ci/travis-ci/issues/646
[14]https://github.com/travis-ci/travis-ci/issues/891
[15]https://github.com/travis-ci/travis-ci/issues/2533

Developers may reduce the amount of logged information in order to not exceed the TRAVIS CI log size limit of 4 MB. Developers may also customize build configuration to fix certain *Environmental* breakages. For example, the `travis_wait` attribute allows developers to let TRAVIS CI wait for commands that may take longer than the default 10 minutes.[16] Developers may also fix issues related to dependencies or long-running commands. However, in many cases, developers have no other way to fix *Environmental* breakages than restarting builds. In Table 3, we highlight (in **bold**) the (sub)categories of *Environmental* breakages that are most likely to be exclusively outside the ability of developers to control. Moreover, we find that changing build configuration may not always be deemed as the cause or fix of a breakage (more details in Subsection 7.1). Therefore, we consider all (sub)categories of *Environmental* breakages to be noisy.

55% of broken builds in our dataset are impacted by Environmental, Cascading, and/or Allowed breakages. Internal TRAVIS CI *errors, connection issues, and exceeding limits issues cause most, i.e., 78%, of environmental breakages in CI builds.*

5.2 RQ$_2$: What is the impact of using uncleaned build data on modeling build breakages?

Motivation: Uncleaned build breakage data can distort prediction/regression models as broken builds are assumed to be associated with the development process (e.g., triggering commits) and developer metrics. Not considering the breakages that can introduce noises to build breakage data may affect the accuracy of prediction/regression models and drive researchers to misleading conclusions.

Approach: We measure the impact of noises on modeling build breakages using two approaches: (1) we construct two regression models: one model using the *uncleaned data*—the original data containing *Environmental*, *Cascading*, and *Allowed* breakages—and another model using the *cleaned data*; (2) we replicate a prior study [4] that investigates the associations between build breakages and 16 process and CI metrics using build data from 14 projects.

Approach$_1$: We use the build status (i.e., *passed* or *broken*) as a dependent variable in both models. We use a set of metrics (presented in Table 4) as independent variables in both models. These metrics have been used by prior research to study their association with build breakages [4, 8–11]. All these metrics are computed at the commit level. We compute the *previous build status* metric for the uncleaned datasets and recompute such a metric again for the cleaned dataset. For example, assume we have the builds $\{B_1$:'*passed*', B_2:'*broken*', B_3:'*broken*'$\}$ and the breakage of build B_2 is identified to be noisy. In such a case, the previous status of build B_3 is '*broken*' in the uncleaned data but '*passed*' in the cleaned data. We remove highly correlated variables, since they can adversely affect our models [29]. To this end, we use the `varclus` function (from the `rms`[17] R package) that performs the Spearman rank ρ [30]. For each pair of correlated variables that have a correlation of $|\rho| \geq 0.7$, we remove one variable and keep the other in our models. For example, we remove the *Configuration lines added* and *Configuration lines deleted* metrics, since they are highly correlated with the *Configuration files changed*. Similarly, we remove the *Source files changed* metric, since it is highly correlated with the *Files changed* metric. The number of builds in the uncleaned and cleaned datasets are $350,246$ and $296,982$, respectively.

Builds in our datasets are from different languages, projects, and ages. We use these variables as random effects in our models to control (i) the overrepresentation of Ruby projects, (ii) the variation between projects in terms of sizes and domains, and (iii) the potential impact of triggering builds at different stages of CI adoption on the obtained results. To this end, we use the Generalized Linear Mixed Models (GLMM) for logistic regression. GLMM uses mixed (i.e., fixed and random) effects to investigate the variables that are associated with build breakages [31]. This means that our models assume a different random intercept for each category of the random effect [32]. We use the ANOVA test [33] to find the significant variables to model build breakages (i.e., variables that have $Pr(< |\chi^2|)$ less than 0.05). $Pr(< |\chi^2|)$ is the p-value that is associated with the χ^2 test, which shows if our model is statistically different from the same model in the absence of a given independent variable—according to the degrees of freedom in our model. We also use *upward* and *downward* arrows to indicate whether a variable has a direct or an inverse relationship, respectively, with build breakages.

We evaluate the performance of our models using the Area Under the receiver operating characteristic Curve (AUC), the marginal R^2, and the conditional R^2. The AUC evaluates the diagnostic ability of our mixed-effect models to discriminate *broken* builds [34]. AUC is the area below the receiver operating characteristic (ROC) curve created by plotting the true positive rate (TPR) against the false positive rate (FPR) using all possible classification thresholds. The value of AUC ranges between 0 (worst) and 1 (best). An AUC that is greater than 0.5 indicates that the explanatory model outperforms a random predictor. We use the method proposed by DeLong *et al.* [35] to compare the ROC curves of the models. We compute the number of Events Per Variable (EPV) for the uncleaned and cleaned datasets to investigate the likelihood of our models to be overfitting. EPV measures the ratio of the number of build breakages to the number of factors used as independent variables to train the models [36]. The EPV values of the cleaned and uncleaned datasets are $4,726$ and $2,127$, respectively (i.e., EPV ≥ 40). Hence, our models are considered stable and have reliable AUC values (i.e., the optimism is small) [37]. Moreover, our models are unlikely to have overfitting problems [36], since the EPV values are above 10. Higher values of the conditional R^2 indicate that the proportion of variance explained by both fixed and random effects is much higher than the proportion of variance explained by fixed effects only. A high difference between the conditional and marginal R^2 values suggests that random effects significantly help explain the dependent variable (i.e., the build breakage).

One could argue that the differences between the results of the two models could be influenced by the differences in the sizes of the cleaned and uncleaned datasets. To study

[16] https://docs.travis-ci.com/user/common-build-problems/#build-times-out-because-no-output-was-received

[17] https://cran.r-project.org/web/packages/rms/rms.pdf

TABLE 4: Metrics used as independent variables in our models

Metric	Data type	Used by	Description	Effect
Project name	Factor		The name of the project in which a build is triggered	Random
Language	Factor		The programming language of the project in which a build is triggered (Ruby or Java)	
Build age	Factor		The difference (in days) between a build starting date and the date of the first build in a project	
Core developer	Factor	[4, 8, 11, 12]	The author(s) who triggered the build have committed at least once in the last three months	Fixed
Team size	Numeric	[8, 10, 11]	The number of developers in the team	
Source churn	Numeric	[8, 10, 11]	The number of source lines of code changed by the commits that trigger the builds	
Test churn	Numeric	[8, 10, 11]	The number of test lines changed by the commits that trigger the builds	
Doc files changed	Numeric	[8, 10, 11]	The number of documentation files by all build commits	
Other files changed	Numeric	[8, 9, 11]	The number of non production nor documentation files by all build commits	
Previous build status	Factor	[4, 8–11]	The status of the build that precedes the triggered build (*passed*, *broken*, or *NA*)	
Files changed	Numeric	[4, 8–11]	The number of files modified by all the build commits	
Source files changed	Numeric	[8–11]	The number of source files changed the commits that trigger the builds	
Configuration files changed	Numeric	[9, 10]	The number of build configuration files (e.g., `.xml` and `.yml`) modified by the build commits	
Configuration lines added	Numeric	[9, 10]	The number of lines added to configuration files	
Configuration lines deleted	Numeric	[9, 10]	The number of lines deleted from configuration files	
Tests added	Numeric	[8, 11]	The number of added test cases	
Tests deleted	Numeric	[8, 11]	The number of deleted test cases	
Commits on touched files	Numeric	[8, 11]	The number of unique commits on the files touched by the commits that triggered the build	
Is a pull request	Factor	[4, 11]	Whether a commit belongs to a pull request or a Git push	
Time of day	Factor	[4, 10]	The time-zone adjusted time of day of the commit that triggered the build	
Day of week	Factor	[4, 10]	The time-zone adjusted day of week of the commit that triggered the build	
Belongs to the default branch	Factor	[3]	Whether the build is triggered by a commit that belongs to the default branch of the repository	

this argument, we randomly remove sample builds from the original (uncleaned) datasets. The size of the removed sample builds is equal to the size of the builds identified as noisy. We repeat this process 10 times, each with a different random set of samples.

Approach$_2$: We use the data used by Rausch *et al.* [4] containing $\overline{121,258}$ builds from 14 projects. We apply our criteria to identify *Environmental*, *Cascading*, and *Allowed* breakages in the dataset. Then, we process the data similarly to Rausch *et al.*, i.e., (a) we stratify the data by removing builds with unknown previous builds; and (b) we remove data that are outside the 99^{th} percentile of numeric metrics (e.g., the number of commits or lines added). After that, instead of removing builds that perpetuate or fix build failures (as Rausch *et al.* have done), we remove builds that have *Environmental*, *Cascading*, and *Allowed* breakages. Our results are unlikely to be impacted by the differences in the sizes of the uncleaned and cleaned datasets, since Rausch *et al.* also filtered out builds from the original dataset. Finally, we perform the statistical tests used by Rausch *et al.* to study the association between build breakages and 16 metrics. Similarly to Rausch *et al.*, we exclude inconclusive results obtained from projects that have insufficient data for some metrics (e.g., missing build types). Finally, we compare our observations on the cleaned version of the data used by Rausch *et al.* with their reported findings.

Findings: Table 5 shows the variable importance results obtained from our both mixed-effect logistic models. All the metrics are descendingly sorted based on the χ^2 values of the cleaned model. For each metrics, we show its χ^2 value, the p-value ($Pr(< \chi^2)$), its significance, and whether the variable has a direct or an inverse impact on build breakages (the *upward* and *downward* arrows).

The model fit on the cleaned data significantly improves the AUC by 6%. Our model fit on the clean data obtains a good AUC value of 0.83 for discriminating build breakages. This AUC outperforms the model fit on the uncleaned data, which obtains an AUC value of 0.77. By comparing the ROC curves of the two models, we observe that the model fit on the cleaned data has a statistically significant improvement in discriminating build breakages (i.e., we obtain a DeLong's test p-value $< 2.2e^{-16}$). Moreover, when we remove build samples randomly from the uncleaned data, we observe that the (i) average AUC obtained from all the models is 0.77 and (ii) the significant variables obtained from the uncleaned model hold in 90% of the models (results of all the generated models can be found in our online appendix [22]). This result suggests that the noises in build breakage data is likely to distort prediction/regression models for build breakages. In particular, not excluding builds with noisy breakages may influence models to generate incorrect associations between the independent variables and the build breakage.

The uncleaned breakage data generates models that are more sensitive to language, project, and build age variations. The marginal R^2 of the model fit on the cleaned data is 0.21 and the conditional R^2 is 0.36 (i.e., an increase of 71%). On the other hand, the marginal R^2 of the model fit on the uncleaned data is 0.07 and the conditional R^2 is 0.30 (i.e., 329% more). Such results suggest that the difference between the conditional and marginal R^2 values in the cleaned model is smaller than the difference in the model fit on the uncleaned data.

Build breakages have conflicting associations with the variables before and after cleaning the dataset. We observe that the model fit on the cleaned data produces contradicting results compared to the model fit on the uncleaned data. The significant variables observed in the uncleaned data are in agreement with the results reported in prior studies on modeling build breakages [4, 10, 12]. However, according to the model fit on the cleaned data, the vast majority of the explanatory power of the model comes from the status of the previous build (i.e., $\chi^2 = 0.97$). The explanatory power of the status of the previous build in the uncleaned data is most likely reduced due to noisy build breakages. In conclusion, our findings using the cleaned data disagree with the findings of prior research, in the sense that:

- builds triggered at night are less likely to produce build breakages (as opposed to [4]).
- being a core developer does not have a strong association with build breakages (as opposed to [4, 12]).
- team size is unlikely to be associated with build breakages (as opposed to [4, 10]).

TABLE 5: Results of our mixed-effects logistic models — sorted by $Pr(<\chi^2)$ of the cleaned model

Metric	Cleaned model				Uncleaned model			
	χ^2	$Pr(<\chi^2)$	Signf.*	Rel.	χ^2	$Pr(<\chi^2)$	Signf.*	Rel.
(Intercept)	0.0176	$< 2.2e^{-16}$	***	-	0.0623	$< 2.2e^{-16}$	***	-
Previous build status	0.9672	$< 2.2e^{-16}$	***	-	0.5550	$< 2.2e^{-16}$	***	-
Is Pull Request	0.0044	$< 2.2e^{-16}$	***	↗	0.0820	$< 2.2e^{-16}$	***	↗
Belongs to the default branch	0.0040	$< 2.2e^{-16}$	***	↘	0.2425	$< 2.2e^{-16}$	***	↘
Source churn	0.0036	$< 2.2e^{-16}$	***	↗	0.0270	$< 2.2e^{-16}$	***	↗
Commits on touched files	0.0016	0.000	***	↗	0.0019	0.014	*	↗
Day of week	0.0007	0.005	**	-	0.0170	0.000	***	-
Files changed	0.0004	0.001	**	↗	0.0018	0.016	*	↗
Triggered at night	0.0002	0.025	*	↘	0.0009	0.089	.	↘
Configuration files changed	0.0001	0.070	.	↗	0.0002	0.415		↗
Team size	0.0000	0.259		↘	0.0045	0.000	***	↗
Other files changed	0.0000	0.272		↗	0.0008	0.108		↗
Tests added	0.0000	0.294		↗	0.0011	0.057	.	↗
Doc files changed	0.0000	0.445		↗	0.0002	0.427		↘
Core developer	0.0000	0.451		↘	0.0019	0.013	*	↗
Tests deleted	0.0000	0.458		↘	0.0001	0.531		↗
Test churn	0.0000	0.969		↗	0.0008	0.114		↘

*Significance codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

The use of noisy build breakages may have a considerable impact on the observations reported by prior research. We compare the findings reported by Rausch *et al.* [4] with our observations, as follows:

- "*The day of week has no significant influence on build breakages, with little evidence that the time of day influences build breakages.*" However, we observe that builds triggered on weekends have significantly less breakage ratios than builds triggered on weekdays in 80% of the projects. We also observe that late night commits cause fewer breakages in 75% of the projects.
- "*Despite the significant impact of file type changes on build breakages, there is no evidence that certain file types lead to breakages more frequently.*" However, we observe that commits that change *system* files (either alone, with *test*, or with *configuration* files) cause more breakages in 60% of the projects.
- "*Pull requests cause more breakages.*" However, we observe that, in 67% of the projects, integration and merge commits generate builds with higher breakage ratios than any other commits. A possible reason behind such higher breakage ratios is the incompatibility or conflicts that may occur when integrating or merging development branches.
- "*Developers who commit less frequently cause fewer build breakages (in 4 out of 6 projects).*" However, we observe that there is a slight difference between the cases in which developers who commit daily or less frequently cause fewer breakages. In particular, we observe that both daily and less frequently committers cause significantly fewer breakages in 4 and 5 out of 9 projects, respectively. This result suggests that frequently committing is likely to be associated with other factors (e.g., complexity of changes) to impact on build breakages.

Noise in build breakage data can negatively impact the performance of modeling build breakages. Observations reported by prior research may not hold and researchers may gain misleading insights if noisy build breakages are filtered out from build breakage data.

6 Discussion

Build breakage data has been used as the foundation for several empirical conclusions regarding CI [3, 4, 6–10, 26], which is widely used in open source and industry settings. In this section, we discuss how our findings of noisy build breakage data may lead to direct implications for CI researchers and tool builders.

6.1 Researchers

Researchers should be more careful about the quality of historical build breakage data. Researchers have invested a considerable effort to studying CI builds. Hilton *et al.* [26] found that build breakages introduce a barrier that hinders developers from adopting CI. Hence, studies on CI builds have been conducted to help development teams overcome that barrier. A build breakage may occur due to several reasons, such as configuration errors, installation errors, compilation errors, or test failures [4, 5]. Rausch *et al.* [4] and Vassallo *et al.* [5] explored the types of build errors in CI and how frequent they break CI builds. Researchers also studied the possibility of modeling build breakages to understand the factors that share a strong relationship with CI build breakages [4, 6–9]. However, our results show that build breakages can be noisy and have a great impact on the analyses of CI build breakage data. Therefore, we advise future research to revisit the prior analyses regarding CI build breakages to verify whether the prior observations and insights would change considerably after taking noisy builds into account.

6.2 Tool builders

Feedback on CI builds should be enriched with more information about the breakage. Existing CI tools provide developers with an abstract build status (i.e., *passed* or *broken*). Tool builders should consider supplying developers with an enhanced user interface to help them understand (a) the differences between individual job breakages, (b) the types of build errors occurred, (c) whether the build would possibly pass if restarted, (d) whether certain build breakages occurred in the past, and (e) what actions developers have previously made to fix that breakages. Build logs are quite rich of information about the reasons behind build

breakages. Therefore, such information may be regularly collected by a tool to build a history of the frequent root causes of build breakages. In addition, if builds are frequently broken due to connection errors, a useful tool may consider increasing the number of times to rerun failing commands. Likewise, if builds are frequently broken due to exceeding time or log size limits, a tool may consider increasing the time required to wait for CI commands to run or removing redundant or unnecessary information from being logged, respectively.

7 Threats to Validity

This section discusses the threats to the validity of our study.

7.1 Construct validity

Construct threats to validity are concerned with the degree to which our analyses measure what we claim to analyze. We use open coding to categorize and label build logs of broken jobs, which can be subjective. To mitigate such a threat, multiple iterations of manual analysis are performed by two co-authors of this paper to validate the build log labeling and categorization using statistically significant samples of build logs. To identify *Environmental* build breakages, we rely on the last error message in the build logs.If the `script` phase fails, the build continues to run the subsequent phases (e.g., `after_success`, `after_failure`, or `after_script`).[18] Errors in the subsequent phases do not impact the build status, except for timeout errors. Hence, the last error message in the build log might not represent the actual breakage cause. To mitigate this threat, we validate the results of $1,512$ cases (i.e., 1.6%) of environmentally broken builds that are configured to implement the aforementioned phases. We find that: 63% of the builds are *errored* (i.e., errors in the `install` phase), 15% are virtual machine issues, memory issues, or abrupt crashes, 14% are connection or resources issues 6% are failed due to '`unknown host nexus.codehaus.org`' (an issue reported to Travis CI[19]), and 2% are timeout errors. In addition, after investigating a statistically significant sample of 81 cases (a confidence level of 95% and a confidence interval of $\pm 10\%$), we find that the vast majority (i.e., 96%) of the errors occurred during the *script* phase. Hence, errors that occur in the phases that follow the *script* phase are unlikely to impact our obtained results.

We consider a build breakage to be *Environmental* if the error is documented or reported to be related to environmental issues. Nevertheless, developers may commit code changes that may likely induce environmental breakages. For example, adding expensive testing code (e.g., poor choice of algorithms) may potentially lead to time-exceeding errors, which may break the build unexpectedly. To mitigate such a threat, we perform additional manual analyses on statistically significant samples of environmental build breakages (95% confidence level and $\pm 10\%$ confidence interval) that might possibly be caused by developers. Overall, we find less than 2% of *Environmental* breakages in which developers change build configuration (i.e., script file), with only 0.2% not being identified as *Cascading* or *Allowed* breakages. We analyze 58 builds having build script compilation errors. We observe that, in 93% of the cases, developers do not change the script file. We also analyze 92 builds having tests that timed out due to exceeding predefined limits. We observe that developers add more expensive code (e.g., loops) only in 9% of the cases. Of such cases, we find that only one case in which a passing build that follows a breakage contains code changes that fix the issues of long-running tests. Moreover, we analyze a similarly statistical sample of builds that have dependency-related breakages. We find that developers do not add or change dependencies in 89% of the cases. Such results suggest that environmental build breakages are most likely to be related to environmental issues rather than development activities.

Developers may exclude jobs at later stages (i.e., in builds that are beyond our dataset). Predicting whether a developer will exclude a certain job at a later stage is out of the scope of this study. Therefore, we assume that a build breakage is an *Allowed* breakage if the dataset has evidence that the jobs breaking the build are excluded by developers later. Moreover, we manually investigate the reasons behind excluding build jobs. We find that developers identify the exclude jobs to be noisy or flaky in 69% of the time. However, identifying a job to be noisy at a certain point of time may not imply that the job was noisy in the project all the time. Still, we advise researchers to filter builds that are primarily broken by such jobs out, since the breakages could be due to the abnormal behavior of the excluded jobs.

All of reported results and statistics about noisy build breakages are computed using our written Python and R scripts. Our scripts may contain defects that might affect the reported results. To address this threat, we perform additional manual analyses to verify the results of each of our proposed criteria.

7.2 Internal validity

Internal threats to validity are concerned with the ability to draw conclusions from the relationship between build breakages and other build characteristics. Our study does not consider *passed* builds as noisy. However, in rare cases, RSpec in Ruby may sometimes return 0 due to overwriting the `at_exit` handler by different gems.[20] Our scope in this study is to focus on the factors that may lead to noisy breakages. We leave the investigation of the potential noises in *passed* builds for future work.

7.3 External validity

External threats are concerned with our ability to generalize our results. Our study investigates noises in build breakage data of 153 projects that have enough samples (i.e., over $1,000$ builds per each project) to perform our analyses. Although the studied projects are of different languages (i.e., Ruby and Java), code sizes, team sizes, and domains, we cannot generalize our conclusions to other projects where these settings heavily vary. For example, we may observe different kinds of build breakages and findings if we analyze data of projects written in other programming languages (e.g., Python or JavaScript) or linked with different CI build services (e.g., Circle CI or Appveyor). Replication of this work using additional software projects and other CI services is required in order to reach more general conclusions.

[18] https://docs.travis-ci.com/user/job-lifecycle#breaking-the-build

[19] https://github.com/travis-ci/travis-ci/issues/4629

[20] https://docs.travis-ci.com/user/common-build-problems

8 Related Work

In this section, we present the related research while highlighting our contributions. Related work includes five study areas: (i) study replication/reproduction, (ii) noises in software engineering datasets, (iii) types of build breakages, (iv) modeling build breakages, and (v) flaky (unstable) tests.

8.1 Study replication/reproduction

Previous studies have explored the feasibility of replicating and reproducing the results of empirical studies in software engineering [38–40]. Shull *et al.* [38] highlighted that obtaining similar results in a study replication can confirm a certain phenomenon, whereas obtaining contradicting results can provide insights on why the results are not the same. Robles [39] and Rodríguez-Pérez *et al.* [40] investigated the potential replicability of the papers published in the areas of mining software repositories and empirical software engineering. The authors found that the majority of the published papers are unlikely to be replicated due to sharing *raw* data and general tool names. Study reproduction, on the other hand, aims to reproduce or validate a phenomenon of an original study using different data and setup [41, 42]. In our study, we strive to reproduce observations reported by prior studies in modeling build breakages before and after cleaning the noises from (i) a large dataset of 153 projects and (ii) a dataset of 14 projects used by a prior study [4].

8.2 Noise in software engineering datasets

The literature contains studies that investigate the bias in datasets to improve the quality of the obtained results. Mockus [43] highlighted that data quality is more important than the choice of analysis methods, and emphasized that invalid data may lead to a sampling bias in software engineering. Liebchen and Shepperd [44] found that only a few empirical studies in software engineering assessed the quality of the data used. Herzig *et al.* [45] found that over 33% of bug reports are misclassified, which may impact the validity of the results reported by earlier studies. Antoniol *et al.* [46] and Bird *et al.* [47] reported that heuristics used to tag bug reports or link reports to source code may introduce bias to the data, which may impact the quality of the produced data. Nguyen *et al.* [48] observed that the bug linkage bias is more likely to be a property of the software process itself, whereas the tagging bias does not really impact bug prediction models. Tantithamthavorn *et al.* [49] observed that mislabelling issue reports in defect prediction data is not random, and models that are trained using mislabelled data obtain only $56 - 68\%$ of the recall of models that are trained using cleaned data. da Costa *et al.* [50] evaluated the data generated by a tool that identifies bug-introducing changes. They found data inaccuracies, which suggests that the quality of the retrieved data still needs improvement.

Kalliamvakou *et al.* [51] indicated that researchers need to be cautious when mining data from GitHub, since repositories could be used for data storage rather than software projects. Howison and Crowston [52] studied the perils and pitfalls of mining SourceForge data. They found that a considerable amount of data are partially hosted on other websites, which may impair the analyses performed by researchers. Gallaba *et al.* [12] studied the noise and heterogeneity in build breakage data. They observed that builds may contain ignored breakages or breakages that are not caused by the building tool (e.g., Maven). However, the authors did not consider cleaning the data to generate the reported observations. In addition, the impact of noise on the findings of prior research was not studied. Ghaleb *et al.* [53] studied CI build durations and observed that builds broken due to timeouts may take longer than passed builds.

Complementing the aforementioned studies, we investigate the noises that may exist in build breakage data. Our results reveal that not considering *Environmental*, *Cascading*, and *Allowed* build breakages may lead to incorrect conclusions regarding build breakages.

8.3 Types of build breakages

Existing research has investigated the reasons behind CI build breakages [4, 5, 19–21]. A study by Beller *et al.* [19] shows that tests highly impact the build status. Another study by Zolfagharinia *et al.* [20] shows that operating systems and runtime environments of CI heavily impact the build status. Rausch *et al.* [4] studied the causes of build breakages and reported 14 breakage categories of CI builds. The authors reported associations between development activities and build breakages. Vassallo *et al.* [5] reported 20 categoriesof build breakages and found that open source and industrial projects share common breakage patterns. We complement their work by studying the categories for environmental build breakages. Ziftci and Reardon [21] proposed a technique to identify code changes that may introduce test failures in CI builds and their results show that 78% of the feedback provided by the technique was beneficial. Vassallo *et al.* [54] proposed a tool that summarizes reasons of build breakages and suggests how to resolve the breakages using online resources. van der Storm [55] proposed an approach that allows developers to backtrack CI builds and to use earlier successful versions of builds in which the component(s) failed.

In our work, we aim to thoroughly investigate the *Environmental*, *Cascading*, and *Allowed* build breakages that can potentially introduce noises in build breakage data but not studied in prior studies.

8.4 Modeling build breakages

Prior to CI, studies introduced prediction models to predict the status of builds [6, 7, 56]. Hassan and Zhang [6] used trained decision trees with project features to predict the certification status of a build. Wolf *et al.* [56] and Kwan *et al.* [7] constructed build breakage prediction models by leveraging measurements of the communication networks used by developers and measurements of coordination between social and technical activities of developers, respectively.

For CI builds, research has investigated the possibility of predicting whether the CI build will pass or break [8, 9]. Xia and Li [8] built 9 prediction models to predict the build status. They validated their models using a cross-validation scenario and an online scenario. Their models achieved a prediction AUC of over 0.80 for the majority of the projects they studied. They found that predicting a build status using the online scenario performed worse than that of cross-validation, with a mean AUC difference of 0.19. They observed that the prediction accuracy falls down due to the frequent changes of project characteristics, development

phases, and build characteristics across different version control branches. Ni and Li [9] used cascade classifiers to predict build failures. Their classifiers achieve higher AUC values than other basic classifiers, such as decision trees and Naive Bayes. They also observed that historical committers and project statistics are the best indicators of build statuses.

The prior studies derive their conclusions based on the build statuses provided by TRAVIS CI, which are identified in this paper to be potentially noisy. In our study, we study how modeling build breakages and subsequent findings could be negatively impacted by the potential noises in the build breakage data.

8.5 Flaky (unstable) tests

Flaky tests are software tests that maintain a non-deterministic status (i.e., they sometimes pass and sometimes fail) for a given software version [57]. Errors in flaky tests are unlikely to be related to the committed code changes but rather due to resource unavailability/unresponsiveness or environmental factors [14, 16, 17]. Luo *et al.* [15] identified the most common root causes of flaky tests and suggested ways to capture the flaky behavior of tests. Shamshiri *et al.* [16] proposed an approach that detects flaky tests by running every particular test several times. To better understand non-determinism in tests, Memon *et al.* [17] studied the relationship between test status transitions and other development factors, such as developer, code, committing frequency, and test execution factors. The authors suggested guidelines for developers to avoid test flakiness.

Despite the research invested to study flaky tests, little is known about flakiness in CI build breakages and its impact on research findings. In our work, we study *Environmental*, as well as *Cascading* and *Allowed* build breakages that may occur in all build phases (i.e., installation, compilation, and testing). Moreover, we show that such breakages can impact the findings of prior research.

9 CONCLUSION

In this paper, we conduct an empirical study to investigate the noises that may exist in build breakage data. We define three criteria to identify noises in build breakage data. In particular, our criteria identify the builds that have (1) *Environmental* breakages, (2) *Cascading* breakages, and (3) *Allowed* breakages. We first clean our studied dataset and propose a catalogue of the categories of possible *Environmental* build breakages in CI. Second, we measure the impact of using noisy build breakage data on modeling build breakages. Our results reveal that noisy build breakage data can (a) reduce the of modeling build breakages and (b) distort the association between build breakages and other metrics. Moreover, we find that observations reported by prior research may not hold and researchers may gain misleading insights if noisy build breakages are filtered out from build breakage data.

Our study suggests that researchers and practitioners should be more careful when they deal with build breakage data. In particular, the build statuses (i.e., *errored*, *failed* and *passed*) provided by CI services are not reliable enough to judge that builds are broken due to development activities. In fact, builds may be broken due to environmental factors or simply cascade previous (unfixed) breakages. Therefore, using such noisy data may consequently lead to lower performance and incorrect observations when modeling build breakages. Although our study identifies the proportion of noise in build breakage data, we observe that projects still have long sequences of consecutive breakages. We aim in the future to conduct a qualitative study to investigate other reasons behind not fixing build breakages as soon as they occur. Furthermore, we aim to study the common development practices to deal with build breakages.

REFERENCES

[1] M. Fowler and M. Foemmel. (2006) Continuous integration. [Online]. Available: http://www.dccia.ua.es/dccia/inf/asignaturas/MADS/2013-14/lecturas/10_Fowler_Continuous_Integration.pdf

[2] B. Vasilescu, S. Van Schuylenburg, J. Wulms, A. Serebrenik, and M. G. van den Brand, "Continuous integration in a social-coding world: Empirical evidence from github," in *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME 2014)*. IEEE, 2014, pp. 401–405.

[3] M. Hilton, T. Tunnell, K. Huang, D. Marinov, and D. Dig, "Usage, costs, and benefits of continuous integration in open-source projects," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016)*. ACM, 2016, pp. 426–437.

[4] T. Rausch, W. Hummer, P. Leitner, and S. Schulte, "An empirical analysis of build failures in the continuous integration workflows of Java-based open-source software," in *Proceedings of the 14th International Conference on Mining Software Repositories (MSR 2017)*, 2017, pp. 345–355.

[5] C. Vassallo, G. Schermann, F. Zampetti, D. Romano, P. Leitner, A. Zaidman, M. Di Penta, and S. Panichella, "A tale of ci build failures: An open source and a financial organization perspective," in *Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on*. IEEE, 2017, pp. 183–193.

[6] A. E. Hassan and K. Zhang, "Using decision trees to predict the certification result of a build," in *Proceedings of the International Conference on Automated Software Engineering (ASE 2006)*. IEEE, 2006, pp. 189–198.

[7] 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, 2011.

[8] J. Xia and Y. Li, "Could We Predict the Result of a Continuous Integration Build? An Empirical Study," in *Proceedings of the IEEE International Conference on Software Quality, Reliability and Security Companion (QRS 2017)*, 2017, pp. 311–315.

[9] A. Ni and M. Li, "Cost-effective build outcome prediction using cascaded classifiers," in *Proceedings of the 14th International Conference on Mining Software Repositories (MSR 2017)*, 2017, pp. 455–458.

[10] F. Hassan and X. Wang, "Change-aware build prediction model for stall avoidance in continuous integration," in *Proceedings of the 11th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM 2017)*. IEEE Press, 2017, pp. 157–162.

[11] Y. Luo, Y. Zhao, W. Ma, and L. Chen, "What are the factors impacting build breakage?" in *Proceedings of the 14th Conference on Web Information Systems and Applications Conference (WISA 2017)*. IEEE, 2017, pp. 139–142.

[12] K. Gallaba, C. Macho, M. Pinzger, and S. McIntosh, "Noise and heterogeneity in historical build data: an empirical study of travis ci," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE 2018)*. ACM, 2018, pp. 87–97.

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

[14] F. J. Lacoste, "Killing the gatekeeper: Introducing a continuous integration system," in *Agile Conference*. IEEE, 2009, pp. 387–392.

[15] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An empirical analysis of flaky tests," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 643–653.

[16] S. Shamshiri, R. Just, J. M. Rojas, G. Fraser, P. McMinn, and A. Arcuri, "Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges (t)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2015, pp. 201–211.

[17] A. Memon, Z. Gao, B. Nguyen, S. Dhanda, E. Nickell, R. Siemborski, and J. Micco, "Taming google-scale continuous testing," in *Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track*. IEEE Press, 2017, pp. 233–242.

[18] 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 International Conference on Mining Software Repositories (MSR 2014)*, 2017, pp. 447–450.

[19] 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 International Conference on Mining Software Repositories (MSR 2017)*, 2017, pp. 356–367.

[20] M. Zolfagharinia, B. Adams, and Y.-G. Guéhéneuc, "Do not trust build results at face value: an empirical study of 30 million CPAN builds," in *Proceedings of the 14th International Conference on Mining Software Repositories (MSR 2017)*, 2017, pp. 312–322.

[21] C. Ziftci and J. Reardon, "Who broke the build? automatically identifying changes that induce test failures in continuous integration at google scale," in *Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP 2017)*. IEEE, 2017, pp. 113–122.

[22] "Studying the Impact of Noises in Build Breakage Data [Online Appendix] ," http://doi.org/10.5281/zenodo.3401555.

[23] J. Fereday and E. Muir-Cochrane, "Demonstrating rigor using thematic analysis: A hybrid approach of inductive and deductive coding and theme development," *International journal of qualitative methods*, vol. 5, no. 1, pp. 80–92, 2006.

[24] J. M. Corbin and A. Strauss, "Grounded theory research: Procedures, canons, and evaluative criteria," *Qualitative sociology*, vol. 13, no. 1, pp. 3–21, 1990.

[25] J. Cohen, "A coefficient of agreement for nominal scales," *Educational and psychological measurement*, vol. 20, no. 1, pp. 37–46, 1960.

[26] M. Hilton, N. Nelson, T. Tunnell, D. Marinov, and D. Dig, "Trade-offs in continuous integration: assurance, security, and flexibility," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (FSE 2017)*. ACM, 2017, pp. 197–207.

[27] 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 2014)*. ACM, 2014, pp. 724–734.

[28] N. Kerzazi, F. Khomh, and B. Adams, "Why do automated builds break? an empirical study," in *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME 2014)*. IEEE, 2014, pp. 41–50.

[29] P. Domingos, "A few useful things to know about machine learning," *Communications of the ACM*, vol. 55, no. 10, pp. 78–87, 2012.

[30] W. Sarle, "The VARCLUS Procedure," *SAS/STAT User's Guide*, 1990.

[31] J. J. Faraway, *Extending the linear model with R: generalized linear, mixed effects and nonparametric regression models*. CRC press, 2016, vol. 124.

[32] A. J. Lewis, *Mixed effects models and extensions in ecology with R*. Springer, 2009.

[33] P. Pinheiro, "Linear and nonlinear mixed effects models. r package version 3.1-97," *http://cran. r-project. org/web/packages/nlme*, 2010.

[34] J. A. Hanley and B. J. McNeil, "The meaning and use of the area under a receiver operating characteristic (ROC) curve," *Radiology*, vol. 143, no. 1, pp. 29–36, 1982.

[35] E. R. DeLong, D. M. DeLong, and D. L. Clarke-Pearson, "Comparing the areas under two or more correlated receiver operating characteristic curves: a nonparametric approach," *Biometrics*, vol. 44, no. 3, pp. 837–845, 1988.

[36] P. Peduzzi, J. Concato, E. Kemper, T. R. Holford, and A. R. Feinstein, "A simulation study of the number of events per variable in logistic regression analysis," *Journal of clinical epidemiology*, vol. 49, no. 12, pp. 1373–1379, 1996.

[37] E. W. Steyerberg, F. E. Harrell Jr, G. J. Borsboom, M. Eijkemans, Y. Vergouwe, and J. D. F. Habbema, "Internal validation of predictive models: efficiency of some procedures for logistic regression analysis," *Journal of clinical epidemiology*, vol. 54, no. 8, pp. 774–781, 2001.

[38] F. J. Shull, J. C. Carver, S. Vegas, and N. Juristo, "The role of replications in empirical software engineering," *Empirical software engineering*, vol. 13, no. 2, pp. 211–218, 2008.

[39] G. Robles, "Replicating msr: A study of the potential replicability of papers published in the mining software repositories proceedings," in *Proceedings of the 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*. IEEE, 2010, pp. 171–180.

[40] G. Rodríguez-Pérez, G. Robles, and J. M. González-Barahona, "Reproducibility and credibility in empirical software engineering: A case study based on a systematic literature review of the use of the szz algorithm," *Information and Software Technology*, 2018.

[41] C. Drummond, "Replicability is not reproducibility: nor is it good science," in *In Proceedings of the Evaluation Methods for Machine Learning Workshop at the 26th International Conference for Machine Learning*, 2009, pp. 1–4.

[42] P. Ivie and D. Thain, "Reproducibility in scientific computing," *ACM Computing Surveys (CSUR)*, vol. 51, no. 3, p. 63, 2018.

[43] A. Mockus, "Missing data in software engineering," in *Guide to advanced empirical software engineering*. Springer, 2008, pp. 185–200.

[44] G. A. Liebchen and M. Shepperd, "Data sets and data quality in software engineering," in *Proceedings of the 4th International Workshop on Predictor Models in Software Engineering (PROMISE 2008)*. ACM, 2008, pp. 39–44.

[45] K. Herzig, S. Just, and A. Zeller, "It's not a bug, it's a feature: how misclassification impacts bug prediction," in *Proceedings of the 2013 international conference on software engineering*. IEEE Press, 2013, pp. 392–401.

[46] G. Antoniol, K. Ayari, M. Di Penta, F. Khomh, and Y.-G. Guéhéneuc, "Is it a bug or an enhancement?: a text-based approach to classify change requests," in *Proceedings of the Conference of the Center for Advanced Studies on Collaborative Research (CASCON 2008)*. ACM, 2008, p. 23.

[47] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu, "Fair and balanced?: bias in bug-fix datasets," in *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC/FSE 2009)*. ACM, 2009, pp. 121–130.

[48] T. H. Nguyen, B. Adams, and A. E. Hassan, "A case study of bias in bug-fix datasets," in *Proceedings of the 17th Working Conference on Reverse Engineering (WCRE 2010)*. IEEE, 2010, pp. 259–268.

[49] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, A. Ihara, and K. Matsumoto, "The impact of mislabelling on the performance and interpretation of defect prediction models," in *Proceedings of the 37th IEEE International Conference on Software Engineering (ICSE 2015)*, vol. 1. IEEE, 2015, pp. 812–823.

[50] D. A. da Costa, S. McIntosh, W. Shang, U. Kulesza, R. Coelho, and A. E. Hassan, "A framework for evaluating the results of the szz approach for identifying bug-introducing changes," *IEEE Transactions on Software Engineering*, vol. 43, no. 7, pp. 641–657, 2017.

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

[52] J. Howison and K. Crowston, "The perils and pitfalls of mining sourceforge," in *Proceedings of the International Workshop on Mining Software Repositories (MSR 2004)*. IET, 2004, pp. 7–11.

[53] T. A. Ghaleb, D. A. da Costa, and Y. Zou, "An empirical study of the long duration of continuous integration builds," *Empirical Software Engineering*, pp. 1–38, 2019.

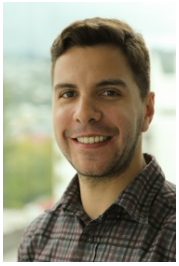
[54] C. Vassallo, S. Proksch, T. Zemp, and H. C. Gall, "Un-break my build: Assisting developers with build repair hints," in *Proceedings of the International Conference on Program Comprehension (ICPC 2018)*, 2018, p. to appear.

[55] T. Van Der Storm, "Backtracking incremental continuous integration," in *Proceedings of the 12th European Conference on Software Maintenance and Reengineering*, 2008, pp. 233–242.

[56] T. Wolf, A. Schroter, D. Damian, and T. Nguyen, "Predicting build failures using social network analysis on developer communication," in *Proceedings of the 31st International Conference on Software Engineering (ICSE 2009)*. IEEE Computer Society, 2009, pp. 1–11.

[57] M. Fowler. (2011) Continuous integration. [Online]. Available: https://martinfowler.com/articles/nonDeterminism.html

Taher Ahmed Ghaleb is a Ph.D. candidate at the School of Computing at Queen's University in Canada. Taher is the holder of an Ontario Trillium Scholarship, a highly prestigious award for doctoral students. He obtained a B.Sc. in Information Technology from Taiz University in Yemen (2008) and an M.Sc. in Computer Science from King Fahd University of Petroleum and Minerals in Saudi Arabia (2016). He worked as a Teaching Assistant at Taiz University in Yemen (2008−2011). His research interests include continuous integration, program analysis, mining software repositories, data analytics, applied machine learning, and empirical software engineering.



Daniel Alencar da Costa is a Lecturer (Assistant Professor) at the University of Otago, New Zealand. Daniel obtained his PhD in Computer Science at the Federal University of Rio Grande do Norte (UFRN) in 2017 followed by a Postdoctoral Fellowship at Queens University, Canada, from 2017 to late 2018. His research goal is to advance the body of knowledge of Software Engineering methodologies through empirical studies using statistical and machine learning approaches as well as consulting and documenting the experience of Software Engineering practitioners.

Ying (Jenny) Zou is the Canada Research Chair in Software Evolution. She is a professor in the Department of Electrical and Computer Engineering, and cross-appointed to the School of Computing at Queen's University in Canada. She is a visiting scientist of IBM Centers for Advanced Studies, IBM Canada. Her research interests include software engineering, software reengineering, software reverse engineering, software maintenance, and service-oriented architecture. More about Dr. Zou and her work is available online at http://post.queensu.ca/~zouy

Ahmed E. Hassan is an IEEE Fellow, an ACM SIGSOFT Influential Educator, an NSERC Steacie Fellow, the Canada Research Chair (CRC) in Software Analytics, and the NSERC/BlackBerry Software Engineering Chair at the School of Computing at Queen's University, Canada. His research interests include mining software repositories, empirical software engineering, load testing, and log mining. He received a PhD in Computer Science from the University of Waterloo. He spearheaded the creation of the Mining Software Repositories (MSR) conference and its research community. He also serves/d on the editorial boards of IEEE Transactions on Software Engineering, Springer Journal of Empirical Software Engineering, and PeerJ Computer Science. More information at: http://sail.cs.queensu.ca