Travis, Why Do the Builds Keep Failing?: An Analysis of CI Failure Causes and Severity

Colin Cummings
Computer Science and Engineering
University of Nebraska-Lincoln
Lincoln, NE
ccummings3@huskers.unl.edu

Easton Joachimsen
Computer Science and Engineering
University of Nebraska-Lincoln
Lincoln, NE
ejoachimsen@huskers.unl.edu

Matt Sichterman
Computer Science and Engineering
University of Nebraska-Lincoln
Lincoln, NE
mattsichterman@huskers.unl.edu

***Abstract*—Continuous Integration is a practice that has become widely used and is a standard in the industry. Travis CI can be dropped into any GitHub repository and automate various tasks including the build and test processes. With that much convenience at face value, there also come hiccups when builds fail which can get tedious. This paper strives to quantify the different types and reasons that builds fail and how quickly a successful subsequent build is completed. Using TravisTorrent's data set [2], we apply various data analysis and data visualization techniques to compare and portray failing builds to understand their cause and severity based on the time to the next successful build. Our findings suggest that tests cause the most most build failures in CI pipelines and we categorize each failure/error into high medium, or low severity.**

***Index Terms*—continuous integration, continuous delivery, build failures, Travis CI, GitHub**

I. Introduction

Continuous integration (commonly abbreviated ”CI”) has become a standard practice in modern software engineering. CI helps developers by more frequently implementing their code changes into the shared branch of a code-base. The changes are instantly validated by the CI pipeline, where automated tests are ran to help ensure that changes do not break anything and, in the case that they do, notify developers so that the changes can be reevaluated and again resubmitted for validation. Validation often includes unit and integration tests to ensure that everything including classes, modules, and functions are working as expected within an application.

Finding the relationship between CI and what kinds of problems are being caught in the pipelines is important to determine the complexity of bugs being caught. We utilize the TravisTorrent data set [2] and analyze past work to help us determine what types of problems are most often causing pipeline failures. We also analyze the time needed to fix these problems in efforts to uncover specific data to show what most often is causing failures. The impact of our work is that our results can help developers have a better idea of what to look out for, what typically causes the most problems, and where to look first when pipelines are failing and their development team is pointing fingers and scrambling to make the needed fixes.

The work of Beller et al. [2] produced a resource that has been crucial to continued research on how CI is utilized by developers. Through their production of Travis-Torrent, researchers have explored how users interact with Travis-CI through the analysis of builds and how they pass or fail. Existing research refines the overarching data-set into extremely low-level patterns such as test cases, static code analyzers, and user experience. Despite the other related studies surrounding CI including how unit tests impact the way that CI is integrated, contributors' involvement and the effect on the build process, the effect of static code analysis in continuous integration pipelines, and many others there is still work needed to determine what is causing failures most often and how complex the problems found are with respect to the time to the next successful build after a failure.

We aim at answering the following primary research question: **What types of problems are causing continuous integration pipelines to fail most often?**

To answer this question, we performed a quantitative analysis on the TravisTorrent data set [2] to better understand what kinds of problems are most commonly caught in the CI pipelines, with an emphasis on the average time it took to recover from identified build failures.

The paper makes the following contributions:

- Analyzes the **frequency** of continuous integration failures due to failing test cases vs. other causes.
- Categorizes the **severity** of build failures/errors based on time between a failure/error and the subsequent successful build.

The paper is organized as follows: Section II discusses related work in CI and important findings in related work. Section III introduces our study goal, details our research questions, describes the data set used in our study, states the hypotheses that we developed from our research questions, elaborates on the methodology that we used, and discusses the variables in the study using greater detail. Section IV contains information from our analysis and the results of the study including the results found specifically for each research question. Section V has further discussion of our results including threats to validity. In Section VI, we conclude our findings and present opportunities for future work.

II. Related Work

CI has become an integral quality assurance practice [4], but what exactly has sparked this interest in CI and how is it used? In software engineering, continuous integration (CI) is a practice in which contributions from multiple developers are integrated back into a shared mainline multiple times every day. Through the use of this process the quality of their contributions is also able to be monitored. Thanks to the work of Beller et al. [2], research of CI has become possible through their production of TravisTorrent, a publicly available data set composed of Travis CI data liked back to GitHub repositories.

Beller et al. [3] provided further insights into the importance of tests and their contributions to how CI is used. Within their work, they provide a purely quantitative analysis on the impact and usage of tests within Travis CI. The research was derived from the TravisTorrent data set [2] (which the authors created) and limited their analysis window to projects that were written with Ruby and Java. When examining the results of their tests, they state that testing is the single most important reason for pipelines to break, more prevalent than compile errors, missing dependencies, build cancellations, and provisioning problems together [3].

Rebouças et al. [11] explored the impacts of a subject's involvement within a project to the CI method in their analysis of CI. They explore data within the TravisTorrent data set to try to find a pattern that suggests that non-casual developers are more likely to introduce code to Travis CI that will cause the build to fail. Within their analysis, they combine both the data from the data set along with data extracted from the Travis API to gather statistics of users, their contribution percentages, and the number of builds with their respective status (pass or fail). Their findings suggest that in eighty-five percent of the cases, there is no representative difference between contributions placed from casual and non-casual contributors, meaning that being casual is not a strong indicator for creating failing builds [11].

Zampetti et al. [13] set a goal to further investigate static code analysis and its role in the continuous integration practices. Their investigation efforts were divided into three primary questions: Which tools are being used and how are they configured for the CI, What types of issues make the build fail or raise warnings, and After how long are broken builds and warnings resolved [13]. Upon analyzing, they found that there is huge variability in terms of how strict the projects had their analysis tools set to. Certain tools were more strict than others and were meant to accomplish different tasks. In the end, this was found to be a key point when analyzing data. The separation of tools used to interrupt builds and the tools used to warn about possible bugs or vulnerabilities is apparent. Results of these findings greatly impacted the remaining questions of the paper in the way that the second and third research questions were very dependent on the first. That being said, the findings for *RQ2* are very similar to the findings of *RQ1*. As for *RQ3*, they found that the time taken to resolve broken builds caused by static code analyzers was almost immediately fixed (in median within 8 hours), in most cases by actually fixing the problem in the source code and only in few cases by modifying the build script [13].

Through the use of CI, defects within coding projects can easily be detected through the inclusion of tests within their pipelines. This brings up the question, what type of tests detect defects more efficiently, integration tests or unit tests? Orellana et al. [10] explore further into this topic exploring areas including which type of tests expose more defects, do defects exposed by one type of test take longer to resolve, and which type of defect requires more coordination to resolve. Upon conducting the analysis of the TravisTorrent data set, Orellana et al. [10] found that unit tests are not only responsible for uncovering more defects in code, they are also the result of the most time and coordination taken to resolve the issues. Through these findings, it is clear that the MSR community should be careful when analyzing how software teams handle defects [10].

CI has many usages to track statistics within projects as well. Atchison et al. [1] analyzed some of these trends, specifically to track seasonal productivity within development teams on projects using Travis CI. To extract growth trends and periodic behavior related to the number of builds in a continuous integration environment, Atchison et al. put together a study that analyzes the frequency of CI builds relative to the day of the week along with other statistics associated with those builds. The analysis was conducted both on a macro level analyzing all records in the TravisTorrent data set and on a micro-level analyzing only Apache Drill projects [1]. After the analysis was complete, the stats indicated clear spikes of CI usage earlier in the week and a spike in bugs introduced into certain projects towards the end of the week.

The integration of code from a different source can be a quick workaround for solving problems encountered when writing code for certain systems. However, in some cases this integration could lead to unexpected errors that might occur at run time. To explore how these external code integrations impact CI, Muylaert et al. [9] put together a study to answer questions like how often do these code integrations lead to defects, what is the level of effort required to fix these defects, and what type of files are these defects most prevalent in. To perform an analysis to find the answers to the questions raised, Muylaert et al. combined data from both GHTorrent and TravisTorrent data sets. After refining the data pulled from the data sets the analysis concluded that code integrations are not only responsible for a much smaller percentage of build breakage than regular commits, but are also extremely easy to fix taking on average less than one day to resolve and less than 10 lines of code [9]. The findings also suggest that the majority of these defects can be resolved in the source code rather than in other places such as test code.

Predicting the result of a build could be valuable to developers to save them time by detecting bugs and defects before building [7]. Luo et al. [7] realized the importance of understanding the factors that cause builds to fail and conducted a study that analyzed the correlation between certain features within a pipeline and the result of the build. After applying correlation analysis on 27 different features and their build results from the TravisTorrent data set, Luo et al. [7] were able to conclude that failure is most common for builds that have a higher density of commits. Additionally, they found that a higher number of files added, deleted, or modified also increases the possibility for a build to break. After the initial analysis of the primary causes for build failures, they applied their findings to four different prediction models to discover which prediction model is best suited to discover build failures before the failure happens. After comparing SVM, Decision Tree, Random Forest, and Logistic Regression models they concluded that, in this case, SVM is the best-suited model with the highest average prediction accuracy [7].

Similar to the work of Luo et al. [7], Madeyski et al. [8] also explore possible ways that CI build failures can be predicted. They took the same approach as Luo et al. gathering resources from TravisTorrent and GHTorrent, however instead of analyzing the data sets they combined them into a central data set. This can be used in future research projects that investigate the relationships of factors like revision amount, modified lines, author identifications, and much more to their build status. Through their work of combining the data gathered, they have produced their own publicly available data set called ”Continuous Defect Prediction” as a CSV file.

Xia et al. [12] continued with this work to explore how prediction models can be utilized to predict the outcome of a build. Within the TravisTorrent data set, an analysis was run on a total of 126 open-source projects with approximately 300,000 build records [12]. A total of 9 different classifiers were utilized to train certain prediction models, more than most other related studies. Additionally an online scenario, a previously unexplored method, was introduced into the study in which builds are ordered and predicted chronologically according to the starting time of the build [12]. The findings indicated that cross-validation prediction performance is superior when used with AUC compared to the online model in almost every scenario.

Through a very prestigious project, Hassan et al. [6] explored how HireBuild, a tool to help developers save time by automatically resolving build failures caused by build scripts, performs when ran on a set of 175 build failures from the TravisTorrent data set. Within the 175 build failures, a mere 24 cases were determined to be reproducible. When running HireBuild on the 24 cases, a total of 11 of them were able to be resolved automatically [6]. Through the examination of time spent resolving issues with HireBuild it was determined that HireBuild has a higher average time required to resolve issues when compared to manual action at 44 minutes compared to 42 minutes [6]. Further insights within Hassan et al. include fixed size analysis and failure analysis.

In the paper *Noise and Heterogeneity in Historical Build Data: An Empirical Study of Travis CI*, by K. Gallaba, C. Macho, M. Pinzger and S. McIntosh, they find that on average two out of three build breakages are stale, meaning the breakage occurs multiple times in a project's build history. These findings are relevant since they show that most build breakages are not necessarily dealt with immediately, and tend to be ignored as developers become desensitized to the breakage [5].

In this paper we study the primary causes of build failures through the analysis of the TravisTorrent data set. The primary motivation of this work is to discover the most frequent cause for build failures and analyze the severity for each type of failure with a quantitative measure of hours taken to resolve the issues. This work is very closely related to the other works discussed prior in terms of finding the most frequent cause for build failures and attempting to save the time of the developers. Other papers explore areas such as prediction algorithms, impacts of developer experience, time taken to resolve errors in tests, usage of unit test vs integration test, and much more, however there remains a gap in research that analyzes both the most frequent cause for failures in the Java/Ruby project space and also quantifies the cause severity which is where our research is valuable.

III. Study Design

The following subsections will detail our study including our study goal, research questions, data set, hypotheses, methodology, and important variables. Each section explains our processes and gives important context to understand when we present our results and discussion in later sections.

A. Study Goal

Following our main research question presented in the introduction, we defined the main goal of our quantitative analysis by applying the Goal Question Metrics template as so:

Analyze the TravisTorrent data set
for the purpose of evaluating pipeline failures
with respect to the type of errors causing the failures and severity of the errors considering the time elapsed until the next successful build
from the point of view of the developer
in context of Travis CI data mined and linked back to GitHub

Based on the metrics listed above, we have defined severity as follows:

- **Low:** issues from failures take less than one hour to resolve.
- **Medium:** issues from failures take less than eight hours to resolve.
- **High:** issues from failures take eight or more hours to resolve.

B. Research Questions

With this, we defined and explored research questions as follows:

RQ1. Do failing unit tests cause the most failures in continuous integration pipelines?
RQ2. Does the majority (i.e., greater than 50%) of continuous pipeline failures result from issues that take less than 1 hour to resolve?

RQ3. Does the size of a commit or the number of jobs in a pipeline correlate with pipeline failures more frequently?

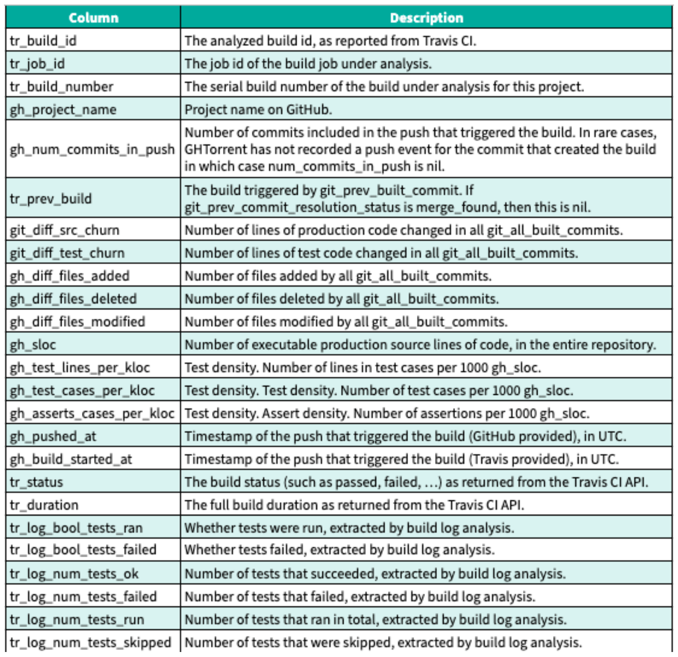
These research questions are important to fully understand CI and give developers additional context into the causes and observed severity of the issues commonly identified in the pipelines.

C. Data Set

The TravisTorrent data set contains a deep analysis of the project source code, process and dependency status of 1,359 projects. The data set comes filtered in an effort to only include projects that that are non-forks, non-toys, have greater than 10 watchers on GitHub, and have a Travis CI history of greater than 50 builds. In addition, 936 of the projects in the data set use Ruby and 423 projects use Java. Some notable projects in the data set include Ruby on Rails, Google Guava and Guice, Chef, RSpec, Checkstyle, ACIIDoctor, Ruby and Travis [2].

TravisTorrent provides very convenient access to the data set, which can be queried directly using Google Big Query or managed and manipulated locally using a SQL dump download or CSV file. We opted to use each of these techniques in an attempt to maximize our productivity during the study.

The data provides us with an extensive amount of columns giving us an abundance of information about the Travis CI data and associated GitHub source code data. More information can be seen on the TravisTorrent website. The table in Fig. 1 below highlights the columns and their descriptions most relevant to our study:



Column	Description
tr_build_id	The analyzed build id, as reported from Travis CI.
tr_job_id	The job id of the build job under analysis.
tr_build_number	The serial build number of the build under analysis for this project.
gh_project_name	Project name on GitHub.
gh_num_commits_in_push	Number of commits included in the push that triggered the build. In rare cases, GHTorrent has not recorded a push event for the commit that created the build in which case num_commits_in_push is nil.
tr_prev_build	The build triggered by git_prev_built_commit. If git_prev_commit_resolution_status is merge_found, then this is nil.
git_diff_src_churn	Number of lines of production code changed in all git_all_built_commits.
git_diff_test_churn	Number of lines of test code changed in all git_all_built_commits.
gh_diff_files_added	Number of files added by all git_all_built_commits.
gh_diff_files_deleted	Number of files deleted by all git_all_built_commits.
gh_diff_files_modified	Number of files modified by all git_all_built_commits.
gh_sloc	Number of executable production source lines of code, in the entire repository.
gh_test_lines_per_kloc	Test density. Number of lines in test cases per 1000 gh_sloc.
gh_test_cases_per_kloc	Test density. Test density. Number of test cases per 1000 gh_sloc.
gh_asserts_cases_per_kloc	Test density. Assert density. Number of assertions per 1000 gh_sloc.
gh_pushed_at	Timestamp of the push that triggered the build (GitHub provided), in UTC.
gh_build_started_at	Timestamp of the push that triggered the build (Travis provided), in UTC.
tr_status	The build status (such as passed, failed, …) as returned from the Travis CI API.
tr_duration	The full build duration as returned from the Travis CI API.
tr_log_bool_tests_ran	Whether tests were run, extracted by build log analysis.
tr_log_bool_tests_failed	Whether tests failed, extracted by build log analysis.
tr_log_num_tests_ok	Number of tests that succeeded, extracted by build log analysis.
tr_log_num_tests_failed	Number of tests that failed, extracted by build log analysis.
tr_log_num_tests_run	Number of tests that ran in total, extracted by build log analysis.
tr_log_num_tests_skipped	Number of tests that were skipped, extracted by build log analysis.

Fig. 1. A table showing the columns of interest from the data set with a brief description.

To make our analysis more feasible, we decided to take a subset of the data set. We decided to take the code from 3 projects: facebook/presto, reactjs/react-rails, and Shopify/shopify_api. The data from these projects was then filtered to include only rows where tests were actually ran as part of the pipeline.

D. Hypotheses

Based on the related work that we reviewed and personal experience that we have accumulated in the space of continuous integration, we expect test failures to most commonly cause the pipelines to fail. Therefore, we formulated the following null hypothesis to check the frequency of unit tests being the cause of failures in the Travis CI pipelines:

- $H0_{FREQ}$: The frequency in which unit tests cause continuous integration pipelines to fail is no greater than the frequency of any other single cause of failure.

We formulated the following null hypothesis to check the relative severity (i.e., time taken until next successful build) of issues that cause continuous pipeline failures:

- $H0_{SEV}$: The severity of issues caught in continuous integration pipelines is low (i.e., take less than 1 hour to resolve) no more than 50% of the time.

Finally, we formulated the following null hypothesis to check which factor correlates with build failures more frequently:

- $H0_{COR}$: The size of a commit correlates with overall pipeline failures no more frequently than just the number of jobs within a pipeline.

The alternative hypotheses were formulated as follows:

- $H1_{FREQ}$: The frequency in which unit tests cause continuous integration pipelines to fail is greater than the frequency of all other single causes of failure.
- $H1_{SEV}$: The severity of issues caught in continuous integration pipelines is low (i.e., take less than 1 hour to resolve) a majority of the time.
- $H1_{COR}$: The size of a commit correlates with overall pipeline failures more frequently than just the number of jobs within a pipeline.

We formulated $H0_{FREQ}, H0_{SEV}$, and $H0_{COR}$ to study RQ1, RQ2 and RQ3, respectively.

E. Methodology

Since we used a pre-established data set, our study focused mainly on quantitative analysis. First, we imported the TravisTorrent data set into a Jupyter notebook via Google Colab in the form of a CSV file. Google Colab gives us the ability to run our analysis in the cloud with free, dedicated GPUs while utilizing Python as our programming language of choice. To focus our attention on the most important properties, we filtered the data set quite a bit to only include what we saw as most vital to our results. To do so, we used the Python package called ”pandas” to help us better organize and aggregate the data.

After organizing the data, we decided to follow a fairly structured process to analyze the data in three main steps, following our research questions closely:

Step 1: Identify the cases in which tr_status equals ”failed” and determine the cause of the failure. To do so, we considered metrics such as if tests failed, the number of tests failed, and the overall number of tests. This analysis allowed us to easily visualize the cause of failures and identify the frequency of failing pipelines with respect to test failures and other factors.

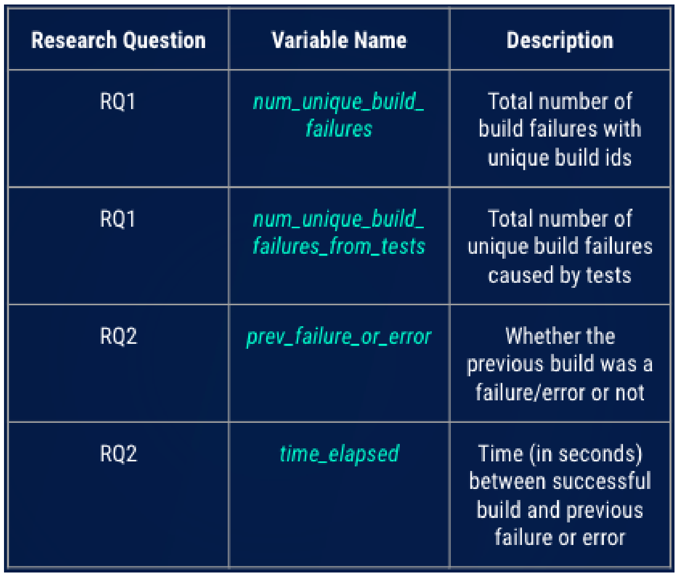
Step 2: Identify the cases in which the Travis CI pipeline ”failed” and determine the elapsed time until the next successful build. To do so, we considered factors such as tr_status (the build status), tr_prev_build which provides information about the previous build, gh_pushed_at (timestamp of the push that triggered the build provided by GitHub), gh_build_started_at (timestamp of the push that triggered the build provided by Travis CI), and more.

Step 3: Compare the factors associated with each build failure in an effort to find if the number of jobs or size of commit correlates with pipeline failures more frequently. To do so, we used visualization techniques and clustering to identify a correlation in the mentioned factors to pipeline failures.

We used the ”matplotlib” library to generate visualizations to better show trends in the data. We also used descriptive statistics to show the distribution of data.

F. Variables

The variables of importance are detailed in Fig. 2 .



Research Question	Variable Name	Description
RQ1	*num_unique_build_failures*	Total number of build failures with unique build ids
RQ1	*num_unique_build_failures_from_tests*	Total number of unique build failures caused by tests
RQ2	*prev_failure_or_error*	Whether the previous build was a failure/error or not
RQ2	*time_elapsed*	Time (in seconds) between successful build and previous failure or error

Fig. 2. Table showing the important variables for each research question analyzed.

The variables for RQ1 include *num_unique_build_failures* and *num_unique_build_failures_from_tests*. These variables track unique builds where *tr_status = ”failed”* and where tests were the cause of the failure so that we can use the count to determine the frequency of the failure causes. The variables for RQ2 include *prev_failure_or_error* and *time_elapsed*. These variables determine the cases in which a successful build’s previous status was ”failed” or ”errored”, and if so calculate the time elapsed between the failed/errored build and the successful build in order for us to categorize the severity.

IV. Analysis and Results

In this section, we will present information about the analysis and share the results found for our research questions. We will state the results of each research question independently in order to clearly portray our findings. Notice that we do not present results for RQ3 due to time constraints during our analysis. We will discuss this in more detail in Section VI.

A. RQ1 Results: ”Do failing unit tests cause the most failures in continuous integration pipelines?”

Our analysis for RQ1 included checking where *tr_status = ”failed”*. We identified 645 total unique pipeline failures that matched this constraint. From there, we evaluated which of those cases were due to failing test cases. We found that 407 of the failures were due to test cases while 238 resulted from other cases. Fig. 3 shows the frequency of build failures from tests vs. other cases in a pie chart.

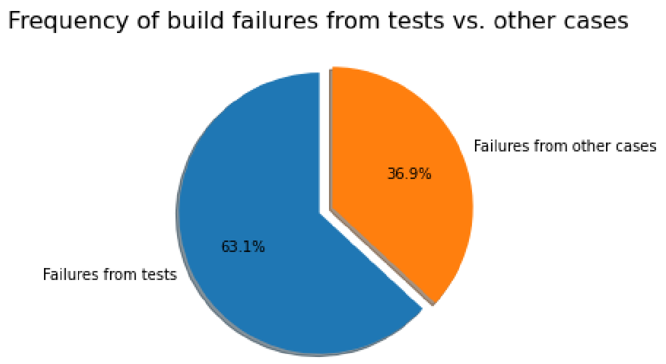


Fig. 3. Pie chart showing the frequency of build failure causes.

B. RQ2 Results: ”Does the majority (i.e., greater than 50%) of continuous pipeline failures result from issues that take less than 1 hour to resolve?”

RQ2 tackled a question that was a bit more involved. Our analysis included detecting successful builds in which the previous build was a failure or error by checking if *tr_status = ”failed” or ”errored”* on the row of the previous build. 300 cases among the three projects were identified where a passing build was preceded by a ”failing” or ”errored” pipeline. The *time_elapsed* was calculated for each of these 300 cases and can be seen in Fig. 4 and Fig. 5.

These results show that the median *time_elapsed* of 3.3 hours lies within our medium complexity range. The box plot also suggests a skewed distribution in which there are a number of outliers. In order to categorize our results into the low, medium, and high severity categories that we established, we created a pie chart to show the results. Of the 300 cases analyzed, 73 were of low severity (i.e., less than 1 hour to

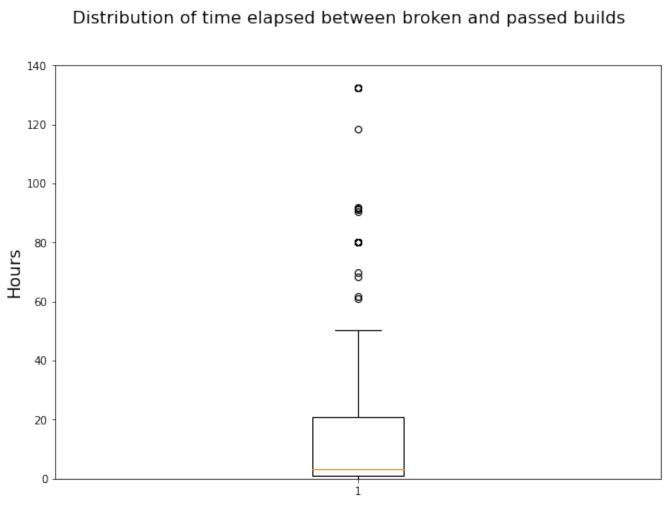
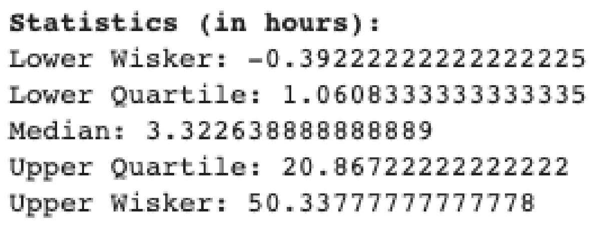


Fig. 4. Box plot showing the distribution of time elapsed between broken and passed builds.



```
Statistics (in hours):
Lower Wisker: -0.3922222222222225
Lower Quartile: 1.0608333333333335
Median: 3.322638888888889
Upper Quartile: 20.86722222222222
Upper Wisker: 50.33777777777778
```

Fig. 5. Descriptive statistics (in hours) of the distribution of time elapsed between broken and passed builds.

resolve), 108 were of medium severity (i.e., less than 8 hours to resolve), and 119 were of high severity (i.e., 8 or more hours to resolve). The results are visualized in Fig. 6.

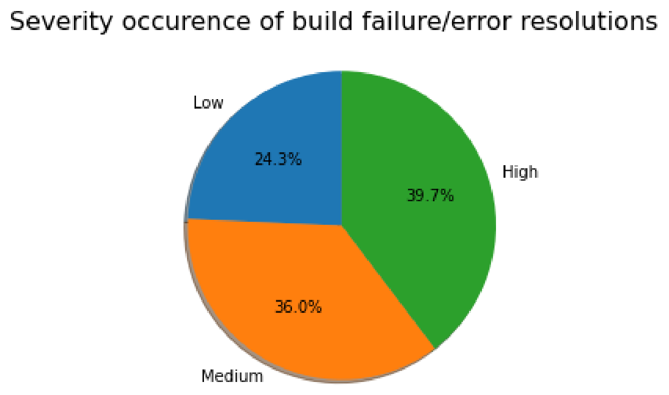


Fig. 6. Pie chart showing the severity occurrence of build failures/errors.

V. Discussion and Implications

In the following subsections, we answer the research questions, address our hypotheses, and discuss implications drawn from our work. We also elaborate on the threats to validity of our results including some limitations of our study.

A. RQ1 Discussion

Let's take another look at RQ1: *Do failing unit tests cause the most failures in continuous integration pipelines?* Based on our results the answer to this question is ”Yes”, as tests caused 63.1% of the failures that we evaluated. For that reason, we reject $H0_{FREQ}$ which states: *The frequency in which unit tests cause continuous integration pipelines to fail is no greater than the frequency of any other single cause of failure.* Thus, we accept $H1_{FREQ}$ which states *The frequency in which unit tests cause continuous integration pipelines to fail is greater than the frequency of all other single causes of failure.* Of the three projects evaluated, we found that when tests were included in the build they were the reason of the build failure most often.

B. RQ2 Discussion

RQ2 poses the following question: *Does the majority (i.e., greater than 50%) of continuous pipeline failures result from issues that take less than 1 hour to resolve?* Based on our severity categorizations we found that in the 300 cases analyzed: **high** severity errors occur most often, **medium** severity errors occur next often, and **low** severity errors actually occur the least often. This is different from what we expected, and we accept $H0_{SEV}$ which states *The severity of issues caught in continuous integration pipelines is low (i.e., take less than 1 hour to resolve) no more than 50% of the time.* With the existence of clear outliers in our data, stale builds is an intersting topic that may be worth evaluating in the future though we did not look into them during this study. Of the 300 cases evaluated in which successful builds were preceded by erroring or failing builds, we found that errors are most commonly in the **high** severity category.

C. Implications

The following implications can be concluded based on our results:

1) Based on RQ1 results, **tests** cause the most failures in continuous integration pipelines.
2) Based on RQ2 results, continuous integration build errors and failures are of **high** or **medium** severity a majority of the time, rather than of **low** severity.

D. Threats to Validity

Internal Validity: When selecting projects to analyze we decided to analyze data based on quality over quantity, thus our analysis consists of only three project spaces. This could be seen as an internal threat to validity as only choosing three projects could easily sway the results of the analysis depending on the projects selected. If we were to exchange a project that heavily relies on unit tests with a project that has an extremely strict static code analysis, our findings could yield a different outcome.

External Validity: We analyzed a data set specific to projects that used GitHub and Travis CI. Therefore, it may pose a threat to validity when generalizing the results for projects that use something different than Travis CI for their continuous integration. The projects being analyzed were likely all completing different tasks and functionalities, this can lead to a threat of interaction of setting and treatment, since various projects may have inconsistencies in their usage and dependability on continuous integration within each project.

External Validity: Through the research presented in this paper, we discovered that the primary cause for build failures was unit tests, however we only analyzed projects that were written with Java and Ruby. This aspect might pose a threat to our external validity as our findings are not applicable to projects written in any other language apart from the Java/Ruby languages.

External Validity: Other than the languages of the projects that we analyzed, it is also worth noting that the projects selected (facebook/presto, reactjs/react-rails, and Shopify/shopify_api) are all developed by large companies and corporations. This may also pose a threat to external validity due to the fact that there are no mid- to low-tier projects included in our study, making the application of our analysis to smaller projects invalid.

Construct Validity: When selecting the projects to undergo our analysis, we only selected projects that included test cases within their CI pipelines. This poses a threat to construct validity in the way that our analysis will always include the possibility for tests to fail and does not include certain possibilities for other causes. Because of this aspect our findings could be skewed.

VI. Conclusion and Future Work

In this paper, we discussed our method and results of analyzing three projects' data from the TravisTorrent data set [2] including facebook/presto, reactjs/react-rails, and Shopify/shopify_api. We conducted a quantitative analysis to determine the **frequency** of continuous integration failures due to failing test cases vs. other causes, and the **severity** of build failures/errors based on categories that we established considering the time between a failure/error and the subsequent successful build. There are still a number of questions surrounding the usage statistics of Travis CI. Through our findings, we have determined that tests cause the most failures in continuous integration pipelines and discovered that continuous integration build errors and failures are of high or medium severity a majority of the time, rather than of low severity.

Additional research into the topics of CI is still needed and questions like RQ3, stating *Does the size of a commit or the number of jobs in a pipeline correlate with pipeline failures more frequently?* have yet to be answered in the context of severity. Future work could analyze the factors that contribute to build failures like the size of the commit or the number of jobs ran in a pipeline and contribute those factors to our severity categorizations to discover what kinds of variables lead to severe failures. Another aspect of future work is to expand the breadth of our analysis, in which others could verify our findings by running analysis on the entire TravisTorrent data set [2] (or more than just three projects).

References

[1] A. Atchison, C. Berardi, N. Best, E. Stevens, and E. Linstead. A time series analysis of travistorrent builds: To everything there is a season. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 463–466, 2017.

[2] M. Beller, G. Gousios, and A. Zaidman. Travistorrent: Synthesizing travis ci and github for full-stack research on continuous integration. In *Proceedings of the 14th working conference on mining software repositories*.

[3] M. Beller, G. Gousios, and A. Zaidman. Oops, my tests broke the build: An explorative analysis of travis ci with github. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 356–367, 2017.

[4] P. Duvall, S. Matyas, and A. Glover. *Continuous Integration: Improving Software Quality and Reducing Risk*. Addison-Wesley Professional, first edition, 2007.

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

[6] F. Hassan and X. Wang. Hirebuild: An automatic approach to history-driven repair of build scripts. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 1078–1089, 2018.

[7] Y. Luo, Y. Zhao, W. Ma, and L. Chen. What are the factors impacting build breakage? In *2017 14th Web Information Systems and Applications Conference (WISA)*, pages 139–142, 2017.

[8] L. Madeyski and M. Kawalerowicz. Continuous defect prediction: The idea and a related dataset. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 515–518, 2017.

[9] W. Muylaert and C. De Roover. Prevalence of botched code integrations. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 503–506, 2017.

[10] G. Orellana, G. Laghari, A. Murgia, and S. Demeyer. On the differences between unit and integration testing in the travistorrent dataset. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 451–454, 2017.

[11] M. Rebouças, R. O. Santos, G. Pinto, and F. Castor. How does contributors' involvement influence the build status of an open-source software project? In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 475–478, 2017.

[12] J. Xia and Y. Li. Could we predict the result of a continuous integration build? an empirical study. In *2017 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, pages 311–315, 2017.

[13] F. Zampetti, S. Scalabrino, R. Oliveto, G. Canfora, and M. Di Penta. How open source projects use static code analysis tools in continuous integration pipelines. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 334–344, 2017.