Noise and Heterogeneity in Historical Build Data

An Empirical Study of Travis CI

Keheliya Gallaba
McGill University
Montréal, Canada
keheliya.gallaba@mail.mcgill.ca

Christian Macho
University of Klagenfurt
Klagenfurt, Austria
christian.macho@aau.at

Martin Pinzger
University of Klagenfurt
Klagenfurt, Austria
martin.pinzger@aau.at

Shane McIntosh
McGill University
Montréal, Canada
shane.mcintosh@mcgill.ca

ABSTRACT

Automated builds, which may pass or fail, provide feedback to a development team about changes to the codebase. A passing build indicates that the change compiles cleanly and tests (continue to) pass. A failing (a.k.a., broken) build indicates that there are issues that require attention. Without a closer analysis of the nature of build outcome data, practitioners and researchers are likely to make two critical assumptions: (1) build results are not noisy; however, passing builds may contain failing or skipped jobs that are actively or passively ignored; and (2) builds are equal; however, builds vary in terms of the number of jobs and configurations.

To investigate the degree to which these assumptions about build breakage hold, we perform an empirical study of 3.7 million build jobs spanning 1,276 open source projects. We find that: (1) 12% of passing builds have an actively ignored failure; (2) 9% of builds have a misleading or incorrect outcome on average; and (3) at least 44% of the broken builds contain passing jobs, i.e., the breakage is local to a subset of build variants. Like other software archives, build data is noisy and complex. Analysis of build data requires nuance.

CCS CONCEPTS

• **Software and its engineering → Software verification and validation**; *Software post-development issues*;

KEYWORDS

Automated Builds, Build Breakage, Continuous Integration

ACM Reference Format:
Keheliya Gallaba, Christian Macho, Martin Pinzger, and Shane McIntosh. 2018. Noise and Heterogeneity in Historical Build Data: An Empirical Study of Travis CI. In *Proceedings of the 2018 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18), September 3–7, 2018, Montpellier, France.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3238147.3238171

1 INTRODUCTION

After making source code changes, developers execute automated builds to check the impact on the software product. These builds are triggered while features are being developed, when changes have been submitted for peer review, and/or prior to integration into the software project's version control system.

Tools such as Travis CI facilitate the practice of Continuous Integration (CI), where code changes are downloaded regularly onto dedicated servers to be compiled and tested [1]. The popularity of development platforms such as GitHub and CI services such as Travis CI have made the data about automated builds from a plethora of open source projects readily available for analysis.

Characterizing build outcome data will help software practitioners and researchers when building tools and proposing techniques to solve software engineering problems. For example, Rausch et al. [18] identified the most common breakage types in 14 Java applications and Vassallo et al. [26] compared breakages from 349 open source Java projects to those of a financial organization. While these studies make important observations, understanding the nuances and complexities of build outcome data has not received sufficient attention by software engineering researchers. Early work by Zolfagharinia et al. [29] shows that build failures in the Perl project tend to be time- and platform-sensitive, suggesting that interpretation of build outcome data is not straightforward.

To support interpretation of build outcome data, in this paper, we set out to characterize build outcome data according to two harmful assumptions that one may make. To do so, we conduct an empirical study of 3,702,071 build results spanning 1,276 open source projects that use the Travis CI service.

Noise. First, one may assume that build outcomes are *free of noise*. However, we find that in practice, some builds that are marked as successful contain breakages that need attention yet are ignored. For example, developers may label platforms in their Travis CI configurations as `allow_failure` to enable experimentation with support for a new platform. The expectation is that once platform support has stabilized, developers will remove `allow_failure`; however, this is not always the case. For example, the *zdavatz/spreadsheet*[1] project has had the `allow_failure` feature enabled for the entire lifetime of the project (five years). Examples like this suggest that noise is likely present in build outcome data.

[1]https://github.com/zdavatz/spreadsheet

There are also builds that are marked as broken that do not receive the immediate attention of the development team. It is unlikely that such broken builds are as distracting for development teams as one may assume. For example, we find that on average, two in every three breakages are *stale*, i.e., occur multiple times in a project's build history. To quantify the amount of noise in build outcome data, we propose an adapted signal-to-noise ratio.

Heterogeneity. Second, one may assume that builds are *homogeneous*. However, builds vary in terms of the number of executed jobs and the number of supported build-time configurations. For example, if the Travis CI configuration includes four Ruby versions and three Java versions to be tested, twelve jobs will be created per build because 4×3 combinations are possible. Zolfagharinia et al. [29] observed that automated builds for Perl package releases take place on a median of 22 environments and seven operating systems. Builds also vary in terms of the type of contributor. Indeed, build outcome and team response may differ depending on the role of the contributor (core, peripheral).

In this paper, we study build heterogeneity according to matrix breakage purity, breakage reasons, and contributor type. We find that (1) environment-specific breakages are as common as environment-agnostic breakages; (2) the reasons for breakage vary and can be classified into five categories and 24 subcategories; and (3) broken builds that are caused by core contributors tend to be fixed sooner than those of peripheral contributors.

Take-away messages. Build outcome data is noisy and heterogeneous in practice. If build outcomes are treated as the ground truth, this noise will likely impact subsequent analyses. Therefore, researchers should filter out noise in build outcome data before conducting further analyses. Moreover, tool developers and researchers who develop and propose solutions based on build outcome data need to take the heterogeneity of builds into account.

In summary, this paper makes the following contributions:

- An empirical study of noise and heterogeneity of build breakage in a large sample of Travis CI builds.
- A replication package containing Travis CI specification files, metadata, build logs at the job level, and our data extraction and analysis scripts.[2]
- A taxonomy of breakage types that builds upon prior work.

Paper organization. The remainder of the paper is organized as follows: Section 2 describes the research methodology. Sections 3 and 4 present our findings related to noise in build outcome and build heterogeneity, respectively. Section 5 discusses the broader implications of our study for the research and tool building communities. Section 6 outlines the threats to validity. Section 7 surveys related work. Finally, Section 8 concludes the paper.

2 STUDY DESIGN

In this section, we describe our rationale for selecting the corpus of studied systems and our approach to analyze this large corpus of build data, which follows Mockus' four-step procedure [17] for mining software data. Figure 1 provides an overview of our approach.

[2] https://github.com/software-rebels/bbchch

2.1 Corpus of Candidate Systems

We conduct this study by using openly available project metadata and build results of GitHub projects that use the Travis CI service to automate their builds. GitHub is the world's largest hosting service of open source software, with around 20 million users and 57 million repositories, in 2017.[3] A recent analysis shows that Travis CI is the most popular CI service among projects on GitHub.[4]

2.2 Retrieve Raw Data

We begin by retrieving the TravisTorrent dataset [3], which contains build outcome data from GitHub projects that use the Travis CI service. As of our retrieval, the TravisTorrent dataset contains data about 3,702,595 build jobs that belong to 680,209 builds spanning 1,283 GitHub projects. Those builds include one to 252 build jobs (median of 3). In addition to build-related data, the TravisTorrent dataset contains details about the GitHub activity that triggered each build. For example, every build includes a commit hash (a reference to the build triggering activity in its *Git* repository), the amount of churn in the revision, the number of modified files, and the programming language of the project. TravisTorrent also includes the number of executed and passed tests.

TravisTorrent alone does not satisfy all of the requirements of our analysis. Since TravisTorrent infers the build job outcome by parsing the raw log, it is unable to detect the outcome of 794,334 jobs (21.45%). Furthermore, TravisTorrent provides a single *broken* category, whereas Travis CI records build breakage in three different categories (see Subsection 2.4).

To satisfy our additional data requirements, we complement the TravisTorrent dataset by extracting additional data from the REST API that is provided by Travis CI. From the API, we collect the CI specification (i.e., `.travis.yml` file) used by Travis CI to create each build job and the outcome of each build job. To enable further analysis of build breakages, we also download the plain-text logs of each build job in the TravisTorrent dataset.

2.3 Clean and Process Raw Data

Since we focus on build breakages, we filter away projects that do not have any broken builds. This excludes from our analysis toy projects that have configured CI initially but do not use CI services. 1,276 projects (out of 1,283) survive this filter.

We observe that 996 build logs do not parse cleanly. When retrieving these logs, the Travis CI API returned a truncated or invalid response. We also filter these logs out of our analysis; however, we do note that these 996 logs account for a negligible proportion of the sample of analyzed build logs (996/3,702,595 = 0.03%).

2.4 Construct Meaningful Metrics

In this subsection, we first define the Travis CI concepts that are useful for understanding our work. Then, we define the metrics that we use to operationalize the study dimensions.

Core Concepts in Travis CI. In this paper, we adhere to the terminology as defined in the official Travis CI documentation.[5]

[3] https://github.com/blog/2345-celebrating-nine-years-of-github-with-an-anniversary-sale
[4] https://github.com/blog/2463-github-welcomes-all-ci-tools
[5] https://docs.travis-ci.com/user/for-beginners/

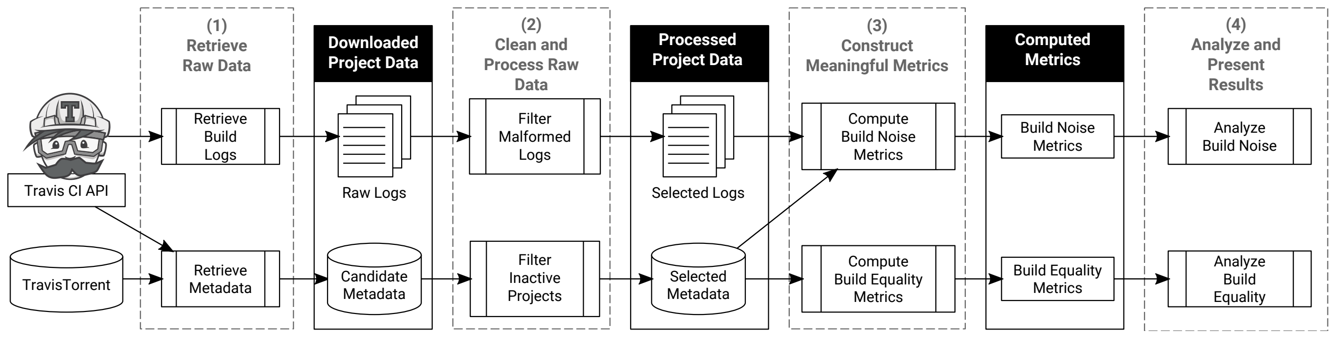


Figure 1: An overview of the approach we followed for data analysis.

A **job** is an automated process that clones a particular revision of a *Git* repository into a (virtual) environment and then carries out a series of tasks, such as compiling the code and executing tests.

Each job is comprised of three main **phases**: *install*, *script*, and *deploy*. Each phase may be preceded by a before sub-phase or followed by an after sub-phase. These sub-phases are often used to ensure that all of the pre-conditions are satisfied before the main phase is executed (*before install, before script, before deploy*), and all of the post-conditions are met after executing the main phase commands (*after success, after failure, after deploy*).

A **build** is comprised of jobs. For example, a build can have multiple jobs, each of which tests the project with a different variant of the development or runtime environment. Once all of the jobs in the build are finished, the build is also finished.

For each job, Travis CI reports one of four outcomes:

- **Passed.** The project was built successfully and passed all tests. All phases terminate with an exit code of zero.
- **Errored.** If any of the commands that are specified in the *before_install, install,* or *before_script* phases of the build lifecycle terminate with a non-zero exit code, the build is labelled as errored and stops immediately.
- **Failed.** If a command in the *script* phase terminates with a non-zero exit code, the build is labelled as failed, but execution continues with the *after_failure* phase.
- **Cancelled.** A Travis CI user with sufficient permissions can abort the build using the web interface or the API. Such builds are labelled as cancelled.

Projects that use the Travis CI service inform Travis CI about how build jobs are to be executed using a `.travis.yml` configuration file. The properties that are set in this configuration file specify which revisions will initiate builds, how the build environments are to be configured for executing builds, and how different teams or team members should be notified about the outcome of the build. Furthermore, the configuration file specifies which tools are required during the build process and the order in which these tools need to be executed.

Metrics. Based on the above concepts, we define seven metrics to analyze build breakage. These metrics are not intended to be complete, but instead provide a starting point for inspecting build breakage for suspicious entries that future work can build upon. Our initial set of metrics belong to two dimensions.

- **Build noise metrics.** In this dimension, we compute the rate at which build breakage is *actively ignored* and *passively ignored*. In addition, we measure the *staleness* of each broken build, i.e., the rate at which breakages are recurring. Finally, we compute the *Signal-To-Noise Ratio (SNR)* to measure the proportion of noise in build outcome data caused by passively and actively ignored build breakage.
- **Build heterogeneity metrics.** In this dimension, for each broken build, we compute the *matrix breakage purity* and classify broken builds by the *root cause*. For practical reasons, we extract root causes for build breakage from the 67,267 jobs that use the Maven build tool. This allows us to build upon the Maven Log Analyzer [15], which can classify five types and 24 subtypes of Maven build breakage. Finally, we classify each of the version control revisions that are associated with each build, according to *contributor type* (i.e., core or peripheral contributors).

2.5 Analyze and Present Results

Using the metrics that we define in Section 3.4, we (1) plot their values using bar charts, line graphs, scatterplots, and bean plots [10]; and (2) conduct statistical analyses using Spearman's ρ, Wilcoxon signed rank tests, and Cliff's δ.

3 NOISE IN BUILD BREAKAGE DATA

The final build outcome does not always tell the complete story. Indeed, a broken build outcome may not indicate a problem with the system, but rather a problem with the build system or test suite. Conversely, a passing build outcome may only be labeled as such because breakages in particular jobs are being ignored.

In this section, we present the results of our noisiness study in terms of outcomes of builds that are actively ignored (3.1), passively ignored (3.2), or stale (3.3). We also provide an overview of the signal-to-noise ratio in the studied corpus (3.4).

3.1 Actively Ignored by Developers

Motivation. Support for new runtime environments is often slowly rolled out through adaptive maintenance [20]. While support for new platforms are in the experimental stage, developers may ignore build breakage on these platforms.

To prevent failing jobs in experimental areas of the codebase from causing build breakage, Travis CI users can set the `allow_failure` property when testing against versions or configurations that developers are not ready to officially support.[6] In other words, a job may fail; however, because the developers chose to ignore the outcome of its configuration, the outcome of the build is passing. So if analyses assume the build is successful because the reported outcome is passing, *actively ignored breakages* may introduce noise. Therefore, we analyze how often breakages are actively ignored in our corpus.

Approach. We begin by selecting all of the 496,240 passing builds in our dataset. From those builds, we select the ones with failing jobs. Then, we retrieve the corresponding version of the `.travis.yml` for each of those selected builds and check if the `allow_failure` property is enabled for the failing jobs.

Results. In addition to computing how often passing builds contain failing jobs, Figure 2 shows how the percentage of actively ignored failing jobs is distributed in passing builds that had at least one ignored failed job.

Observation 1*: 12% of passing builds have an actively ignored failure.* Of the 496,240 passing builds in our corpus, 59,904 builds had at least one actively ignored failure. Moreover, Figure 2 shows that in the passing builds that had at least one actively ignored failing job, the median percentage of ignored failing jobs is 25%.

In an extreme case, 87% of the jobs were actively ignored. We observe this in the *rubycas/rubycas-client*[7] project where the `allow_failure` property is set in 33 out of the 38 jobs.[8] Upon closer inspection, we observe that this is an example of the intended use of the `allow_failure` property. This build specifies eleven *Ruby* versions as runtimes and four *Gemfiles* for dependency management. Six of the combinations are explicitly excluded. Thus, 38 jobs are created for each build ($11 \times 4 - 6$). All of the 33 jobs that have the `allow_failure` property set fail. In subsequent builds, after several source code changes by the development team, all of these failing jobs begin to pass. Finally, the development team removes the `allow_failure` property from these jobs with an accompanying commit message that states that "builds should fail on released versions of ruby and rails". The development team only ignored failures while they improved their support for multiple ruby and rails versions.

On the other hand, the `allow_failure` setting can be misused. For example, in the *zdavatz/spreadsheet*[9] project, the `allow_failure` property, which is set in the initial build specification of the project, is never removed from the build specification throughout the five-year history of the project.[10] Furthermore, in our corpus, we detect 23 projects that had the `allow_failure` property set in all of their builds. These projects were not short-lived, with 31 to 769 builds in each project (median of 151). This suggests that although the intended purpose of the `allow_failure` property is to temporarily hide breakages, development teams do not always disable this property after it has been set, leaving the breakages hidden.

[6]https://docs.travis-ci.com/user/customizing-the-build/#Rows-that-are-Allowed-to-Fail
[7]https://github.com/rubycas/rubycas-client
[8]https://travis-ci.org/rubycas/rubycas-client/builds/5604025
[9]https://github.com/zdavatz/spreadsheet
[10]https://github.com/zdavatz/spreadsheet/blame/master/.travis.yml

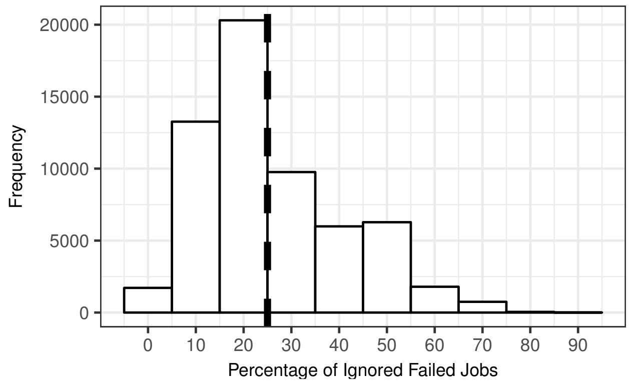


Figure 2: Percentage of ignored failed jobs in passing builds that had at least one ignored failed job across all projects. Up to 87% of the jobs are actively ignored.

> *Passing build outcomes do not always indicate that the build was entirely clean.*

3.2 Passively Ignored by Developers

Motivation. Build breakage is considered to be distracting because it draws developer attention away from their work to fix build-reported issues [11, 13, 19]. If development can proceed without addressing a build breakage, we suspect that the breakage is not distracting. Since these *passively ignored breakages* may introduce noise in analyses that assume that all breakages are distracting. We set out to analyze how often breakages are passively ignored.

Approach. To detect passively ignored breakages, we construct and analyze the directed graph of revisions from the version history that have been built using Travis CI.

(1) **Build Filtering.** We start by selecting the `git_trigger_commit` and the `git_prev_built_commit` fields of each build from TravisTorrent. The `git_trigger_commit` field refers to the revision within the repository that is being built. The `git_prev_built_commit` field refers to the revision that was the target of the immediately preceding build. Multiple builds may be associated with one `git_trigger_commit` because developers can configure Travis CI to run builds at scheduled time intervals, even if no new commits have appeared in the repository.[11] Builds can also be triggered by the Travis CI API, regardless of whether there are new commits in the repository.[12] We remove such duplicate builds by checking for builds that have *event_type* property set to `cron` or `api`. This reduces the number of builds from 680,209 to 676,408. Travis CI also triggers builds when *Git* tags are created even if the tagged commit has already been built. We remove builds that were triggered by tag pushes by checking for non-null values for the *tags* property. This reduces the number of builds to 659,048. However, there are

[11]https://docs.travis-ci.com/user/cron-jobs/
[12]https://docs.travis-ci.com/user/triggering-builds/

multiple builds remaining for one `git_trigger_commit` because manual build invocations can be made via the Travis CI web interface. These manual invocations cannot be distinguished from regular builds that were triggered by Git pushes. Therefore, when multiple builds are encountered for one `git_trigger_commit`, the earliest build is selected. This reduces the number of builds to 610,550.

(2) **Graph Construction.** Nodes in the graph represent build-triggering commits, while edges connect builds chronologically. All nodes are connected by edges from `git_prev_built_commit` node to `git_trigger_commit` node.

(3) **Graph Analysis.** We use the directed graph to identify build-triggering commits from which others branch. We select those branch point build-triggering commits that have a non-passing outcome. Then, we traverse all of the branches of such builds in a breadth-first manner to find the earliest build where the outcome is passing. Finally, we count the number of builds along the shortest path between the breakage branch point and the earliest fix.

Results. Figure 3 shows the broken builds that are at the branch points in the version history of the project. To some degree, developers passively ignored these failures by not immediately fixing them and continuing development in multiple paths.

Observation 2*: Breakages often persist after branching.* Of the 23,068 builds that are triggered by commits at branch points, 4,136 (18%) are broken. Of those commits that are branched when the build was broken, 3,426 builds (83%) are not fixed in the immediately subsequent build. These breakages are suspicious because developers have not immediately fixed these breakages and have continued development.

Figure 3 shows that commits are branched from up to twelve times when the build was broken. In the 13,102 builds that were not immediately fixed, several commits appear before the fix does. Figure 4 shows the maximum and median durations where the projects remained broken in the studied projects.

Observation 3*: Breakages persist for up to 423 days, and seven days on average, before being fixed.* In one extreme case, the *orbeon-/orbeon-forms* project[13] had 485 consecutive build breakages over 423 days before finally the breakage was addressed. Upon further investigation of this breakage, we find that the build is broken due to multiple test failures over time. By analyzing the commit messages of the broken builds in this sequence, we find only 10% of these commits mention fixing the broken build (# of Occurrence of each term: build=3, regression=2, test=46). However, near the end of the long build breakage sequence, two commits before the build started passing again, the developer has started skipping tests mentioning "For now, don't run integration and database tests".[14] This shows that the build breakages were not the focus of the development activity until the end of the sequence when they turned off the tests that were causing the breakage.

We find that 761 projects have breakages that persist for more than one day, 547 projects have breakages that persist for more than one week, and 227 projects have breakages that persist for more than one month before getting fixed. In eight projects, consecutive build breakages persist for more than one year before getting fixed. The overall median length of the failure sequences is five, while project-specific medians range between 2–29.

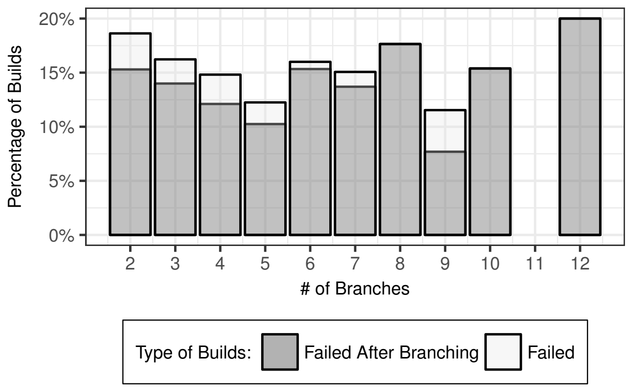


Figure 3: Developers branch out into multiple development paths (branches) even after build breakages. Percentage of broken builds at branch points are shown in white. Percentage of broken builds that continued to be broken after branching are shown in grey. There are no broken builds with 11 branches.

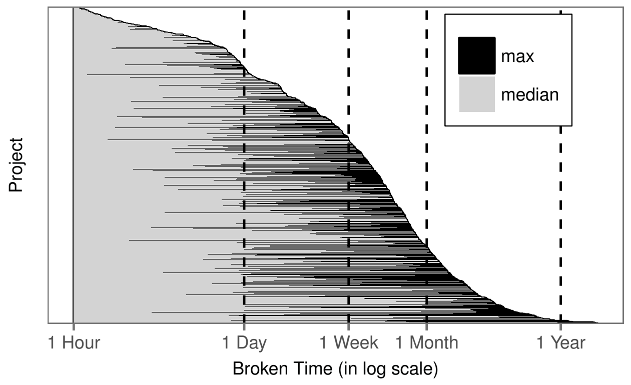


Figure 4: In some cases, builds can remain broken for 423 days. The graph shows the maximum and median durations that each project's build remained broken, ordered by the maximum duration.

In 83% of branches from broken builds, the breakage persists. These breakages persist for up to 485 commits.

3.3 Staleness of Breakage

Motivation. Developers can passively ignore breakages for different reasons. We identify the *staleness* of a build breakage (whether the project has encountered a given breakage in the past) as one of the reasons for ignoring a build breakage. A new breakage is different from a stale breakage because developers may have become desensitized to stale breakages.

[13] https://github.com/orbeon/orbeon-forms
[14] https://github.com/orbeon/orbeon-forms/compare/f137cfb555f1...eb1a8095a025

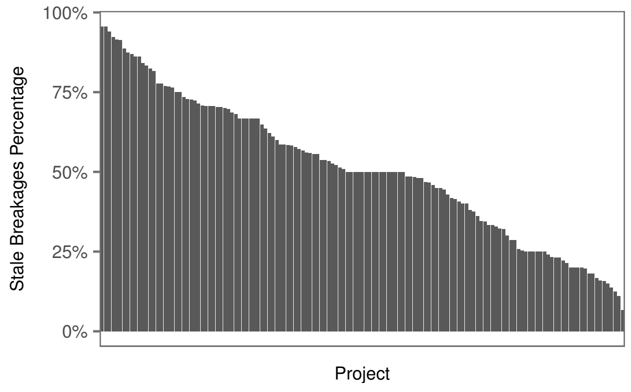


Figure 5: Percentage of stale breakages in each project can range from 7% to 96%.

Approach. In this section, we investigate how many times developers come across the same breakage repeatedly in the history of a project with respect to the length of build breakage sequences. These stale breakages can occur either consecutively or intermittently. Hence, we extend the *Maven Log Analyzer* developed by Macho et al. [15]. We use it to compare two Travis CI build jobs and check the similarity of the breakages. To make the comparison efficient, this is done in two steps. First, the logs of build jobs are parsed and checked if they are breaking due to the same reason (e.g., compilation failure, test execution failure, dependency resolution failure). If the reason for failures are equal then the details of the failure are also checked (e.g., if both breakages are due to compilation failure, check if the compilation error is the same).

Results. Figure 5 shows the percentage of stale build breakages in each project in descending order.

Observation 4*: 67% of the breakages (6,889 out of 10,816) that we analyze are stale breakages.* On the project level, staleness of breakages ranges from 7% to 96% with a median of 50%. In the *eirslett/frontend-maven-plugin*[15] project, where we observe the maximum percentage of stale breakages (96%), it was due to the same dependency resolution failure recurring in 23 builds.

Two of every three build breakages (67%) that we analyze are stale.

3.4 Signal-To-Noise Ratio

Motivation. In previous analyses, we find that build breakages that are ignored by developers and build successes that include ignored breakages can introduce noise in build outcome data. However, the overall rate of noise in build outcome data is not yet clear. Such an overview is useful for researchers who use build outcome data in their work, to better understand the degree to which noise may be impacting their analyses.

Approach. To quantify the proportion of noise in build outcome data caused by passively and actively ignored build breakage, we adopt the *Signal-To-Noise* ratio (SNR) as follows:

$$SNR = \frac{\#TrueBuildBreakages + \#TrueBuildSuccesses}{\#FalseBuildBreakages + \#FalseBuildSuccesses} \quad (1)$$

where *#TrueBuildBreakages* (i.e., signal) is the number of broken builds that are not ignored by developers, *#TrueBuildSuccesses* (i.e., signal) is the number of passing builds without ignored breakages, *#FalseBuildBreakages* (i.e., noise) is the number of broken builds that are ignored by developers, and *#FalseBuildSuccesses* (i.e., noise) is the number of passing builds with ignored breakages.

To compute *#FalseBuildBreakages*, a threshold t_c must be selected such that if the number of consecutive broken builds is above t_c, all builds in such sequences are considered false build breakages. Instead of picking any particular t_c value, we plot an SNR curve as the threshold (t_c) is changed.

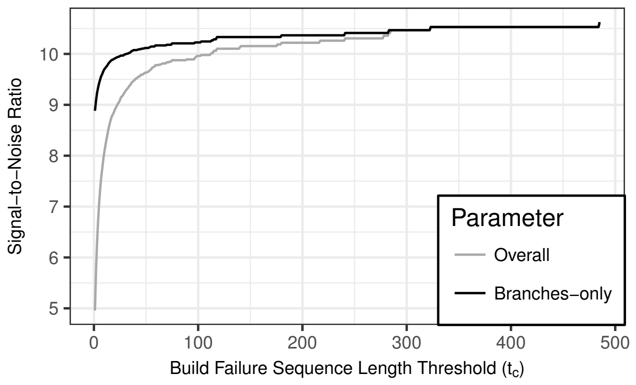


Figure 6: For every 11 builds there is at least one build with an incorrect status. The Signal-To-Noise ratio increases when a higher build breakage sequence length is chosen.

Results. Figure 6 shows the SNR curve for the subject systems.

Observation 5*: As t_c decreases from 485 to 1, the SNR decreases from 10.62 to 6.39.* Since *#FalseBuildSuccesses* is not impacted by t_c, the maximum SNR is observed when *#FalseBuildBreakages* is zero (i.e., when t_c is set to the maximum value). The minimum of SNR is observed when t_c is one and therefore all broken builds that are not immediately fixed are considered false build breakages. If false breakages are defined to be only in consecutive breakages with branches in them, the Signal-to-Noise ratio ranges from 10.19 to 10.62.

One in every 7 to 11 builds (9%–14%) is incorrectly labelled. This noise may influence analyses based on build outcome data.

4 HETEROGENEITY IN BUILD BREAKAGE DATA

The way in which builds are configured and triggered vary from project to project. This heterogeneity should be taken into consideration when designing studies of build breakage. Below, we demonstrate build heterogeneity using three criteria.

[15]https://github.com/eirslett/frontend-maven-plugin

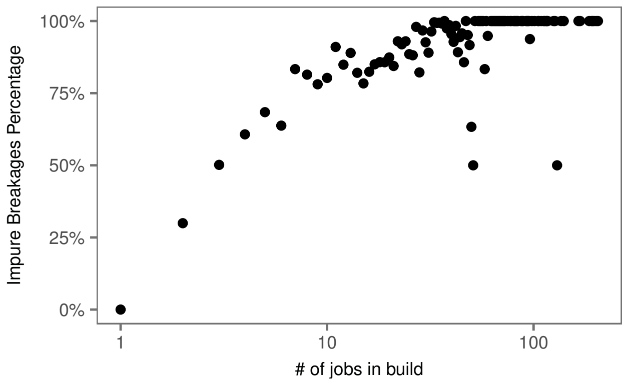


Figure 7: Percentage of impure build breakages increases with the number of jobs in each build.

4.1 Matrix Breakage Purity

Motivation. If a software project needs to be tested in multiple environments with different runtime versions, CI services like Travis CI provide the ability to declare these options in a matrix of *runtime, environment,* and *exclusions/inclusions* sections. A build will execute jobs for each combination of included runtimes and environments.

If a build is broken only within a subset of its jobs, the breakage may be platform- or runtime-specific. These environment-specific build breakages may need to be handled differently from the environment-agnostic breakages. Thus, we want to know the extent to which real breakages are environment specific.

Approach. To study the environments that are affected by a build breakage, we define *Matrix Breakage Purity* as follows:

$$\text{Matrix Breakage Purity} = \frac{\#FailedJobsInBuild}{\#AllJobsInBuild} \quad (2)$$

A *Matrix Breakage Purity* below one indicates that the jobs that were run in some environments passed. We compute the *Matrix Breakage Purity* for all builds in our dataset and count the number of builds with values below one. We label all builds that have a *Matrix Breakage Purity* below one as *impure build breakages.*

Results. Figure 7 shows how the percentage of impure build breakages varies with respect to the number of jobs per build.

Observation 6*: At least 44% of broken builds contain passing jobs.* Indeed, environment-specific breakages are almost as common as environment-agnostic breakages.

Given the difference in semantics between pure and impure build breakage, researchers should take this into account when selecting build outcome data for research. For example, in build outcome prediction, if prediction models are trained using data that treats environment-specific and environment-agnostic breakages identically, the model fitness will likely suffer. Moreover, the insights that are derived from the models will likely be misleading, since the two conflated phenomena will be modelled as one phenomenon.

Observation 7*: Builds with a greater number of jobs are more likely to suffer from impure build breakages.* Figure 7 shows that the number of jobs in a build and the percentage of broken builds that have passing jobs are highly correlated. A Spearman correlation test yields a ρ of 0.8, with $p = 2.2 < 10^{-16}$. While pure build breakage is common in builds with few jobs, when the number of jobs per build exceeds three, impure build breakage are more frequent than pure ones (i.e., impure breakage percentage > 50%).

Environment-specific breakage is commonplace. Once the number of jobs exceeds three, impure breakages occur more frequently than pure breakages.

4.2 Reason for Breakage

Motivation. Builds can break for reasons that range from style violations to test failures. Different types of failures have different implications. For example, while a style violation might be corrected easily, fixing a test failure might require time and effort to understand and address. Since subsequent analyses of build data should handle different types of breakages in different ways, we want to know how types of build breakage vary in reality.

Approach. To analyze the reasons for build breakage in our corpus, we extend the *Maven Log Analyzer* (MLA) [15]. Our extension first parses the Travis CI log file and extracts the sections of the log that correspond to executions of the *Maven* build tool. Then, each of these *Maven* executions are fed to MLA to automatically classify the status of each execution. In addition to the breakage types that were identified in the original work [15], our extended version of MLA also detects the build breakage types that were reported by Vassallo et al. [26] and Rausch et al. [18], as well as ten previously unreported breakage categories.

If MLA classifies all *Maven* executions within a broken build as successful, the build is labelled as a *Non-Maven* breakage. Non-Maven breakages are further classified as *Pre-Maven* if a failing command is detected in the Travis CI log before the *Maven* commands and *Post-Maven* otherwise.

In total, using our extended MLA, we classify 67,267 broken build jobs of projects that use *Maven* as the build tool.

Results. Table 1 classifies the broken *Maven* builds by reason.[16]

Observation 8*: Although a large proportion of build breakages are due to the execution of* Ant *from within* Maven*, most of these breakages belong to one project.* Table 1 shows that there are 15,850 instances of breakage where the external goal of executing an *Ant* build from within a *Maven* build failed. This accounts for 92.59% of the goal failed breakages in our corpus. However, this is an example of an anomaly that dissipates when examined more closely. Indeed, we find that all of these breakages occur in only two of the studied projects, the overwhelming majority (15,857) of which occur in the *jruby/jruby*[17] project. According to developer discussions, *Ant* is used inside the *Maven* build of *jruby/jruby* for executing tests.[18] However, this complex build setup, which requires 250MB of dependencies, causes build to fail intermittently. The developers hope that the breakages will not occur once the build is completely migrated to *Maven.*

Observation 9*: In our corpus, most breakage is due to commands other than main build tool,* Maven. We observe 41% (27,289) jobs are

[16]More details about reasons for breakage are available online: https://github.com/software-rebels/bbchch/wiki/Build-Breakages-in-Maven
[17]https://github.com/jruby/jruby
[18]https://gitter.im/jruby/jruby/archives/2016/05/27

Table 1: Distribution of Build Breakages in *Maven* Projects based on the Categories proposed by Vassallo et al. [26]. and Rausch et al. [18]. Global percentage of each category is shown in brackets.

Category	Subcategory	#	%	Projects
Dependency Resolution*		**2,257**	**(3.41%)**	**18**
Test Execution Failed	Unit	10,759	62.87%	165
	Integration	6,354	37.13%	18
	Total	**17,113**	**(25.89%)**	**171**
Compilation Failed	Production	2,015	87.08%	18
	Test	248	10.72%	12
	Total	**2,314**	**(3.50%)**	**47**
Goal Failed	Pre-processing	44	0.26%	4
	Static-Analysis	210	1.23%	10
	Dynamic-Analysis	8	0.05%	3
	Validation	33	0.19%	5
	Packaging	25	0.15%	4
	Documentation	25	0.15%	7
	Release Preparation	1	0.01%	1
	Deployment - Remote	120	0.70%	9
	Deployment - Local	7	0.04%	1
	Support	3	0.02%	1
	Ant inside Maven*	15,850	92.59%	2
	Run system/Java program*	70	0.41%	2
	Run Jetty server*	8	0.05%	1
	Manage Ruby Gems*	65	0.38%	1
	Polyglot for Maven*	32	0.19%	1
	Total	**17,119**	**(25.90%)**	**47**
Broken Outside Maven	No Log available*	1,554	5.69%	28
	Failed Before Maven*	808	2.96%	3
	Failed After Maven*	7,151	26.20%	46
	Travis Aborted*	16,141	59.15%	172
	Travis Cancelled*	1,635	5.99%	20
	Total	**27,289**	**(41.29%)**	**175**

* New build breakage categories that did not appear in prior work.

broken due to reasons other than *Maven* executions failing. This can be either due to a command that was executed by Travis CI (outside of *Maven*) returning an error, the user canceling the build, or Travis CI runtime aborting the build because it exceeded the allocated time.

Observation 10: *Only a small amount of breakage can be automatically fixed by focusing on tool-specific breakage.* For example, 2,257 build jobs are broken because dependency resolution has failed. This suggests that recent approaches that automatically fix dependency-related build breakage [15] will only scratch the surface of the build breakage problem. Moreover, Compilation and Test Execution failures only account for another 29.39% of the breakage in our corpus. Future automatic breakage recovery efforts should look beyond tool-specific breakages to the CI scripts themselves in order to yield the most benefit for development teams.

41% of the broken builds in our corpus failed due to problems outside of the execution of the main build tool. Since tool-specific breakage is rare, future automatic breakage recovery techniques should tackle issues in the CI scripts themselves.

4.3 Type of contributor

Motivation. Both core and peripheral contributors trigger builds. Since core contributors likely have a deeper understanding of the project than peripheral contributors, builds that are triggered by core contributors might have breakage rates and team responses

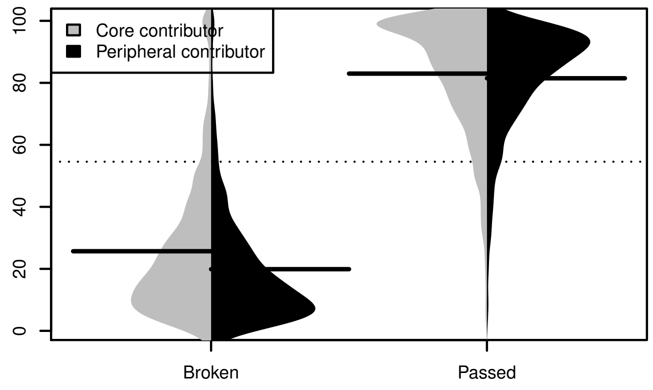


Figure 8: Percentage of broken and passing builds classified by contributor type. Horizontal black lines show the median values.

that differ from those of peripheral contributors. We set out to investigate the differences of build outcome, in these two categories of contributors.

Approach. For analyzing this dimension, we use the two main outcomes of each build (passed or failed) and whether the builds were triggered by a commit that was authored by a core team member. We use the core member indicator from the TravisTorrent dataset,[19] which is set for contributors who have committed at least once within the three months prior to this commit (`gh_by_core_team_member`). Then, we use the broken time and the length of broken build sequences to investigate the relationship between the contributor type and build breakage.

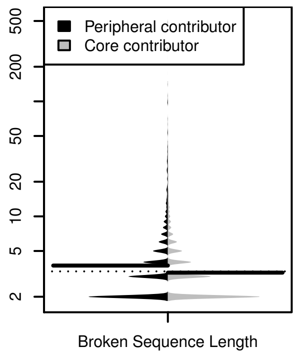
Results. Figure 8 shows how the percentage of build outcomes are distributed across projects classified by contributor type.

Observation 11: *Builds triggered by core team members are break significantly more often than those of peripheral contributors.* A Wilcoxon signed rank test indicates that breakage rates in core contributors are higher than those of peripheral contributors ($p = 1.28 < 10^{-8}$); however, the effect size is negligible (Cliff's $\delta = 0.13$). Due to having more experience, core team members in the development teams are assigned to complex tasks, which may explain why breakage rates tend to be a little higher. The Wilcoxon test is inconclusive when comparing rates of passing builds among core and peripheral contributors.

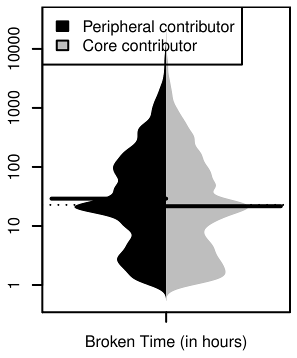
Figure 9 shows how long build breakages persist classified by contributor type. Figures 9a and 9b show the length of build breakage sequences in terms of commits and time, respectively.

Observation 12: *Breakages that are caused by core contributors tend to be fixed sooner than those of peripheral contributors.* A Wilcoxon signed rank test indicates that breakages caused by core contributors tend to persist for significantly less time than those of peripheral contributors ($p = 1.86 < 10^{-7}$); however, the effect size is negligible (Cliff's $\delta = 0.09$). Another Wilcoxon signed rank test indicates that breakages of core contributors persist for fewer consecutive builds than those of peripheral contributors ($p = 1.81 < 10^{-8}$); however, the effect size is also negligible (Cliff's $\delta = 0.09$).

[19]https://travistorrent.testroots.org/page_dataformat/



(a) **Chains of consecutive breakages caused by peripheral contributors tend to be longer.**



(b) **Build breakages caused by peripheral contributors take more time to repair.**

Figure 9: Build breakages caused by peripheral contributors remain broken significantly longer than those of core contributors. Horizontal black lines show the median values.

The longer time taken by peripheral contributors might be due to multiple attempts of trial and error before fixing a breakage, while core members might be able to identify the root cause of the breakage sooner. Therefore, it may be worthwhile for the researchers working on automatic build breakage repair to focus on build breakages that are caused by peripheral contributors.

Broken builds that are caused by core contributors tend to be fixed sooner than those of peripheral contributors.

5 IMPLICATIONS

We now present the broader implications of our observations for researchers and tool builders.

5.1 Research Community

Build outcome noise should be filtered out before subsequent analyses. Passing builds might contain breakages that are ignored. Long sequences of repeated breakages might be ignored by the developers as false breakages. If the noise due to false successes and false breakages is not filtered out, the results from prediction models may lead to spurious or incorrect conclusions.

Heterogeneity of builds should be considered when training build outcome prediction models. Some breakages are limited to specific environments while others are not. The reason for breakages vary from trivial issues like style violations to complex test failures. Breakages are often not caused by development mistakes, but by resource limitations in the CI environment. Indeed, build outcome includes many complex categories that can not be accurately represented in the prediction models using only a "broken" or "clean" label.

5.2 Tool Builders

Automatic breakage recovery should look beyond tool-specific insight. While recently proposed tools can automatically recover from tool-specific build breakage [15], we find that this category only accounts for a small proportion of CI build breakage in our corpus. Future efforts in breakage recovery should consider CI-specific scripts, for example, detecting those scripts that are at risk of exceeding the allocated time prior to execution.

Richer information should be included in build outcome reports and dashboards. Currently, build tools and CI services provide users with dashboards that show passing builds in green and broken builds in red. However, we found hidden breakages among passing builds and non-distracting breakages among broken builds. Moreover, heterogeneity of breakages introduce further complexities. Build outcome reporting tools and dashboards should consider providing more rich information about hidden, non-distracting, and stale breakages, as well as breakage purity and type.

6 THREATS TO VALIDITY

Construct Validity. Threats to construct validity refer to the relationship between theory and observation. It is possible that there are build failure categories that our scripts are unable to detect. By implementing the categories that were reported in prior work [15, 18, 26] and then manually checking a subset of logs along with their detected failure categories, we ensure most of the *maven* plug-ins and their failure categories are covered by our scripts.

There are likely to be other factors that introduce noise and variability in build outcomes. As an initial study, we focus on the aspects that we think demonstrate noise and heterogeneity of builds in this work. Our list of aspects is not intended to be exhaustive.

Internal Validity. Threats to internal validity are related to factors, internal to our study, that can influence our conclusions. In the analysis of passively ignored breakages, we associate continuous breakage of a build with developers ignoring the breakage. However, developers may be unsuccessfully attempting to fix these breakages during the breakage chain. We do not suspect that this is the most frequent explanation because we find several cases where the initial breakage has several branches (implying that several developers inherited the breakage). Although it can be assumed that these branches are created to fix the build, it is unlikely that twelve branches are created only for bug fixing. We further investigate the staleness of breakages, observing that the same build breakages are often repeated in these long chains.

External Validity. Threats to external validity are concerned with the generalizability of our findings. We only consider open source projects that use the Travis CI service and are hosted on GitHub. However, because GitHub is one of the most popular hosting platforms for open source software projects and Travis CI is the most widely adopted CI service among open source projects, our findings are applicable to a large number of open source projects. Similar to that, we only consider projects that use *Maven* for analyzing reasons for build breakage. However, *Maven* is one of the most popular build tools for Java projects [16] and therefore our findings are widely applicable. Nonetheless, replication studies using data from other hosting platforms, other CI services, and other build tools may provide additional insight.

7 RELATED WORK

In this section, we describe the related work with respect to build breakage and continuous integration.

7.1 Build Breakage

Build breakage has attracted the attention of software engineering researchers at many occasions during the past decade.

The rate at which builds are broken has been explored in the past. Kerzazi et al. [11] have conducted an empirical study in a large software company analyzing 3,214 builds that were executed over a period of six months to measure the impact of build breakages, observing a build breakage rate of 17.9%, which generates an estimated cost of 904.64 to 2034.92 person hours. Seo et al. [19] studied nine months of build data at Google, finding that 29.7% and 37.4% of Java and C++ builds were broken. Tufano et al. [21] found only 38% of the change history of 100 subject systems is successfully compilable and that broken snapshots occur in 96% of the studied projects. Hassan et al. [7] showed that at least 57% of the broken builds from the top-200 Java projects on GitHub can be automatically resolved.

To better understand and predict build breakage, past studies have fit prediction models. Hassan and Zhang [6] have demonstrated that decision trees based on project attributes can be used to predict the certification result of a build. Wolf et al. [27] used a predictive model that leverages measures of developer communication networks to predict build breakage. Similarly, Kwan et al. [12] used measures of socio-technical congruence, i.e., the agreement of the coordination needs established by the technical domain with the actual coordination activities carried out by project members, to predict build outcome in a globally distributed software team. In recent work, Luo et al. [14] have used the TravisTorrent dataset to predict the result of a build based on 27 features. They found that the number of commits in a build is the most important factor that can impact the build result. Dimitropoulos et al. [4] use the same dataset to study the factors that have the largest impact on build outcome based on K-means clustering and logistic regression.

For communicating the current status of the build, Downs et al. [5] proposed the use of ambient awareness technologies. They have observed by providing a separate, easily perceived communication channel distinct from standard team workflow for communicating build status information, the total number of builds increased substantially, and the duration of broken builds decreased. To help developers to debug build breakage, Vassallo et al. [25] propose a summarization technique to reduce the volume of build logs. For mitigating the impact of build breakage in the context of component-based software development, van der Storm [22] have shown how backtracking can be used to ensure that a working version is always available, even in the face of failure.

Broadly speaking, the prior work has treated build breakage as a boolean, pass or fail label. In this paper, we advocate for a more nuanced interpretation of build breakage that recognizes the noise in build outcome data and heterogeneity of build executions.

7.2 Continuous Integration

Recent work has studied adoption of CI in the open source community. Beller et al. [3] have released a dataset based on Travis CI and GitHub that provides easy access to hundreds of thousands of CI builds from more than 1,000 open-source projects. Efforts similar to this have made several studies of CI builds possible. By analyzing open source projects on GitHub and surveying developers, Hilton et al. [9] report on which CI systems developers use, how developers use CI, and reasons for using CI (or not).

The adoption of CI has an impact on other project characteristics. Hilton et al. [8] found that, when using CI, developers have to choose between speed and certainty, better access and information security, and more configuration options and greater ease of use. Vasilescu et al. [24] observe that CI adoption is often accompanied by a boost in the productivity of project teams.

Recent community interest in CI has yielded a resurgence of build breakage research. For example, Vasilescu et al. [23] studied 223 GitHub projects and found that the CI builds started by pull requests are more likely to fail than those started by direct commits. Beller et al. [2] have studied testing practices in CI of Java and Ruby projects, observing that testing is the most frequently occurring type of build breakage. Rausch et al. [18] identified the most common error categories in CI builds based on 14 open source Java applications. Zampetti et al. [28] have studied the usage of static analysis tools in 20 open source Java projects that are hosted on GitHub and using Travis CI. Vassallo et al. [26] compare the CI processes and occurrences of build breakages in 349 open source Java projects and 418 projects from a financial organization. Labuschagne et al. [13] have also observed that there are long stretches of broken builds due to projects not configuring Travis CI correctly or not examining the Travis CI output. To account for this, they removed the 20% of the projects that had the most and the fewest failures.

While prior work helps to understand build breakage and CI in practice, we focus on the trustworthiness of off-the-shelf historical CI build data. Our observations yield insights that can guide future studies of and tool development for build breakage in the CI context (see Section 5).

8 CONCLUSION

Automated builds are commonly used in software development to verify functionality and detect defects early in software projects. An off-the-shelf usage of build outcome data is implicitly susceptible to harmful assumptions about build breakage. By empirically studying build jobs of 1,276 open source projects, we investigate whether two assumptions hold. First, that build results are not noisy; however, we find in every eleven builds, there is at least one build with a misleading or incorrect outcome on average. Second, that builds are homogeneous; however, we find breakages vary with respect to the number of impacted jobs and the causes of breakage.

Researchers who make use of build outcome data should make sure that noise is filtered out and heterogeneity is accounted for before subsequent analyses are conducted. Build reporting tools and dashboards should also consider providing a richer interface to better represent these characteristics of build outcome data.

In future work, we plan to study how much of an impact noise and heterogeneity can have on common analyses of historical build data. We also plan to investigate whether breakage type varies with respect to contributor type and other commit factors.

REFERENCES

[1] Bram Adams and Shane McIntosh. 2016. Modern Release Engineering in a Nutshell: Why Researchers should Care. In *Proceedings of the International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 5. 78–90.

[2] Moritz Beller, Radjino Bholanath, Shane McIntosh, and Andy Zaidman. 2016. Analyzing the State of Static Analysis: A Large-Scale Evaluation in Open Source Software. In *Proceedings of the International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. 470–481.

[3] Moritz Beller, Georgios Gousios, and Andy Zaidman. 2017. TravisTorrent: Synthesizing Travis CI and GitHub for Full-Stack Research on Continuous Integration. In *Proceedings of the International Conference on Mining Software Repositories (MSR)*. 447–450. https://doi.org/10.1109/msr.2017.24

[4] Panagiotis Dimitropoulos, Zeyar Aung, and Davor Svetinovic. 2017. Continuous integration build breakage rationale: Travis data case study. In *International Conference on Infocom Technologies and Unmanned Systems (Trends and Future Directions) (ICTUS)*. https://doi.org/10.1109/ictus.2017.8286087

[5] John Downs, Beryl Plimmer, and John G. Hosking. 2012. Ambient awareness of build status in collocated software teams. In *Proceedings of the International Conference on Software Engineering (ICSE)*. https://doi.org/10.1109/icse.2012.6227165

[6] Ahmed E. Hassan and Ken Zhang. 2006. Using Decision Trees to Predict the Certification Result of a Build. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*. https://doi.org/10.1109/ase.2006.72

[7] Foyzul Hassan, Shaikh Mostafa, Edmund S.L. Lam, and Xiaoyin Wang. 2017. Automatic Building of Java Projects in Software Repositories: A Study on Feasibility and Challenges. In *Proceedings of the International Symposium on Empirical Software Engineering and Measurement (ESEM)*. https://doi.org/10.1109/esem.2017.11

[8] Michael Hilton, Nicholas Nelson, Timothy Tunnell, Darko Marinov, and Danny Dig. 2017. Trade-offs in continuous integration: assurance, security, and flexibility. In *Proceedings of Joint Meeting of the European Software Engineering Conference and the International Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 197–207. https://doi.org/10.1145/3106237.3106270

[9] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. 2016. Usage, costs, and benefits of continuous integration in open-source projects. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*. 426–437. https://doi.org/10.1145/2970276.2970358

[10] Peter Kampstra. 2008. Beanplot: A Boxplot Alternative for Visual Comparison of Distributions. *Journal of Statistical Software* 28, Code Snippet 1 (2008), 1–9. https://doi.org/10.18637/jss.v028.c01

[11] Noureddine Kerzazi, Foutse Khomh, and Bram Adams. 2014. Why Do Automated Builds Break? An Empirical Study. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*. https://doi.org/10.1109/icsme.2014.26

[12] Irwin Kwan, Adrian Schroter, and Daniela Damian. 2011. Does Socio-Technical Congruence Have an Effect on Software Build Success? A Study of Coordination in a Software Project. *IEEE Transactions on Software Engineering (TSE)* 37, 3 (2011), 307–324. https://doi.org/10.1109/tse.2011.29

[13] Adriaan Labuschagne, Laura Inozemtseva, and Reid Holmes. 2017. Measuring the cost of regression testing in practice: a study of Java projects using continuous integration. In *Proceedings of the Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. ACM Press, 821–830. https://doi.org/10.1145/3106237.3106288

[14] Yang Luo, Yangyang Zhao, Wanwangying Ma, and Lin Chen. 2017. What are the Factors Impacting Build Breakage?. In *Proceedings of the Web Information Systems and Applications Conference (WISA)*. IEEE. https://doi.org/10.1109/wisa.2017.17

[15] Christian Macho, Shane McIntosh, and Martin Pinzger. 2018. Automatically Repairing Dependency-Related Build Breakage. In *Proceedings of the International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. 106–117. https://doi.org/10.1109/SANER.2018.8330201

[16] Shane McIntosh, Meiyappan Nagappan, Bram Adams, Audris Mockus, and Ahmed E. Hassan. 2015. A Large-Scale Empirical Study of the Relationship between Build Technology and Build Maintenance. *Empirical Software Engineering* 20, 6 (2015), 1587–1633.

[17] Audris Mockus. 2007. Software Support Tools and Experimental Work. In *Empirical Software Engineering Issues. Critical Assessment and Future Directions: International Workshop, Dagstuhl Castle, Germany, June 26-30, 2006. Revised Papers*, Victor R. Basili, Dieter Rombach, Kurt Schneider, Barbara Kitchenham, Dietmar Pfahl, and Richard W. Selby (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 91–99. https://doi.org/10.1007/978-3-540-71301-2_25

[18] Thomas Rausch, Waldemar Hummer, Philipp Leitner, and Stefan Schulte. 2017. An Empirical Analysis of Build Failures in the Continuous Integration Workflows of Java-Based Open-Source Software. In *Proceedings of the International Conference on Mining Software Repositories (MSR)*. 345–355. https://doi.org/10.1109/msr.2017.54

[19] Hyunmin Seo, Caitlin Sadowski, Sebastian Elbaum, Edward Aftandilian, and Robert Bowdidge. 2014. Programmers' build errors: a case study (at google). In *Proceedings of the International Conference on Software Engineering (ICSE)*. 724–734. https://doi.org/10.1145/2568225.2568255

[20] E. Burton Swanson. 1976. The Dimensions of Maintenance. In *Proceedings of International Conference on Software Engineering (ICSE)*. 492–497. http://dl.acm.org/citation.cfm?id=800253.807723

[21] Michele Tufano, Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. 2016. There and back again: Can you compile that snapshot? *Journal of Software: Evolution and Process (JSEP)* 29, 4 (2016), e1838. https://doi.org/10.1002/smr.1838

[22] Tijs van der Storm. 2008. Backtracking Incremental Continuous Integration. In *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*. https://doi.org/10.1109/csmr.2008.4493318

[23] Bogdan Vasilescu, Stef van Schuylenburg, Jules Wulms, Alexander Serebrenik, and Mark G. J. van den Brand. 2015. Continuous integration in a social-coding world: Empirical evidence from GitHub. **Updated version with corrections**. *CoRR* abs/1512.01862 (2015). arXiv:1512.01862 http://arxiv.org/abs/1512.01862

[24] Bogdan Vasilescu, Yue Yu, Huaimin Wang, Premkumar Devanbu, and Vladimir Filkov. 2015. Quality and Productivity Outcomes Relating to Continuous Integration in GitHub. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the International Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 805–816.

[25] Carmine Vassallo, Sebastian Proksch, Timothy Zemp, and Harald C. Gall. 2018. Un-Break My Build: Assisting Developers with Build Repair Hints. In *Proceedings of the International Conference on Program Comprehension (ICPC)*. To appear.

[26] Carmine Vassallo, Gerald Schermann, Fiorella Zampetti, Daniele Romano, Philipp Leitner, Andy Zaidman, Massimiliano Di Penta, and Sebastiano Panichella. 2017. A Tale of CI Build Failures: An Open Source and a Financial Organization Perspective. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*. 183–193. https://doi.org/10.1109/icsme.2017.67

[27] Timo Wolf, Adrian Schroter, Daniela Damian, and Thanh Nguyen. 2009. Predicting build failures using social network analysis on developer communication. In *Proceedings of the International Conference on Software Engineering (ICSE)*. https://doi.org/10.1109/icse.2009.5070503

[28] Fiorella Zampetti, Simone Scalabrino, Rocco Oliveto, Gerardo Canfora, and Massimiliano Di Penta. 2017. How Open Source Projects Use Static Code Analysis Tools in Continuous Integration Pipelines. In *Proceedings of the International Conference on Mining Software Repositories (MSR)*. 334–344. https://doi.org/10.1109/msr.2017.2

[29] Mahdis Zolfagharinia, Bram Adams, and Yann-Gaël Guéhéneuc. 2017. Do Not Trust Build Results at Face Value - An Empirical Study of 30 Million CPAN Builds. In *Proceedings of the International Conference on Mining Software Repositories (MSR)*. 312–322. https://doi.org/10.1109/msr.2017.7