CISC 834 Group Assignment Fall 2022

Kazi Amit Hasan[1], **Brennan Cruse** [1], **Chris Yeung** [1], **Victoria Armstrong** [1]

[1] School of Computing, Queen's University, Ontario, Canada
kaziamit.hasan@queensu.ca, brennan.cruse@queensu.ca,
chris.yeung@queensu.ca, victoria.armstrong@queensu.ca

Abstract

In this project, we replicated the methodology used by Seo et al. on the top 5 largest projects in the TravisTorrent data set (Beller, Gousios, and Zaidman 2022). These projects were written in Python or Ruby, contrasting the original author's analysis of build patterns in C++ and Java. We investigated how often builds fail, why they fail, and how long it takes to resolve these failures. Our replication corroborated the findings in (Seo et al. 2014).

Introduction

In this replication study, we used the TravisTorrent data set to answer the research questions outlined in Programmers' Build Errors: A Case Study (at Google). We compared 5 projects: 2 projects written in Ruby, and 3 projects written in Python. In total, we analyzed 33,532 builds to answer the following three research questions:

RQ1: *How often do builds fail?*
RQ2: *Why do builds fail?*
RQ3: *How long does it take to fix broken builds?*

We found that the majority of build failures were due to testing case failures. We found that while core team members typically produced fewer build errors, analysis shows that these results were not statistically significant. However, we believe this is likely due to our limited number of projects observed. We also analyzed build-failure resolution time and determined that issues were solved faster in Python over Ruby. Our results correlated with findings found in the original work, and highlight the need for further investigation into why builds fail.

Data Set

We used the publicly available TravisTorrent data set (Beller, Gousios, and Zaidman 2022). Beller, Gousios, and Zaidman created this data set based on Travis CI and GitHub, giving access to over 1000 projects. Given the scope of this assignment, we selected the top 5 data sets sorted by number of data points within the Travis Torrent data set. Of these top 5 projects, 3 were written in Python: `DataDog/dd-agent`, `pyca/cryptography`, and `getsentry/sentry`. 2 of the top 5 projects were written in Ruby: `opf/openproject`, and `bundler/bundler`. Within the dataset, each row represents a build job executed in Travis, and contains 55 corresponding data fields describing the build.

Study

In order to build on the results found by Seo et al. in their paper, Programmers' Build Errors: A Case Study (at Google) we replicated their research questions and methodology on a new data set. The following sections outline the steps that were taken to answer each of the 3 research questions.

RQ1: How often do builds fail?

To answer the first research question, we partitioned the projects into 2 categories for analysis: just the Python projects, and just the Ruby projects. We also maintained a collection of all 5 projects together for comparison. We then looked at the number of builds within each language and project. Next, the number of build passes, failures, errors, or cancellations were counted for each project. Unlike the data in the original paper, the TravisTorrent data set did not give us information on which developer made the build/commit. The fact that this data is missing could be for anonymity, as a large number of projects were collected together from Travis CI and GitHub. This contrasts the original paper where the authors completed a case study. at the company where they are currently employed, likely making data easier to access internally.

To resolve this issue, we chose to analyze our results based on whether or not the developer was a core-team member or not. The build status was further subdivided into 2 categories: whether or not the commit was authored by a core team member or a non-core team member. Next, we calculated the percentage occurrence of each build status in the project split by team member status. We then calculated the mean failure (breakage) and passing rates for each team member group and language. We used a Mann-Whitney U tests to determine whether there is a significant difference between breakage and passing rates between core versus non-core team members.

RQ2: Why do builds fail?

For this research question, we looked to examine what type of errors are responsible for the majority of failures. To determine this, we first ensured that we had a large enough sample population to perform further analysis. Similarly to RQ1, we divided the data into 3 categories: just Python projects, just Ruby projects, and all 5 projects together. Within each of these 3 categories, we isolated all instances of failure, then selected the failures with unique build IDs.

Unlike the data used in the original paper (Seo et al. 2014), the TravisTorrent data did not give us specific error messages for analysis. Because of this, we completed our analysis comparing unit-test failures to all other build failures. We were not given information as to what types of failures are included in this other category. This comparison was also completed for all 3 categories.

RQ3: How long does it take to fix broken builds?

Lastly, to answer RQ3 we looked at the length of time between initial build failure and a successful build. Since each data point consists of a job, we first grouped the data by it's build ID. Next, we established a column representing the build chain. This contained a list of build IDs from first failure to a successful build. We then calculated the resolution time in the same fashion as the original paper (Seo et al. 2014). We subtract the start time of the first failure and the duration of the first build from the start time of the successful build. This gave us a measurement for resolution time in seconds. Similarly to the original paper, we then eliminated any samples where the build time was greater than 12 hours. This roughly accounts for a developer changing tasks or heading home for the day. As mentioned in RQ2, we did not have access to specific error messages, so we could not group our data by error message like they did in the paper. Therefore, we analyzed resolutions times across languages.

Results

In the following section, we address each of the research questions and compare them to the results found by Seo et al..

RQ1: How often do builds fail?

In total, our data set contained 33,532 builds. Of these, 21,157 were from the 3 Python projects, and 12,375 were from the 2 Ruby projects. Figure 1 shows the percentage of builds that failed across our 3 categories, contrasting build failures from core team members and non-core team members. Similarly, Figure 2 shows the percentage of builds that passed.

The analysis shows that the build breakage rates are lower in core team members and build passing rates are lower in non-core team members. This finding contradicts the findings reported by Gallaba et al., which shows that builds triggered by core team members break significantly more often than those of peripheral contributors. Our results can be explained by the familiarity of the core developers with the existing code and dependencies. Developers with less experience with the project may be more likely to make errors causing the builds to fail. However, our results were not statistically significant, which is likely attributed to the extremely small sample size.

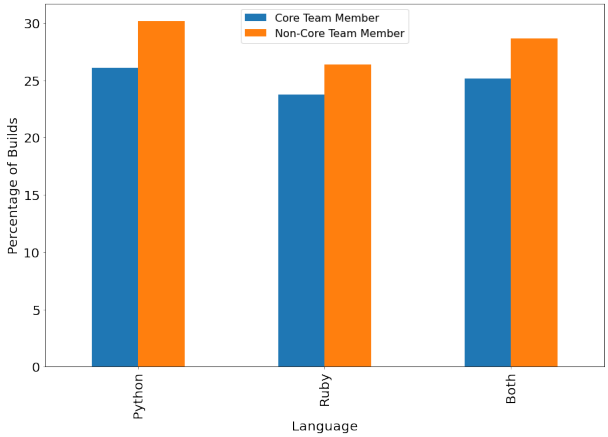

Fig. 1: Percentage of Failed Builds by Language

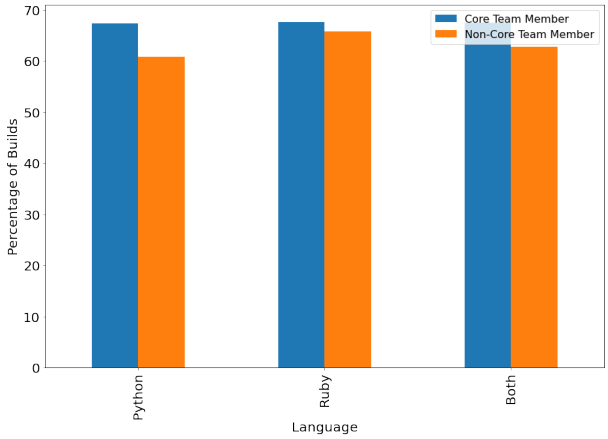

Fig. 2: Percentage of Passed Builds by Language

These results can be compared to the results from Seo et al.. In their work, they found that developers with very low or very high failure rates were not regular developers in this language. In the case of the original authors, it appears that to some extent, unfamiliarity with a given language can produce a higher number of failures. Extrapolating this idea to our results, there is a potential that unfamiliarity with the nuances of a given project could lead to more build failures. As mentioned previously, we did not find statistically significant differences between core team and non-core team members, but we posit that this difference may be statistically significant given a larger sample size.

In addition, the original authors found that there were more build failures in C++ projects, versus Java projects. They attributed this, in part, to the use of an IDE when programming in Java. We did not observe the same differences between the two languages. In the case of Ruby and Python, it is quite common to use an IDE and the languages

are high-level, object oriented languages.

RQ2: Why do builds fail?

Through the methodology outlined in the previous section, we observed the number of unique failures to determine their cause. As mentioned, this data set does not provide information to specific error messages, so we used a Boolean column to determine if it was a unit-test failure or another failure. Insights into what these other failures consist of is not given, which is a short coming of the dataset.

	Ruby	Python	Both
Test Case Failures	2,265	38,949	41,214
Other Failures	891	22050	22,941
Total Unique Failures	3,156	60,999	64,155

TABLE I: Failure Type by Language

Overall, all 5 projects had a total of 9,256 build failures. Of these, 57.57% were failed test cases, as shown in Figure 5. For projects completed in Ruby, there was a total of 3,156 unique build failures, with 77.77% of these failures coming from testing cases, as shown in Figure 4. For projects completed in Python, there were 60,999 unique build failures, with 63.85% of these failures coming from testing cases, as shown in Figure 3. Table I summarizes the numbers used to calculate these percentages. In both languages, we can see that the majority of failures was from testing, rather than a syntactic error for example. This can be partially attributed to the use of an IDE which allows the developer to catch minor errors before she even attempts to build.

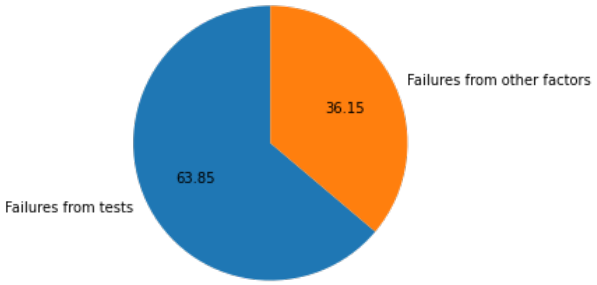

Fig. 3: Distribution of Failure Type for Python Projects

In comparison to the original paper, the most obvious difference is the lack of error message information. Without insight into error messages, we cannot produce the same taxonomy. However, they observed that the most common error type for both C++ and Java were dependency errors. This would be classified in our category of failed tests, which is also the majority in our case. In order to complete a stronger comparison, it would be highly beneficial to examine the specific error messages for each build failure. Additionally, a larger number of projects more comparable to the original paper's data set would also facilitate a more robust comparision.

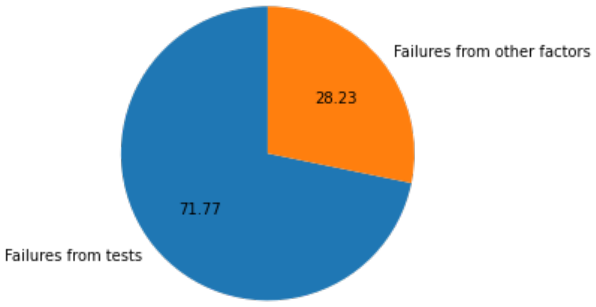

Fig. 4: Distribution of Failure Type for Ruby Projects

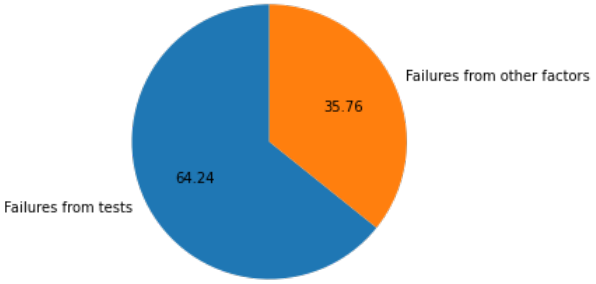

Fig. 5: Distribution of Failure Type for Both Projects

RQ3: How long does it take to fix broken builds?

By observing our calculated resolution times, we can gain insight into our third research question. In Table II we can see that resolutions times were longer for projects written in Ruby, with a mean resolution time of 13,617 seconds, or just under 4 hours. In contrast, the mean resolution time for Python projects was 7,248 seconds, or approximately 2 hours. Looking at the standard deviation, we can see that there is more dispersion in the resolution times for the two projects that used Ruby, versus the 3 projects completed in Python. Additionally, in Table III, we can observe that the distribution of build steps in both languages appears very similar. This means that the difference in resolution time between Ruby and Python is likely derived marginally by build, rather than via jobs within builds.

Figure 6 visualizes the differences in resolution time between Python and Ruby projects. We can see that for Python projects, the results are skewed to the left, whereas Ruby projects follow a more normal distribution. In Figure 6 it becomes clear that there are a number of builds that took almost 12 hours to resolve. There are fewer of these builds for Python projects. This may be because of a smaller project sample size for Ruby projects, or it may mean that we need to consider a different cut off for developer's changing tasks or going home for the evening, given that the standard work day is 8 hours - 4 hours longer than the 12 hour cut off suggested by the original authors.

In comparison to the original work, resolution times were significantly longer in our case. This is likely owing to the fact that we were unable to break down our dataset into the resolution of specific errors, given our limited

	Python	Ruby
Mean	7,248	13,617
Std	6,353	9,441
Min	23	47
25%	2,360	7,846
50%	4,937	11,132
75%	11,568	14,775
Max	39,563	43,173

TABLE II: Resolution Time Statistics

Descriptive statistics of build resolution time of Python versus Ruby programming languages. Values represent seconds from the end of the first build failure to the beginning of the successful build. Values greater than twelve hours are removed from the dataset to account for extreme cases (e.g. the developer went home for the day).

Steps	Python	Ruby
2	90.1%	88.7%
3	9.6%	9.8%
4+	3.3%	1.5%

TABLE III: Resolution Steps Statistics

Distribution of steps required to achieve a successful build, given that the first build fails. For example, if a successful build takes three steps to complete, this means two failed associated builds occurred first.

information. We can see by our fastest fix time for both Python and Ruby that there are some small fixes that take a relatively short amount of time. As mentioned in the original paper, it would be interesting to explore the type of error that can be resolved so quickly and whether this fix can be automatically generated. This would free up time for developers to focus their attention on larger issues.

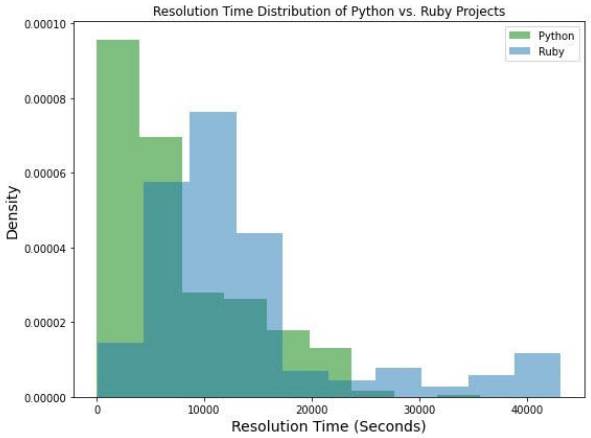

Fig. 6: Resolution Time Distributions

Histogram of resolution time for Python versus Ruby programming languages. Green represents Python projects and blue represents Ruby projects. Since sample quantities from each language differ, density is in contrast to absolute value counts.

Conclusion

Through our analysis of 5 projects from the TravisTorrent (Beller, Gousios, and Zaidman 2022) dataset, we have been able to replicate some of the findings in Programmers' Build Errors: A Case Study (at Google). While we did not find significant differences between core team member and non-core team member build failures, our results suggest that exploring a larger data set might indicate that those more familiar with the project produce fewer build failures. We also observed that the majority of build failures came from specific test case failures, which is similar to the results from the original paper. We did not find the same differences in languages that Seo et al. found in their work, however this can be attributed to the fact that Ruby and Python are more similar programming languages that C++ and Java. Lastly, we observed that resolution times for build failures were significantly higher than that of the original paper, however, we could not break down our data by error message.

Overall, our results have also highlighted the importance of quantifying and understanding build patterns and practices (Seo et al. 2014). The fact that our dataset did not provide specific test cases and error messages did impact our results, but this further impresses the need to understand the specifics behind *why* builds are failing. Although it is beyond the scope of this assignment, it would be interesting to investigate a larger number of projects.

References

Beller, M.; Gousios, G.; and Zaidman, A. 2022. TravisTorrent.

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

Seo, H.; Sadowski, C.; Elbaum, S.; Aftandilian, E.; and Bowdidge, R. 2014. Programmers' build errors: a case study (at google). In *Proceedings of the 36th International Conference on Software Engineering*, 724–734.