# How Trustworthy is Your CI Accelerator? A Comparison of the Trustworthiness of CI Acceleration Products

Zhili Zeng[1], Tao Xiao[2], Maxime Lamothe[3], Hideaki Hata[4], Shane McIntosh[1]
[1]*University of Waterloo,* [2]*Nara Institute of Science and Technology,* [3]*Polytechnique Montréal,* [4]*Shinshu University*
[1]{z75zeng, shane.mcintosh}@uwaterloo.ca; [2]tao.xiao.ts2@is.naist.jp
[3]maxime.lamothe@polymtl.ca; [4]hata@shinshu-u.ac.jp;

***Abstract*—The practice of Continuous Integration (CI) allows developers to quickly integrate and verify projects modifications. Thus, CI acceleration products are a boon to developers seeking rapid feedback. However, if outcomes vary between accelerated and non-accelerated settings, the trustworthiness of the acceleration is called into question.**

**In this paper, we study the trustworthiness of two CI acceleration products, one based on program analysis (PA) and the other on machine learning (ML). We re-execute 50 failing builds from ten open-source projects in non-accelerated (baseline), PA-accelerated, and ML-accelerated settings. We find that when applied to known failing builds, PA-accelerated builds more often (43.83 percentage point difference across ten projects) align with the non-accelerated build results. We conclude that while there is still room for improvement for both CI acceleration products, the selected PA-product currently provides a more trustworthy signal of build outcomes than the ML-product.**

## I. Introduction

Continuous Integration (CI) is a popular practice in modern software development. Accordingly, substantial effort has been invested in improving the performance of each phase of the CI process [1]. Several strategies exist to accelerate the CI process [2], by, e.g., caching build environments, inferring dependencies, skipping CI phases, skipping CI altogether, and accelerating the CI testing phase. Invariably, CI acceleration achieves speed-up by omitting steps (or jobs) during the CI process by either determining that (a) artifacts can be shared between CI jobs; or (b) the outcome and output of the step is unlikely to change based on the modified code. The techniques have led to the recent emergence of CI acceleration products.[1]

There are families of CI acceleration products. Program Analysis (PA) acceleration products rely on rule-based analysis conducted before the build process to determine safe ways to accelerate subsequent builds. For example, those products may build dependency graphs to record and store information from preceding builds and then use this information to accelerate subsequent builds by, e.g., skipping irrelevant test cases [3]. CI acceleration products based on Machine Learning (ML) rely on historical tendencies to determine when steps can be safely skipped. For example, those products may feed historical data into an ML algorithm to train a model whose predictions could minimize the test suite [4]. Aiming at different optimization targets, the ML-product can provide recommended subsets of tests of varying levels of aggressiveness to accelerate builds. Each CI acceleration family, and product, has costs and benefits.

Even if a product performs well in the trade-off between time costs and benefits, if it mislabels change sets (e.g., a faulty build passes) [5], then it may result in more work for developers [6]. Therefore, we systematically evaluate the trustworthiness of two products with respect to non-accelerated builds. There are two types of errors that an acceleration approach can make: (a) builds where expected CI outcomes pass, but actual CI outcomes fail; and (b) builds where expected outcomes fail, but actual outcomes pass. Previous research has shown that the CI outcomes of CI acceleration are consistent [7]. However, builds can have failing outcomes for multiple reasons, and these may not be consistent when using CI acceleration. This can allow some bugs to slip through CI when acceleration is in use.

Therefore, in this paper, we focus on the effect of CI acceleration on failing builds. Aimed at testing phases in the build, we use the full range of parameter settings for the PA[2] and ML[3] products, and replay the builds of 50 failing commits from ten large and active open-source projects in PA-accelerated, ML-accelerated, and non-accelerated settings. The studied PA product infers dependencies and constructs a graph during preceding builds, which is in turn leveraged to accelerate subsequent builds. The studied ML product provides recommended subsets of tests based on the outcomes from classifiers to accelerate builds. The PA and ML based techniques provide an average acceleration of 80.98% and 75.96% in our studied projects, respectively. This results in a total of 100,000 builds. This benchmark allows us to answer the following research question:



[1]https://www.msystechnologies.com/test-automation-accelerator/

[2]https://yourbase.io

[3]https://www.launchableinc.com/predictive-test-selection/

**RQ: How does the trustworthiness of CI acceleration products compare?**

Outcome: While the performance of neither product shows a clear trend across our studied projects, the PA-product is more trustworthy, producing rates of agreement with non-accelerated counterparts that are 4–76 percentage points higher than the ML-product. Furthermore, the most trustworthy ML parameter settings vary so broadly that the best (top 10) settings are only consistent for at most 11–25 commits out of 50 within the studied projects, and 14–18 commits out of 50 across the studied projects.

We conclude that while there is still room for improvement for both products, future work is needed to combat the tendency of ML-based acceleration to produce unstable and untrustworthy results.

## II. Study Design

### A. Data Filtering

We first retrieve a dataset of GitHub repositories from Google BigQuery.[4] The dataset contains the activity and property information for the repositories on GitHub.

**Select Python projects (DF1)**: To mitigate the influence of different programming languages on our experimental results, we select projects that use a single programming language. We focus on Python because it is the only language that is supported by both of our selected acceleration products. We query projects that have 'Python' as a field within the 'language' table in the BigQuery dataset. The query returns the projects that predominantly use the Python programming language (i.e., Python makes up the majority of the source code). After applying our first filter, 549,098 projects survive.

**Select pytest projects (DF2)**: After removing all non-Python projects, we use a second filter to select projects that use the pytest framework. We select the pytest framework because previous research demonstrated that it has a more stable performance profile than other testing frameworks [8]. After applying our second filter, 29,849 projects survive.

The lack of variegation with respect to community, programming language and test framework might be considered an external threat to the validity of this study. We provide the details in our online appendix.[5]

### B. Project Ranking

Although our experimental procedures are largely automated, repeating the experiment on thousands of systems is untenable. We therefore apply a ranking procedure to systematically select a set of studied projects.

**Compute projects relevance measures (PR1):** We inspect the number of commits, files and test cases, which respectively correspond to the repositories' activity, scale of production code, and scale of testing code. Repositories with a large number of commits and files contain more code changes, and are more likely to offer an adequate volume of data for validation. The number of test files is likely associated with the speed at which a build completes. We expect the time consumed by the build of a studied project to be large enough to perform a meaningful analysis of trustworthiness.

**Transform measures to ranking (PR2):** To obtain the impact of metrics during project selection, we sum the above indicators, and rank the surviving projects in descending order.

**Select diversified projects (DF3):** To obtain a diverse set of projects from which reliable conclusions can be drawn, we exclude projects from previously sampled domains, selecting the next highest-ranked project from another domain as a replacement. We also manually excluded extreme cases with less than 10 test cases but with a lot of source code (i.e., over 20,000 files). Finally, we obtain 10 studied projects, from different domains, shown in Table I.

### C. Data Extraction

After selecting the studied projects, we extract their data using the Data Extraction (DE) procedure shown in Figure 1. We describe each step below.

**Extract changesets (DE1):** CI acceleration operates on changesets. Thus, we extract changesets from the Git repositories of our studied projects. We separately select 50 consecutive commits that pass the build and 50 non-consecutive commits that fail for each studied project by walking the commit history in reverse chronological order. The initial commit is the failed commit in the extracted commit sets of each studied project. The number of studied changesets is an internal threat to validity. The exact listing of studied commits and the discussion about this threat to validity are available in our online appendix.[5]

**Label commits with CI outcomes (DE2):** We manually label each of the selected commits according to CI outcomes (e.g., pass or fail). We conduct this manual labelling because of the inconsistency in CI checks for different commits. The CI checks[6] in this paper refer to GitHub's CI feedback for a repository, which contains the passed/failed outcomes from CI services like GitHub Actions, CircleCI and Codecov. We obtain relevant information by checking the label listed after the commit date. Each commit has a different number of checks, which varies with commit branches and build environments. A CI system will label commits as failed if at least one check fails. For example, we found an inconsistency in commit **f3719bf** in the project **cloud-custodian/cloud-custodian**. In that instance, GitHub's automatic system labels the commit as failed; however, the build passes when we reproduce it locally. In contrast, commit **43bd11a** passed in the CI checks but failed in the local build. We choose to exclude any of these inconsistent commits from our trustworthiness assessment. Commit **657af5f** in the project **explosion/spaCy** requires flake8>=3.8.0,<3.10.0, while the plugins for our ML product only support flake8 <=3.7.7. Similar incompatibility of the respective dependencies is also a reason for local build failure.

[4]https://cloud.google.com/bigquery/public-data

[5]https://doi.org/10.5281/zenodo.7641214

[6]https://docs.github.com/en/developers/apps/guides/creating-ci-tests-with-the-checks-api

TABLE I
THE PROJECTS' DETAILS AND THEIR TRUSTWORTHINESS PERFORMANCE

| Project name | # Commits | # Files | # Test Cases | Description **Domain** | % of build results that fully align with original build results | |
|---|---|---|---|---|---|---|
| | | | | | PA build | ML build |
| psf/requests | 6,107 | 126 | 554 | HTTP library **Network** | 72% | 0.26% |
| apache/airflow | 15,629 | 14,062 | 9,890 | Workflow management platform **Workflow management** | 50% | 0.03% |
| ansible/ansible | 52,514 | 5,693 | 3,626 | IT automation platform **IT automation** | 52% | 0.84% |
| asciinema/asciinema | 821 | 91 | 26 | Terminal session recorder **Productivity** | 66% | 23.79% |
| numpy/numpy | 30,262 | 1,963 | 23,644 | Scientific computing Python package **Statistics** | 40% | 35.72% |
| bokeh/bokeh | 19,609 | 4,409 | 9,784 | Interactive data visualization tool **Visualization** | 46% | 1.24% |
| dask/dask | 7,280 | 447 | 593 | Binary analysis platform **Task scheduling** | 70% | 23.23% |
| scikit-learn/scikit-learn | 28,198 | 5,338 | 9,753 | Machine learning library **Machine learning** | 44% | 36.02% |
| cloud-custodian/cloud-custodian | 3,786 | 705 | 8,270 | Cloud security and governance tool **Cloud management** | 76% | 0.34% |
| explosion/spaCy | 15,546 | 1,216 | 4,400 | Natural Language Processing tool **NLP** | 46% | 2.22% |

### *D. Data Analysis*

Figure 1 provides an overview of our data analysis procedure. We describe each step below.

**Execute accelerated builds (DA1):** We consider builds that undergo build acceleration to be warm builds. For the PA-product, the dependency graph's construction is completed during the execution of the cold build, i.e., the initial, complete build in which no steps are skipped. During a warm build, the product uses the graph generated during the cold build to locate changed lines of code and their dependencies. By traversing this graph, the product can skip cases that are irrelevant to a code change.

For the ML-product, the source of the training set is the historical test results (i.e., names and locations of test cases, test status and duration of each test case) and code changes (i.e., names and locations of changed files, number of changed lines, commit hashes and authors of changes). In this paper, the complete training set we applied comes from the 50 passing-build commits we selected in DE1, since the training of ML-model requires complete build data. Specifically, we apply build acceleration on N passing-build commits, where we use the previous commits (i.e., from the 1st to $(N-1)$th selected passing-build commit) for training. The acceleration of the first failing-build commit is guided by the untrained machine learning classifier. The classifier uses the previous commits for training starting with the second commit. During a warm build, the product creates a subset of the test suites using different optimization targets.

**Acceleration outcome validation (DA2):** To validate the trustworthiness of our studied build acceleration products, we conduct a comparison of failed builds between accelerated and non-accelerated environments. We therefore separately explore the pytest summary for the original build, the PA accelerated build, and the ML accelerated build.

We analyze the differences between the sources of failures and the number of failed cases for all three outcomes.

We judge the trustworthiness of accelerated builds by comparing the test outputs of the two acceleration products against the original build. To do so, we determine whether the acceleration products' build outcomes are identical to the original build outcomes, and label them as either consistent (i.e., identical) or inconsistent. Through automated scripts and manual spot-checks, we inspect the consistency of two acceleration products within the 50 studied failing commits for each studied project.

## III. STUDY RESULTS

*RQ: How does the trustworthiness of CI acceleration products compare?*

Approach. To compare the trustworthiness of our selected CI acceleration products, we compare the failed build results for 50 builds accelerated by both the PA and ML-products for each subject project. We select 50 failing builds for each project because we seek to determine whether the CI acceleration products will yield the same failing test results as non-accelerated builds. After identifying 50 failing builds for each project (Section 2, DE2), we extract the number of errors and failures according to the test reports generated by the pytest module. To compare the two products, we use Equation (1), where $EF_A$ represents the sum of predicted and observed errors and failures (i.e., true positives) for accelerated builds and $EF_U$ represents the sum of errors and failures for non-accelerated builds, to represent the trustworthiness for each commit.

$$trustworthiness = \frac{EF_A}{EF_U} \quad (1)$$

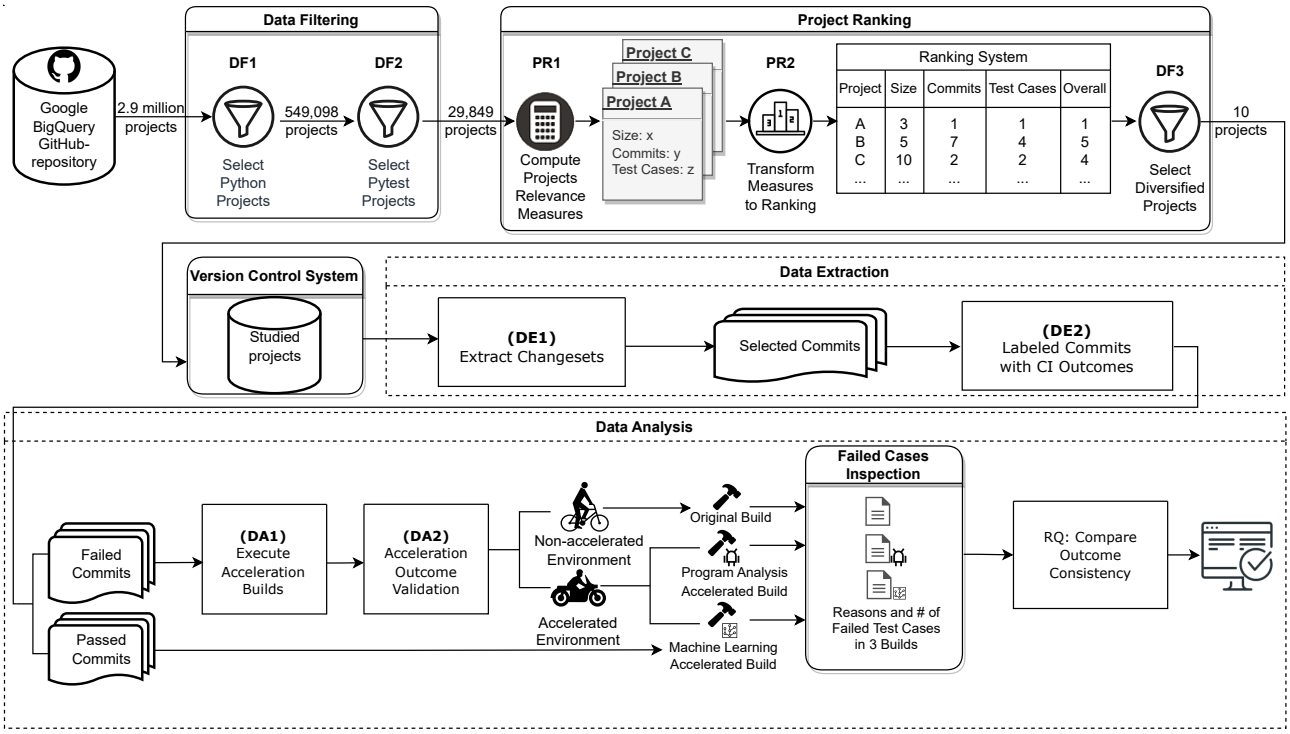


Fig. 1. The overview of study design

Since the studied acceleration products results in fewer tests run by the build, the resulting acceleration cannot identify more errors and failures than the non-accelerated build. Thus, this fraction cannot exceed 1.

To further analyze the optimal parameter setting for the ML-product, we explore how often specific parameter settings offer highly trustworthy results. We do this for the full range of confidence settings (i.e., how high does the model-estimated likelihood need to be before it decides to skip a task) and target settings (i.e., how small of a subset of tasks should the acceleration aim to produce). Thus, we identify for which settings and how often the ML-product's trustworthiness ranks in the top 10 commits in terms of trustworthiness within the 50 commits studied for each project. This exploration aims to uncover the most trustworthy settings for ML acceleration.

Results. Below, we present three observations with respect to the trustworthiness of studied acceleration products.

**Observation 1 – The PA-product more often aligns with non-accelerated build results.** Figure 2 shows that the overall trustworthiness (i.e., mean of multiple trustworthiness from PA or ML builds calculated by Equation (1)) of a PA accelerated build is higher than the trustworthiness of a ML accelerated build. As shown in Table I, a PA accelerated build can fully align with non-accelerated build results for up to 76% of builds (i.e., the number of errors and failures for the accelerated product are equal to the number of errors and failures in the non-accelerated build). We also observe that this alignment fluctuates between 40% and 76% for the PA-product. We believe that this fluctuation is caused by greedy test skipping.

While the ML test acceleration product can sometimes present a trustworthiness that matches or even surpasses the PA-product in specific commits, this is generally not the case. Indeed, the median value for the ML-product in Figure 2 is less than 80% trustworthy. While we only present results for the Requests project, the situation is similar for other projects. The PA-product outperforms the ML-product in terms of overall trustworthiness for all of our studied projects. A complete comparison of the trustworthiness for the two products for all studied projects is available in our online appendix.[5]

**Observation 2 – The trustworthiness of both acceleration products is project dependent.** Neither product presents a unified trustworthiness trend in our studied projects.

For PA-accelerated builds, we observed that the fluctuating ranges for each projects are different. For example, the PA trustworthiness for *Requests* shows a slight fluctuation around 100%. However, the trustworthiness of the product alternates irregularly between 0 and 100% in the *NumPy* and *asciinema* projects. This could be explained by the small number of original errors and failures in these projects. Indeed, the non-accelerated build contains at most two failures in *NumPy* and three failures in *asciinema*. When the PA-product skipped test cases, the accelerated build failed to run any erroneous or failing test cases, leading to a trustworthiness value of 0%.

A small number of original errors and failures also affects the trustworthiness of ML accelerated builds. The distribution of trustworthiness values hovers around 0% trustworthiness in projects with few original errors and failures (e.g. *NumPy* and *asciinema*). Meanwhile, in the projects with more original

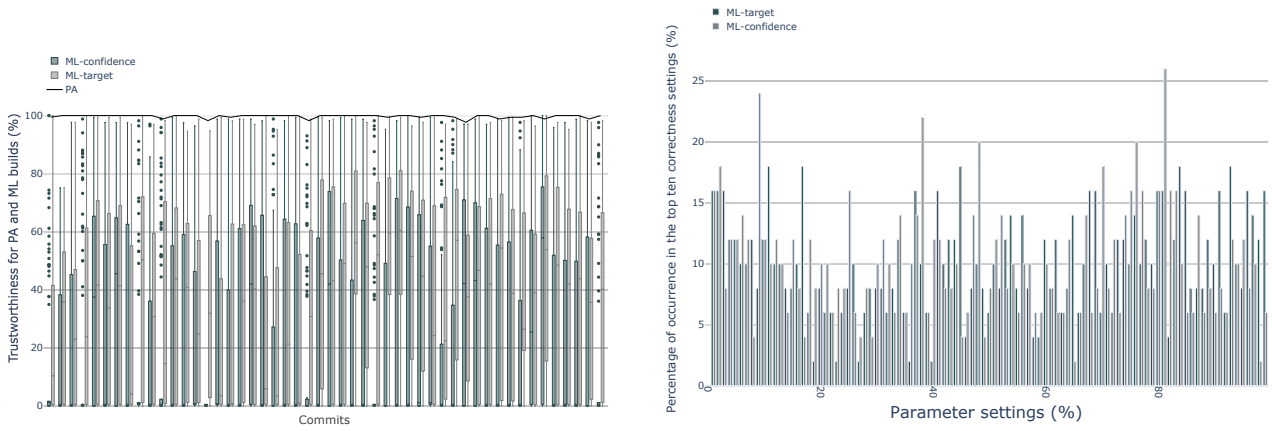


Fig. 2. Trustworthiness of both CI acceleration products and Top 10 trustworthiness occurrence frequency of ML-product in full-range parameter settings for 50 failing Requests builds

errors and failures, the distribution of ML trustworthiness is located in different trustworthiness intervals (e.g., 20% - 60% in *Requests*).

Overall, PA-product performs differently for projects with different test cases and scales. According to the result of the Scott-Knott Effect Size Difference test,[7] the trustworthiness of ten projects can be summarized in six different ranks, which confirms the trustworthiness is project dependent from a statistical perspective.

**Observation 3 – The ML-product does not present any optimal parameter setting for trustworthiness.** To uncover the most trustworthy configuration for ML acceleration within 198 different settings (i.e., 1-99% of target and 1-99% of confidence), we explore 'top 10 trustworthiness occurrence' across ten studied projects. As shown in Figure 2, when accelerating the build of *Requests* project, the parameter setting with the highest trustworthiness (i.e., 81% confidence) is only the best setting 25% of the time. This trend holds for all of our projects. The parameter setting with the highest incidence of trustworthiness is the 67% target setting in the scikit-learn project, and even in that case, the setting was only the best 50% of the time. We provide the figures for all studied projects in our online appendix.[5]

In Figure 2, we also observe that the top five trustworthiness incidences belong to the confidence setting. Thus, the confidence setting generally yields build results that more closely align with non-accelerated builds in the *Requests* project. However, this trend does not hold for the remaining projects. The confidence setting presents the highest trustworthiness in only three of nine studied projects, while the target setting presents highest trustworthiness in five of nine projects. Both settings present the same maximum occurrence of trustworthiness in the *airflow* project. Overall, the highest trustworthiness occurs in the range of 14 to 18 commits out of 50. The indicates that there are no ideal target and confidence settings in ML-accelerated builds. The top ten most frequent trustworthiness settings, both within projects and across projects, are shown in our online appendix.[5]

[7]https://github.com/klainfo/ScottKnottESD

**Outcome**: While the performance of neither product shows a clear trend across our studied projects, the PA-product provides more trustworthy results (4.28–75.66 percentage points). Furthermore, the optimal ML parameter settings vary broadly, with the best (top 10) settings consistent for at most 50% of commits within projects, and at most 36% across projects.

## IV. Conclusion

CI users expect to obtain rapid software development feedback, allowing them to verify if their source code changes integrate cleanly with their existing systems. CI acceleration promises to further accelerate the CI process while maintaining its benefits. However, using CI acceleration is not without costs. To evaluate the practical implications of CI acceleration technology, we conduct an empirical study of 100,000 builds that span 10 projects to compare the trustworthiness of two kinds of commercial-grade CI acceleration techniques. We make three observations from which we conclude the results of builds accelerated by the PA-product more often aligns with non-accelerated build results.

We summarize the following takeaways for practitioners based on the outcomes of this empirical study.

**Balancing Speed and Trustworthiness:** CI acceleration does speed up build but may come at the cost of trustworthiness. Practitioners should carefully consider the trade-off between speed and correctness in their CI processes.

**Practical Value of Program Analysis:** Our study suggests that the PA approach exhibits higher practical value compared to ML for CI acceleration. Practitioners may find greater reliability in PA-based acceleration techniques.

**Improvement Opportunities:** While both acceleration approaches show potential, there is room for improvement to achieve more robust acceleration. Practitioners are encouraged

to explore enhancements in CI acceleration technologies to strike a better balance between speed and reliability.

**User Preferences:** Developers often prioritize a correct build over a fast one. CI acceleration solutions should align with user preferences to ensure broader acceptance and satisfaction within development teams.

Adams [9] showed that developers (at least those within the Linux kernel development context) prefer a correct build to a fast one, so CI acceleration that sacrifices correctness may not be appealing to stakeholders. We therefore encourage future work to explore how the trustworthiness gap can be bridged.

## REFERENCES


[1] Y. Ye and K. Kishida, "Toward an understanding of the motivation of open source software developers," in 25th International Conference on Software Engineering, 2003. Proceedings. IEEE, 2003, pp. 419–429.

[2] X. Jin and F. Servant, "What helped, and what did not? an evaluation of the strategies to improve continuous integration," in 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE). IEEE, 2021, pp. 213–225.

[3] J. Bell, G. Kaiser, E. Melski, and M. Dattatreya, "Efficient dependency detection for safe java test acceleration," in Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, 2015, pp. 770–781.

[4] E. A. Da Roza, J. A. P. Lima, R. C. Silva, and S. R. Vergilio, "Machine learning regression techniques for test case prioritization in continuous integration environment," in 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE, 2022, pp. 196–206.

[5] F. Zampetti, C. Vassallo, S. Panichella, G. Canfora, H. Gall, and M. Di Penta, "An empirical characterization of bad practices in continuous integration," Empirical Software Engineering, vol. 25, no. 2, pp. 1095–1135, 2020.

[6] K. Gallaba and S. McIntosh, "Use and Misuse of Continuous Integration Features: An Empirical Study of Projects that (mis)use Travis CI," IEEE Transactions on Software Engineering, vol. 46, no. 1, p. 33–50, 2020.

[7] K. Gallaba, J. Ewart, Y. Junqueira, and S. McIntosh, "Accelerating Continuous Integration by Caching Environments and Inferring Dependencies," IEEE Transactions on Software Engineering, p. To appear, 2021.

[8] L. Barbosa and A. Hora, "How and why developers migrate python tests," in 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), 2022, pp. 538–548.

[9] B. Adams, "Co-evolution of source code and the build system," in 2009 IEEE International Conference on Software Maintenance, 2009, pp. 461–464.