A Mutation-Guided Assessment of Acceleration Approaches for Continuous Integration: An Empirical Study of YourBase

Zhili Zeng
Software REBELs
University of Waterloo, Canada
z75zeng@uwaterloo.ca

Tao Xiao
Division of Information Science
Nara Institute of Science and Technology, Japan
tao.xiao.ts2@is.naist.jp

Maxime Lamothe
Department of Computer Engineering and Software Engineering
Polytechnique Montréal, Canada
maxime.lamothe@polymtl.ca

Hideaki Hata
Faculty of Engineering
Shinshu University, Japan
hata@shinshu-u.ac.jp

Shane McIntosh
Software REBELs
University of Waterloo, Canada
shane.mcintosh@uwaterloo.ca

ABSTRACT

Continuous Integration (CI) is a popular software development practice that quickly verifies updates to codebases. To cope with the ever-increasing demand for faster software releases, CI acceleration approaches have been proposed; however, adoption of CI acceleration is not without risks. For example, CI acceleration products may mislabel change sets (e.g., a build labeled as failing that passes in an unaccelerated setting or vice versa) or produce results that are inconsistent with an unaccelerated build (e.g., the underlying reasons for failure differ between (un)accelerated builds). These inconsistencies threaten the trustworthiness of CI acceleration products.

In this paper, we propose an approach inspired by mutation testing to systematically evaluate the trustworthiness of CI acceleration. We apply our approach to YourBase, a program analysis-based CI acceleration product, and uncover issues that hinder its trustworthiness. First, we study how often the same build in accelerated and unaccelerated CI settings produce different mutation testing outcomes. We call mutants with different outcomes in the two settings "gap mutants". Next, we study the code locations where gap mutants appear. Finally, we inspect gap mutants to understand why acceleration causes them to survive. Our analysis of ten open-source projects uncovers 2,237 gap mutants. We find that: (1) the gap mutants account for 0.11%–23.50% of the studied mutants; (2) 88.95% of gap mutants can be mapped to specific source code functions and classes using the dependency representation of the studied CI acceleration product; and (3) 69% of gap mutants survive CI acceleration due to deterministic reasons that can be classified into six fault patterns. Our results show that even deterministic CI acceleration solutions suffer from trustworthiness limitations, and highlight the ways in which trustworthiness could be pragmatically improved.

MSR 2024, April 2024, Lisbon, Portugal
© 2024 Association for Computing Machinery.
ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

ACM Reference Format:
Zhili Zeng, Tao Xiao, Maxime Lamothe, Hideaki Hata, and Shane McIntosh. 2024. A Mutation-Guided Assessment of Acceleration Approaches for Continuous Integration: An Empirical Study of YourBase. In *Proceedings of 21st International Conference on Mining Software Repositories (MSR 2024).* ACM, New York, NY, USA, 13 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

The cadence of software development is set by the pace at which Continuous Integration (CI) services process change sets [50, 69]. To enable more rapid development, substantial effort has been invested in improving the performance of each phase of the CI process [78].

Several approaches exist to accelerate the CI process, for example by caching build environments [26], inferring dependencies [7], skipping CI phases [31, 40], skipping CI altogether [1], and accelerating the CI testing phase [73]. These advances have led to the emergence of CI acceleration products for commercial use.[1] CI acceleration reduces build durations 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 steps are unlikely to change based on the modified code.

There are families of approaches to CI acceleration. Program Analysis-based (PA-based) acceleration is a popular deterministic approach that relies on rule-based analysis (conducted prior to the build process) to determine safe ways to accelerate subsequent builds. In this paper, we study a commercial-grade CI acceleration product – YourBase.[2] Similar to other PA-based CI acceleration products, the studied product infers dependencies and constructs a graph [26] during preceding builds, which in turn leveraged to accelerate subsequent builds, e.g., by skipping irrelevant test cases [7, 34].

However, if the CI acceleration process mislabels change sets (e.g., a faulty build passes) [80], then it may ultimately result in more work for developers [28]. Builds can fail for multiple reasons, and these may not be consistent when using CI acceleration. This can allow defects to slip through CI when acceleration is in use. Indeed, when build behaviour is deemed *untrustworthy*, it is not uncommon for developers to rely on sub-optimal workarounds, such

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

as repeated execution [51]. Moreover, since developers tend to prioritize correct build behaviour over efficiency [2], if CI acceleration is untrustworthy, they may hesitate to adopt it.

Despite its importance, systematic approaches to evaluate the trustworthiness of CI acceleration approaches do not yet exist. Early work by Gallaba et al. [26] suggests that at a high level, accelerated outcomes of change sets tend to match, i.e., change sets with a passing (failing) outcome continue to pass (fail) when acceleration is applied; however, the analysis was based on a replay of the builds of 100 historical change sets, which may not capture a full variety or breadth of potential software modifications.

To bridge this gap, we propose an approach inspired by mutation testing [17] to study the trustworthiness of CI acceleration. Mutation testing is a mature program analysis approach that is generally used to evaluate the quality of test suites. Mutants (i.e., perturbed versions of the code under test) are produced, and test suites are re-executed to determine if the mutants are *killed* (i.e., at least one test that passes for the unperturbed version fails) or *survive* (i.e., tests continue to pass on the perturbed version).

In this paper, we assess the trustworthiness of CI acceleration by comparing the outcomes of mutation testing of accelerated and unaccelerated builds of studied change sets. We first measure the percentage of mutants that only survive in the accelerated setting (i.e., gap mutants) to quantify the discrepancy between the accelerated and unaccelerated settings. Then we associate gap mutants with the dependency representation of the studied CI acceleration product to understand the limitations of its decision-making. Finally, we summarize patterns in these limitations by inspecting the gap mutants to identify root causes of discrepancies that undermine the trustworthiness of the studied CI acceleration product. Our analysis of ten open-source projects reveals 2,237 gap mutants, which can be grouped into six patterns. This benchmark allows us to answer the following research questions:

(RQ1) How often do mutants survive due to CI acceleration?
While 60% of the studied mutants survive both accelerated and unaccelerated settings, a gap in mutation outcomes between both settings exists and varies from 0.11%–23.50% across the studied projects.

(RQ2) How are mutants that survive due to CI acceleration mapped to the source code elements?
88.95% of gap mutants can be mapped to specific source code functions and classes using the dependency representation of the studied CI acceleration product; however, 6.66% of gap mutants appear outside of the scope of source code classes, and 4.38% of gap mutants are completely absent from the dependency representation.

(RQ3) What causes mutants to survive in CI acceleration?
A majority (69%) of gap mutants survive in the accelerated setting due to deterministic reasons that can be classified into six fault patterns. However, a considerable proportion (22.5%) of gap mutants survive in CI acceleration because of non-deterministic build behaviour (e.g., mutants that timeout in the unaccelerated setting and survive in the accelerated setting). An analysis of the literature suggests that at least the six deterministic fault patterns can apply to, and help improve, other CI acceleration approaches.

Contributions. This paper makes the following contributions: **(1)** an approach to enhance the verification of CI acceleration approaches using mutation testing; and **(2)** an empirical evaluation of a commercial-grade CI acceleration product – YourBase, that **(a)** uncovers issues that can erode trust and **(b)** summarizes fault patterns within the decision-making process of CI acceleration approaches.

2 RELATED WORK & RESEARCH QUESTIONS

In this paper, we study the trustworthiness of acceleration in Continuous Integration (CI) processes. More specifically, we apply mutation testing to inspect the outcome of a CI acceleration product in order to assess its trustworthiness.

Below, we introduce research related to CI and its acceleration. We then present works related to the rationale behind mutation testing and its applications. Finally, we formulate the research questions that we use to structure our study.

2.1 Continuous Integration and its Acceleration

The CI process begins when a developer (or automated tool) introduces a change in their version control system [46]. A CI build can then be triggered to automatically invoke a series of steps to assess whether the change set integrates safely [22]. When a build is triggered, a build job is created using project-specific criteria (e.g., compile the code then run all tests). To process a build job, a node downloads a specified version of the source code (typically, the latest) and initiates the build process, including acquiring dependencies [48], compiling the code, and running tests [71]. Then, the node sends the build results to a reporting service to broadcast them to the development team. This enables a development feedback loop that empowers its users. It is considered a best practice to keep CI processes short so that feedback is provided quickly [70].

The presence of CI increases the efficiency and quality of software builds. This has made it popular in both proprietary and open-source settings [35, 74]. Since the emergence of CI services, cloud-based CI providers (e.g., TravisCI,[3] CircleCI[4] and GitHub Actions[5]) have bridged the gap between professional CI services and individual users [9, 23, 52, 67]. The adoption of CI has enabled improvements to software quality in various settings [64, 72].

While CI is an improvement over a scheduled build process, if the build process itself is slow, feedback delay can hinder development progress [30]. To reduce this delay, CI acceleration approaches have been proposed. Some approaches rely on using historical code changes [47] to train classifiers [44, 49], which are then used to select tests for execution. Other popular methods for CI acceleration rely on rule-based test case prioritization [12, 68] and test case selection [21]. These techniques accelerate builds by running a subset of the complete test suites or skipping steps of within test cases. In addition, there are techniques that rely on greedy algorithms [73], which decompose the build targets to make the builds faster. Finally, there are techniques that cache environments and infer dependencies [7, 11, 26], which speed up builds by exploiting dependency relations for each build.

[3]https://www.travis-ci.com
[4]https://circleci.com
[5]https://github.com/features/actions

CI acceleration approaches are not without pitfalls. Indeed, the misuse of CI services can cause additional failures [13, 28, 75, 80]. Even if an acceleration approach achieves substantial savings, if it introduces instability with respect to the unaccelerated build, teams will likely hesitate to adopt it. To that end, in this paper, we set out to study the extent to which accelerated builds can be trusted as a replacement for unaccelerated builds.

2.2 Mutation Testing

Mutation testing is a program analysis approach [17] that consists of artificially perturbing source code to inject (likely) faulty behaviour. Mutation testing is primarily used to evaluate the quality of test suites [58, 60, 61]. In practice, mutation testing can be found in both experimental and industrial settings. Indeed, mutation testing has been used at the experimental level to evaluate a model-based approach for security protocols research [15, 16], to guide the input for combinatorial interaction testing [57], and in order to address test flakiness [33]. Meanwhile, at the enterprise level, mutation testing has been applied to optimize Google's deployment workflow [60, 62], and has been integrated into a Facebook production environment [8]. In this paper, we use mutation testing to verify the outcomes of a CI acceleration technique.

As a white box technique, a key feature of mutation testing is its portability [63], which makes it applicable to different programming languages and software testing levels (e.g., unit, integration) [39]. Since Python has become a popular programming language, the application of mutation testing aimed at Python projects has become widespread in recent years [19, 20]. Basic mutations (i.e., statement deletion) has indeed been used to test a CI acceleration in the past [43]. However, in this paper, we conduct mutation tests based on known mutation operators on Python-specific CI acceleration in order to identify CI acceleration weaknesses. Furthermore, the approach tested (Pytest-rts) was not a commercial-grade approach, and only computed the coverage graph once, thus reducing the maximum potential for acceleration. Specifically, we select Mutmut,[6] a mature mutation tool for Python with 12 mutation strategies [25], to generate mutants[7] within our subject systems for the purpose of studying the impact of CI acceleration.

The key metric of mutation testing is the percentage of killed mutants (a.k.a., mutation score) [37, 59], which is calculated by dividing the number of killed mutants (i.e., mutants that cause test cases to fail) by the total number of generated mutants. Mutation analysis has been used to improve various aspects of software engineering [5, 18, 39, 41]; however, to the best of our knowledge, mutation testing has not yet to be applied to assess CI acceleration approaches. In this study, we use the percentage of surviving mutants in accelerated and unaccelerated builds to quantify the gap between accelerated and unaccelerated CI outcomes. To structure this analysis, we formulate the following research question (RQ):

> **RQ1**: How often do mutants survive due to CI acceleration?

[6]https://mutmut.readthedocs.io/en/latest/

[7]A complete list of the mutation operators used can be found in our online appendix

Mutation testing is a code-based analysis method [58] that can complement code coverage analysis [76]. Mutation testing generates mutants in the source code and provides the precise location of each surviving mutant [19]. This feature enables us to associate surviving mutants with features of the programming language. Moreover, it may help to identify limitations in the decision-making process of the CI acceleration approach used by YourBase. Accordingly, we formulate this inquiry through our second research question:

> **RQ2**: How are mutants that survive due to CI acceleration mapped to the source code elements?

Mutation testing can be used to evaluate the quality of test suites [38]. It can therefore allow us to compare the quality of an unaccelerated test suite to the quality of the same test suite when it is the target of CI acceleration. By comparing the results of these two scenarios, we seek to gain insight into the root causes of discrepancies that harm the trustworthiness of our studied CI acceleration product. More precisely, we study why some mutants only survive in accelerated builds through our research question:

> **RQ3**: What causes mutants to survive in CI acceleration?

3 STUDY DESIGN

The goal of our study is to verify the outcomes of deterministic, commercial-grade, CI acceleration and uncover reasons for outcomes that can erode trust. To realize our goal, we apply mutation testing and inspect the mutants that survive in accelerated and unaccelerated CI settings on a commercial solution. To that end, we study the locations in which these mutants appear and reason about why they survive in ten well-established open-source systems.

Multiple commercial-grade CI acceleration approaches exist. However, targeting multiple approaches is both time and cost-prohibitive. We therefore focus on one commercial-grade approach, while keeping our approach as general as possible to allow replication on other PA-based CI acceleration approaches. Recent prior work claims that the approach behind YourBase can provide CI acceleration with minimal resource overhead without compromising build outcome [26]. Since we seek to identify flaws in CI acceleration, an approach that claims build outcome safety provides a prime candidate for our research. We therefore chose YourBase as our chosen commercial-grade CI acceleration product.

While a coverage-guided analysis may seem appropriate, PA-based CI acceleration products, such as YourBase, make acceleration decisions based on a coverage-based source-test mapping. Thus, an evaluation using coverage would be unlikely to reveal faults.

In the remainder of this section, we present how we filter (Section 3.1) and rank (Section 3.2) candidates to obtain our studied projects. We then explain how we collect the mutation results for each studied project and present the approaches that we use to answer our research questions (Section 3.3).

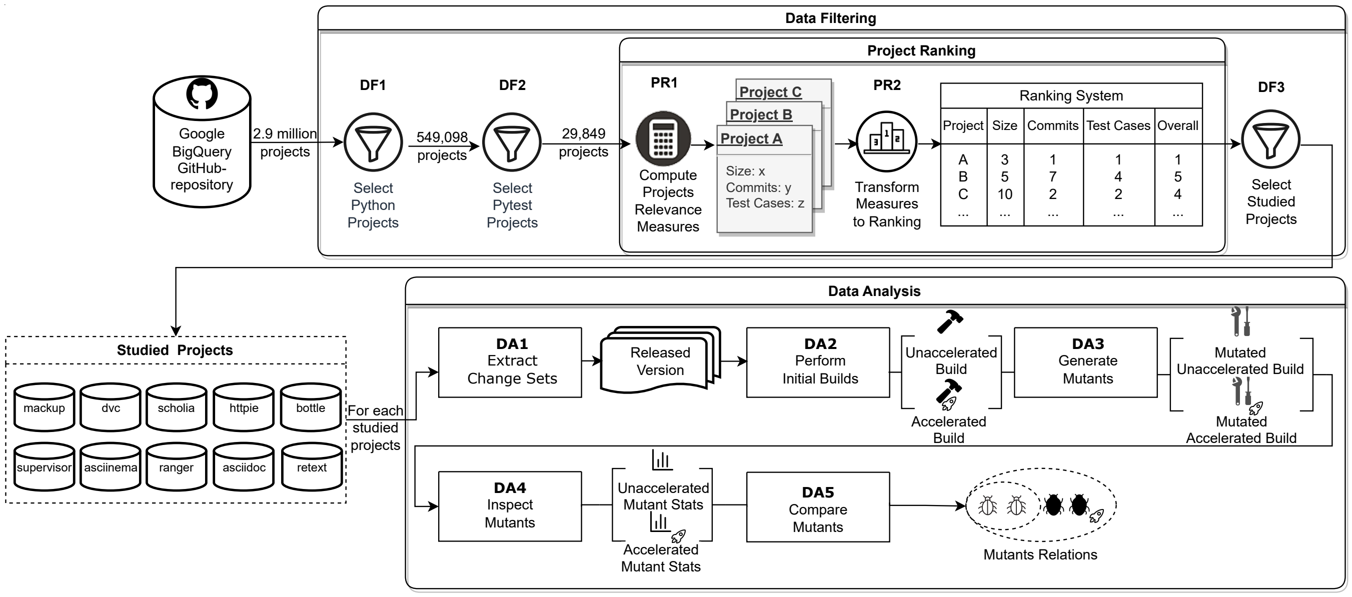

Figure 1: The study design for each selected project

3.1 Data Filtering

As shown in Figure 1, we begin by retrieving a dataset of GitHub repositories from Google BigQuery.[8] This open-source dataset contains the activity (e.g., commits) and property information (e.g., programming language) for the repositories that are hosted on GitHub.

Select Python Projects (DF1): To mitigate the influence of different programming languages on our experimental results, we focus on the builds of projects that are written in a single programming language. To that end, we group candidate projects by programming language and select those that are primarily implemented in Python. We focus on Python because it is the programming language for which the selected CI acceleration product can guarantee the most stable acceleration. We apply the filter by querying for projects that have 'Python' as a field within the 'language' table in the BigQuery dataset. The query returns the projects where 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 select the Python projects that use the pytest framework.[9] We select the pytest framework because previous research demonstrated that it has a more stable performance profile than other testing frameworks [6]. After applying our second filter, 29,849 projects survive.

3.2 Project Ranking

Although our experimental procedures are largely automated, the execution cost for each studied project is large. Thus, repeating the experiment on thousands of systems is untenable. We therefore apply a ranking procedure to systematically select subject systems.

Compute Project Relevance Measures (PR1): For each candidate system, we inspect the number of commits, the number of files, and the number of test cases. We consider the number of commits and files because repositories with a high number of commits and files are more likely to offer an adequate volume of data for validation. Meanwhile, the number of test cases is likely associated with the speed at which a build completes. In this paper, we expect the time consumed by the build of a studied project to be large enough to perform meaningful CI acceleration. Since we cannot build every project to measure its build duration, we instead use the number of test cases as a heuristic.

Transform Measures to Rankings (PR2): To comprehensively quantify the impact of our three chosen measures, we rank each subject system. Figure 1 provides an overview of the ranking procedure. We first compute the rank order of each candidate according to each of the three measures independently. Then, we rank order projects by the sum of its measure-specific ranks.

Select Studied Projects (DF3): Mutation testing is computationally expensive [42, 65]. To maintain a manageable experimental process, we set an upper bound for mutation time when selecting projects. Specifically, we set the maximum allowable mutation duration to one week per project.

To obtain a diverse set of projects from which meaningful conclusions can be drawn, we exclude projects from previously sampled domains, selecting the next highest-ranked project from another domain as a replacement. For example, if the *Supervisor* and the *Nagstamon* projects both belong to the process management domain, we select *Supervisor* because it is the highest-ranked project from the domain in our ranking. We then omit *Nagstamon* and any other projects that belong to the same domain. Finally, we obtain the set of ten studied projects that are summarized in Table 2.

[8]https://cloud.google.com/bigquery/public-data
[9]https://docs.pytest.org/en/7.1.x/contents.html

3.3 Data Analysis

To estimate the trustworthiness of our selected CI acceleration product, we study the quantity, locations, and reasons for the mutants that survive in the accelerated build but do not survive in the unaccelerated build. First, we select a list of change sets for analysis. Then, for each studied change set, we perform an initial unaccelerated build on its preceding commit. This is done to allow the CI acceleration product to generate its internal graph, which is used to reason about build and test invocation steps that may be skipped in the future. Next, we checkout two copies of the studied change set—one for executing unaccelerated (baseline) builds and another for executing accelerated builds. We then apply mutation testing on the independent copies of the codebase, ensuring that the same set of mutants is generated in both settings. Finally, we compare the mutants that we extract from the accelerated and unaccelerated mutation reports. Figure 1 provides an overview of our data analysis procedure. Below, we describe each step in the procedure.

Extract Change Sets (DA1): CI acceleration approaches operate on change sets. In the case of our studied projects, we extract change sets from the version control system. To avoid build failures, which would interfere with our analysis, we select commits tagged with release numbers as our studied change sets. Indeed, release code usually contains fewer failed test cases for a given project because the release tag signals to user that the code has been deemed stable enough for (production) use. The exact details of each studied commit are provided in our online appendix.[10] In addition, the test coverage values for the studied projects are shown in Table 2.

Perform Initial Builds (DA2): We perform initial builds using two settings: (a) unaccelerated build – the complete build of a given project release in which no steps are skipped; (b) accelerated build – the complete build of a given project release in which YourBase skips tests based on its inferred dependency graph. During the unaccelerated build, we invoke the `pytest` command to execute the entire test suite. YourBase constructs and stores a dependency graph for use in future builds. During the accelerated build, we re-invoke the `pytest` command with the studied acceleration plug-in enabled, which will access the dependency graph that was generated during the build of the preceding commit. This graph from the preceding commit replicates the state that YourBase would be in prior to the studied commit. By traversing the graph, the CI acceleration product skips tests that are deemed irrelevant to a code change.

Generate Mutants (DA3): We conduct mutation testing using Mutmut[6] by invoking the `mutmut run` command after the initial accelerated and unaccelerated builds are finished. Mutmut generates mutants according to established rules and provides a mutation report. Mutmut allows us to produce an identical set of mutants that we evaluate in both accelerated and unaccelerated settings. To do so, we first generate mutants in the unaccelerated builds and record the mutation strategies used by Mutmut. We then enforce the same mutation strategies on the same code and commit each produced mutant independently to trigger CI acceleration and test the accelerated setting for each mutant. While we use commits to independently test each mutant, we do so to study the CI acceleration effect on the releases of our studied projects. Indeed, our "commits" do not add new production code, but simply modify existing code through an isolated mutation. Thus, in this case, the concept of commit-relevant mutant [54, 55] does not apply since our mutants are aimed at a release-level evaluation. We record the complete list of mutants by invoking the `mutmut show all` command, storing the output in a mutation report file.

Inspect Mutants (DA4): The generated mutation reports contain detailed information about the mutation process and its outcome, such as the number of killed and surviving mutants, the mutation operator that produced each mutant, and the mutated lines of code. We inspect the mutants that survive in the accelerated and unaccelerated builds of each studied project. The difference between the rates of mutant survival in the two settings quantify the gap between accelerated and unaccelerated CI outcomes.

Compare Mutants (DA5): When comparing the mutants that belong to the accelerated and unaccelerated builds of a studied change set, there are three kinds of potential outcomes: (1) the mutant is killed in both settings, (2) the mutants survives in both settings, and (3) the mutant only survives in the accelerated setting. Outcome 1 and 2 indicate agreement between the two settings, i.e., that the test suite is either capable of detecting the mutant or not, respectively; however, Outcome 3 suggests that the acceleration procedure has erroneously omitted a test that would kill the mutant. Thus, in this paper, we concentrate with Outcome 3, i.e., those that only survive in the accelerated setting.

The principle for this verification is based on the assumption that in the ideal case, a CI acceleration product should only skip test cases that can safely be skipped (i.e., would not affect the result of the build). In this ideal case, the mutation testing outcomes for accelerated and unaccelerated settings should be consistent (i.e., exactly the same). However, if mutation testing were to show that mutants that are killed in the unaccelerated setting survive in the accelerated one, then the trustworthiness of CI acceleration product may be in jeopardy.

4 STUDY RESULTS

In this section, we present the results of our study with respect to our RQs. For each RQ, we present our approach, followed by our observations. We also extend the discussion of our patterns from the CI acceleration product studied in RQ3 to CI acceleration approaches presented in prior works.

(RQ1) How often do mutants survive due to CI acceleration?

RQ1: Approach. To measure the gap between accelerated and unaccelerated CI outcomes, we calculate the percentage of surviving mutants in each studied project. Specifically, we find the number of mutants that survive in accelerated and unaccelerated settings in their respective mutation reports, and use Equation 1 to measure the *Gap Rate*, i.e., the percentage of mutants that only survive during acceleration.

Mutmut reports one of five outcomes for each mutant:

- **Killed:** mutants that have been killed by at least one test;
- **Survived:** mutants that have not been killed by any test;

[10]https://doi.org/10.5281/zenodo.10076515

Table 1: Confusion Matrix for Gap Rate Calculation

	Accelerated	Non-accelerated
Survived	a	b
Killed	c	d
Timeout	e	f
Suspicious	g	h

- **Timeout:** mutants where testing did not complete within a given duration limit;
- **Suspicious:** mutants that increase the test duration substantially, but do not exceed the timeout threshold;
- **Skipped:** mutants that should be omitted from further analysis due to, e.g., introducing syntax errors.

Since Mutmut does not report any skipped mutants in any of our studied projects, we consider the remaining four mutant outcomes in our *Gap Rate* formula (see Table 1 for definitions).

$$Gap\ Rate = \frac{|a \cap d|}{|a \cup c \cup e \cup g|} \tag{1}$$

RQ1: Results. Here, we describe our observations pertaining to the gap between accelerated and unaccelerated CI settings.

Observation 1 – CI acceleration indeed allows mutants that do not survive in the unaccelerated setting to survive in the accelerated setting. The Gap Rate for our studied projects is presented in Table 2. With an average Gap Rate of 7.24% and a standard deviation of 8.17%, the results suggest that the trustworthiness of CI acceleration varies across projects. Indeed, Table 2 illustrates that while the gap between mutants that survive in accelerated and unaccelerated builds can be zero (see the *mackup* project), it can also be as high as 23.5% (see the *retext* project).

In the ideal CI acceleration scenario, the Gap Rate would be zero. Indeed, this is the case for the *mackup* project, where there is no difference between mutants that survive in the accelerated and unaccelerated settings. In this project, mutation testing provides consistent outcomes, indicating that CI acceleration safely skips test cases, and does not affect the results of the build. However, *mackup* is the only project that had a gap rate of zero. The remaining nine studied projects showed varying degrees of inconsistency in the mutation verification outcomes.

The average mutant survival rates across the studied projects are 58.46% and 62.45% in the unaccelerated and accelerated settings, respectively. This high mutant survival rate is important to note because it suggests that the test suites of our studied projects are not particularly effective at detecting mutants. It is also possible that equivalent mutants affect this survival rate. However, while the quality of a test suite may allow a more precise determination of the weak points in a studied project, this does not prevent the use of our methodology. Indeed, it is still possible to determine the gap rate of a lower-quality test suite, and the uncovering of any gap mutants can provide insight into the weaknesses of a CI acceletation approach.

A non-zero Gap Rate indicates that CI acceleration performs untrustworthy test skipping. In the case of the project *retext*, a Gap Rate of 23.5% was observed. This implies that nearly a quarter of the mutants generated for this project survived in the accelerated setting despite being killed by the original test suite. By using the Gap Rate, we can identify mutants that should not survive, which we then inspect more closely to address RQ2 and RQ3 below.

Outcome 1: The overall percentages of mutants that survive in both accelerated and unaccelerated settings of the 10 studied projects are high. Although the quantity varies from project to project, there exist mutants that survive in accelerated settings and were killed in unaccelerated settings in most (9 of the 10) studied projects.

(RQ2) How are mutants that survive due to CI acceleration mapped to the source code elements?

RQ2: Approach. To study the distribution of mutants that survive due to CI acceleration (i.e., gap mutants), we propose four mapping categories. These categories describe the location of each mutant with respect to the source code of the project, and the dependency graph generated by YourBase.

- **Mapping in Function:** The mutated statement appears within the scope of a function and is correctly mapped in the functions of the dependency graph;
- **Mapping in Class:** The mutated statement appears within the scope of a class definition (but outside of the scope of a function) and is correctly mapped in the dependency graph;
- **Not in Class:** The mutated statement appears outside of the scope of source code classes (e.g., in a configuration file), and is untracked in the dependency graph;
- **No Mapping:** The mutated statement appears within the scope of a function, but this mapping is absent in the dependency graph of the CI acceleration product.

The dependency graph generated by YourBase can accurately represent the function-level granularity of a project, which consists of three components: the project's abstract syntax tree, known tests, and their dependencies. To classify the above mapping relations, we first locate each gap mutant in the source code. Then we check if the mapping exists in the dependency graph.

Specifically, if the mutated statement is within the scope of a function, we query for it in the function level of the dependency graph and label the mutant as 'Mapping in Function'. Similarly, if the mutated statement is within the scope of a class but outside of the scope of any function when we query for it in the class level of the dependency graph, and label the mutant as 'Mapping in Class'. Additionally, there are mutants that do not appear within functions or classes. In these cases, YourBase is unable to include them in the dependency graph, and they are labeled as 'Not in Class'. Mutants labeled as 'No Mapping' appear within the scope of a function, but are not tracked in the dependency graph. The 'No Mapping' category is a direct indication of incomplete or missing information in the dependency graph.

This question aims to associate surviving mutants with the dependency graph of YourBase by finding the mapping relation. Through this relation, we aim to identify the limitations of the dependency graph generated by YourBase.

Table 2: Surviving Mutants and Gap Rate of each studied project

Projects	Domain	Coverage	Surviving Mutants (%)		Total	Gap	Gap
			Unaccelerated Settings	Accelerated Settings	Mutants	Mutants	Rate
mackup	Configuration Tool	25%	408 (86.26%)	408 (86.26%)	473	0	0
dvc	Data Version Control	82%	18165 (99.69%)	18185 (99.80%)	18221	20	0.11%
scholia	Organization	89%	4295 (90.14%)	4320 (90.66%)	4765	25	0.58%
httpie	Http Client	91%	3070 (77.19%)	3150 (79.21%)	3977	80	2.54%
bottle	Python Web Framework	2%	62 (2.23%)	64 (2.30%)	2783	2	3.13%
supervisor	Process Management	18%	38 (0.42%)	40 (0.44%)	9153	2	5.00%
asciinema	Productivity	94%	905 (86.44%)	974 (93.03%)	1047	69	7.08%
ranger	File Management	52%	7036 (62.36%)	8037 (71.23%)	11283	1001	12.45%
asciidoc	Documentation	45%	2579 (32.91%)	3147 (40.61%)	7836	568	18.05%
retext	Text Editor	6%	1530 (46.99%)	2000 (61.43%)	3256	470	23.50%

Table 3: Category and Frequency of Gap Mutant Mapping

Mapping Category	Frequency	
	Count	%
Mapping in Function	1902	85.02%
Mapping in Class	88	3.93%
Not in Class	149	6.66%
No Mapping	98	4.38%
• Statement in function	90	4.02%
• Statement in class	8	0.36%

RQ2: Results. Below, we describe our observations pertaining to the mapping of gap mutants in the dependency graph.

Observation 2 – Most of the gap mutants are mapped in the function category of the dependency graph. We extract the 2,237 gap mutants from ten studied projects. As shown in Table 3, nearly 90% of gap mutants could be tracked in the dependency graph. Among them, the majority (95.5%) of gap mutants are in the 'Mapping in Function' category, whereas only 88 gap mutants (4.4%) are in the 'Mapping in Class' category. For the 'Mapping in Class' category, we randomly select five gap mutants (approximately 5%) to verify its validity. We first locate the mutated statement in the source code, and confirm the location is outside the function but inside the class scope. Then we search the class name in the dependency graph and track the class name under the test dependencies.

This distribution is consistent with the encapsulation feature of object-oriented programming, i.e., keeping variables and corresponding methods together in a single unit (class). When we inspect the source code distribution of our ten studied projects, we find that, on average, 11.50% of the code (lines of code) appear outside the scope of classes and functions. This effectively indicates that around 90% of gap mutants should be tracked in the dependency graph. However, this also implies that YourBase is prone to errors despite having access to dependency graph data.

Observation 3 – Incomplete information exists in the dependency graph. According to Table 3, the 'No Mapping' category is rare. Specifically, 4.02% of gap mutants located in the function scope and 0.36% of gap mutants located in the class scope could not be mapped within the dependency graph.

We also study the distribution of the 'No Mapping' mutants across the studied projects. As shown in Table 4, the highest project-specific 'No Mapping' rate is 100%. All mutants from both the *bottle* and *supervisor* projects could not be mapped within the dependency graph. Conversely, all of the gap mutants in the *dvc* and *scholia* projects could be mapped within the dependency graph. For the remaining six projects, the 'No Mapping' rate ranges between 1.45% and 4.70%. We believe that alternative ways to identify test cases in the 'No Mapping' category are needed. Indeed, if the CI acceleration product could be forced to execute these tests, trustworthiness would be improved. The detailed mapping information for each studied project is shown in our online appendix.[7]

Observation 4 – A small number of gap mutants are not mapped within the dependency graph due to their source code locations. There are 6.66% of gap mutants whose mutated statements are neither within the scope of functions nor the scope of classes (e.g., global variable initialization). Since the dependency graph of YourBase only records function and class-level information, the dependencies for these mutants cannot be traced within the dependency graph. These may therefore result in unsafe test skipping, and lower trustworthiness. We consider this to be a limitation of YourBase that should be considered by future products.

Another test acceleration approach [29] bypasses this limitation by using data-flow tracing instead of class or function tracing. This approach records dependencies using a data-flow model without any location constraints for the statements when selecting the point from which the data flow should begin. As a result, this approach is able to track statements outside the scope of classes and functions. However, it also restricts the use of mutation testing for source code inspection. Indeed, many erroneous CI accelerations are caused by failing to adequately analyze source code [13, 27].

Outcome 2: Most of the gap mutants are traceable within the dependency graph. Incomplete dependency data and the locations where mutations are applied explain only a small proportion of the gap mutants that lack mappings.

Table 4: No Mapping Distribution for Studied Projects

Projects	Gap Mutant	No Mapping	
		Count	%
mackup	0	0	0
bottle	2	2	100.00%
supervisor	2	2	100.00%
dvc	20	0	0
scholia	25	0	0
asciinema	69	1	1.45%
httpie	80	3	3.75%
retext	470	18	3.83%
asciidoc	568	25	4.40%
ranger	1001	47	4.70%

(RQ3) What causes mutants to survive in CI acceleration?

RQ3: Approach. To uncover why mutants survive in CI acceleration, and uncover fault patterns that exist in YourBase, we analyze a random sample of 200 mutants. Our random sample was selected to achieve analytic generalizability [24]. Therefore, we stop sampling after we notice saturation [24]. To operationalize saturation, we inspect gap mutants in samples of 50 per round, and continue until we complete a round without discovering new labels. By that definition, saturation was achieved after four rounds (i.e., 200 mutants). The choice of 50 samples per round was chosen to align with similar evaluations from the literature [36, 77]. Before this analysis, we first design a priming procedure to associate the gap mutants with the dependency graph constructed by YourBase. The priming procedure consists of three steps:

(1) Reproduce failure in the unaccelerated build: To verify the integrity of gap mutants (and reduce the impact of flaky tests), for each gap mutant, we first re-apply the suspect mutation to the codebase and re-execute the build without acceleration;

(2) Confirm the test is skipped by CI acceleration: By rerunning the mutated code, we can locate failed tests and verify if the test is skipped in the accelerated setting;

(3) Associate the mutant-exposing test case with the gap mutant: We inspect the test and the source code being tested to determine why the test was skipped.

This priming procedure, specifically the inspection in step (3), is onerous. Indeed, it is impractical to inspect all 2,237 identified gap mutants using our approach. Therefore, to achieve analytical generalization [24], we use the principle of saturation. The inspection is conducted by five individuals with CI acceleration experience. During the inspection, new gap mutants are checked until no new patterns of survival are discovered for at least 50 consecutive gap mutants. Each pattern is discussed by two individuals and confirmed by the rest of the team to resolve disagreements. In the end, using this criterion, we inspect 200 gap mutants, covering all studied projects except *mackup*, since there is no gap between accelerated and unaccelerated settings in that project.

RQ3: Results. Below, we describe our observations concerning the reasons why gap mutants survive the accelerated build. Table 5 outlines the patterns that we observe from the 200 inspected cases. For each pattern, we provide its definition and a mutant example.

Table 5: Reasons and Frequency of Gap Mutant Survival

Reasons	Frequency	
Dependencies untracked by the CI acceleration product	138	69.00%
• Class properties	40	20.00%
• Global variable	31	15.50%
• Constructor	23	11.50%
• Static method decorator	17	8.50%
• Conditional statement	14	7.00%
• Configuration-purpose string	13	6.50%
Non-deterministic build behaviours	45	22.50%
• Inconsistent Labeling	24	12.00%
• Flaky Tests	21	10.50%
Other (Failed to classify)	17	8.50%

Observation 5 – 69% of the inspected gap mutants that survive in the accelerated build had dependencies that were untracked by YourBase. Below, we identify six patterns among the inspected gap mutants in this category.

Class properties: Missing class-related dependencies (20% of Gap Mutants) in the dependency graph would cause the CI acceleration product to incorrectly label test cases that depend on this code as independent. Therefore, the CI acceleration product would erroneously label these test cases as safe to skip. We uncovered this issue during our analysis when checking the dependencies of class properties, where we found class fields with null values in the CI acceleration product's dependency graph. In those cases, if a skipped test case depends on the variable initialization in the class property, it can lead to a mutant surviving mutation testing when it should not, implying a lack of trustworthiness for the CI acceleration product.

```
'''./supervisor/options.py
@@ -69,7 +69,7 @@'''
    pass

class Options:
-    stderr = sys.stderr
+    stderr = None
     stdout = sys.stdout
     exit = sys.exit
     warnings = warnings
```

Pattern 1: The gap mutant survived due to class properties

Global variable: 15.5% of the inspected gap mutants survive because of a lack of, or incomplete dependencies due to, global variables. In these cases, YourBase fails to correctly record the dependency relations of the global variables in the source code. Changes to global variables can affect the code under test, and skipping tests that depend on those variables can lead to erroneous results. In our experiments, this can occur when global variables are mutated and tests are erroneously skipped. If a CI acceleration cannot accurately record global variable dependencies, it may skip tests that are necessary to ensure the trustworthiness of the accelerated build.

```
'''./supervisor/options.py
@@ -60,7 +60,7 @@'''
```

```
    version_txt = os.path.join(mydir, 'version.txt')
    with open(version_txt, 'r') as f:
        return f.read().strip()
−VERSION = _read_version_txt()
+VERSION = None

 def normalize_path(v):
    return os.path.normpath(os.path.abspath(os.path.expanduser(v)))
```

Pattern 2: The gap mutant survived due to incorrect handling of global variables

Constructor: 11.5% of the inspected gap mutants survived because the CI acceleration product did not detect dependencies between a test case and a constructor. These failures occur when mutated statements are within the scope of a constructor. Thus, the dependency graph fails to record the dependency information for the constructor, and their related test cases are erroneously skipped.

```
'''asciinema/pty_.py
@@ −153,7 +153,7 @@'''
 class SignalFD:
    def __init__(self, signals: List[signal.Signals]) −> None:
−       self.signals = signals
+       self.signals = None
        self.orig_handlers: List[Tuple[signal.Signals, Any]] = []
        self.orig_wakeup_fd: Optional[int] = None
```

Pattern 3: The gap mutant survived due to constructor

Static method decorator: 8.5% of inspected gap mutants had code statements mutated in a function annotated with *@staticmethod*, a Python decorator that defines a static method within a class. Static methods belong to a class rather than its objects and can be called without creating a class instance. This can cause the static method dependencies to be inaccurately captured by CI acceleration.

```
'''./asciidoc/asciidoc.py
@@ −1693,7 +1693,6 @@'''
        if not skipsubs:
            Title.attributes['title'] = Title.dosubs(Title.attributes['title'])

−   @staticmethod
    def dosubs(title):
```

Pattern 4: The gap mutant survived due to static method decorator

Conditional statement: 7% of inspected gap mutants survive due to complex conditional statements in the source code. These mutated statements typically involve multiple conditions that are combined using logical operators such as 'and', 'or', and 'not'. Mutmut generates mutants in these logical operators and we use these mutations to discover incomplete information for the functions that contain these complex conditional statements in the dependency graph. We find that the dependencies for the complex conditional statements are not captured accurately by the CI acceleration product, as a result, the mutants survive during acceleration.

```
'''./ReText/tab.py
@@ −185,7 +185,7 @@'''
        errMsg = errMsg.replace('<a href="%s">', '').replace('</a>', '')
       return '<p style="color: red">%s</p>' % errMsg
    headers = ''
−   if includeStyleSheet and self.p.ss is not None:
+   if includeStyleSheet and self.p.ss is  None:
      headers += '<style type="text/css">\n' + self.p.ss + '</style>\n'
      elif includeStyleSheet:
```

Pattern 5: The gap mutant survived due to conditional statement

Configuration-purpose string: 6.5% of inspected gap mutants have mutated statements (e.g., version and date strings) belonging to configuration files. These strings are often separate from the body of functional code; however, configurations can also affect the behaviour of the program and the tests that need to be executed. YourBase does not track changes in the strings of configuration files, so it skips tests linked to the changes made in the configuration file, further leading to the survival of gap mutants.

```
'''httpie/__init__.py
@@ −4,7 +4,7 @@'''
 __version__ = '3.2.1'
−__date__ = '2022−05−06'
+__date__ = 'XX2022−05−06XX'
 __author__ = 'Jakub Roztocil'
 __licence__ = 'BSD'
```

Pattern 6: The gap mutant survived due to configuration-purpose string

Observation 6 – 22.5% of the inspected gap mutants survive due to non-deterministic build behaviour originating from test flakiness and inconsistent mutation outcomes. We identify two patterns among the inspected gap mutants in this category. We define and characterize each pattern below.

Flaky Tests: 10.5% of the inspected gap mutants survive due to flaky tests. These tests produce non-deterministic outcomes when executing on the same program version [3]. Flakiness is often caused by test code that has external dependencies or relies on non-deterministic algorithms [32]. These flaky tests can cause mutants to survive (i.e., the build passes) for a mutant that should have been killed (i.e., the build should have failed). Flakiness can therefore cause mutants to be incorrectly labeled as gap mutants. Therefore, whenever we manually identify a flaky test, we remove it from our analysis.

Inconsistent Labeling: Inconsistent labeling from our selected mutation tool also results in non-deterministic build behaviour between accelerated and unaccelerated settings. When Mutmut is invoked, some mutants are classified as 'suspicious', i.e., test execution time increased tenfold or more, and 'timeout', i.e., their execution time exceeds an upper limit on duration. These additional outcomes may occur in the unaccelerated setting, and may survive in the accelerated setting. This causes such mutants to be labeled as gap mutants by our approach. These gap mutants, caused by non-deterministic build behaviour, cannot be used to reason about the trustworthiness of CI acceleration. Indeed, it is possible to reduce the impact of such issues by repeating the experiment, but there is no guarantee that such experimental noise can be fully eliminated.

In addition, 17 (8.5%) of the inspected mutants could not be classified using our taxonomy. These mutants survive because of project-specific situations. For example, in the *httpie* project, an http request failed because of a mutated URL. In other cases, we lack the domain knowledge to diagnose the issue. We attempt to keep this "Other" category as small as possible by using an iterative procedure when analyzing the mutants. Specifically, whenever a new category appears, we repeatedly re-check whether any of the "Other" mutants belong to the new category.

Outcome 3: Most (69%) mutants survive CI acceleration due to deterministic reasons that can be generally classified into six fault patterns. A smaller, but not insignificant proportion (22.5%) of mutants survive CI acceleration because of non-deterministic build behaviour (e.g., flaky tests).

Potential Generalizability of the Observations

A systematic mapping study [66] found that prior research shows a preference for test selection and prioritization approaches that use failure history, execution history, and test coverage. Approaches that use failure [79, 81] and execution history [45, 53] are at the mercy of prior failures. Rare failures are therefore potential pitfalls for these approaches. The causes for gap mutant survival might also occur in rarely modified code. Using mutation-based testing to analyze the trustworthiness of these approaches might therefore allow the simulation of rarer failure cases, and in turn, may improve the trustworthiness of these approaches.

Approaches that are based on test coverage [14] attempt to maximize the number of faults that can be detected without exceeding a budget of resources (e.g., time, lines). Therefore, by design, such approaches cannot guarantee that faults will be missed due to the prioritization decisions that they make. Indeed, while additional issues might also exist, we suspect that the reasons for the survival of gap mutant that we report in this paper will also impact such approaches if the mutant-detecting tests are deemed inefficient. Therefore, while our studied CI acceleration product makes specific use of a program analysis approach for test selection, we suspect that our methodology, as well as the issues that we report, can help to improve other types of test selection approaches.

A systematic literature review of test case prioritization [56] based on machine learning found that these approaches tend to concentrate on a small number of features that are relatively simple to compute. Indeed, many approaches concentrate on code complexity metrics [56]. We suspect that these approaches would also likely suffer from low trustworthiness when issues occur in class properties, global variables, and constructors, since those areas are unlikely to have low code complexity (e.g., constructors often only initialize fields), yet might cause test errors (e.g., if fields are incorrectly initialized). Approaches based on textual data, such as the approach by Aman et al. [4], prioritize test cases by targeting those that have different textual contents and, therefore, are more likely to test different aspects of the program under analysis. Similarly, to traditional approaches based on code coverage, such approaches might also present low trustworthiness for issues that are exposed by less favoured tests. Finally, we suspect that ML-based test selection and prioritization approaches that concentrate on code coverage are likely to suffer from similar issues as those presented in this paper. Indeed, even state-of-the-art approaches [10] make use of class-level dependency analysis, which we suspect would also be prone to configuration-purpose issues. Therefore, while we study a PA-based CI acceleration approach, our methodology, and the issues that we detect, have the potential to also guide improvements to ML-based test selection and prioritization approaches. Nevertheless, future work is necessary to empirically verify that our findings can indeed apply to other CI acceleration tools.

5 THREATS TO VALIDITY

Below, we describe the threats to the validity of our empirical study.

5.1 Construct Validity

Gap mutants may not accurately indicate the degree of trustworthiness of CI acceleration if the mutants themselves are not realistic. For example, we apply the default setting for the selected mutation tool without customization. The default mutation strategy was not originally designed for assessing the trustworthiness of CI acceleration. Within the pre-established strategy, we obtain a finite number of mutants and consequently extract limited gap mutants. For example, the selected mutation tool (i.e., Mutmut) only generates 408 mutants for the mackup codebase due to the small project size and limited test cases. An analysis of the impact of different configuration settings may prove fruitful; however, we believe that our results demonstrate the promise of our mutation-based approach to trustworthiness assessment.

5.2 Internal Validity

Our capacity to discover fault patterns is based on the inspection of gap mutants. If the mutation tool fails to generate mutants that relate to the changed code or generates equivalent mutants, we cannot detect trustworthiness problems associated with this part of the code. This, in turn, prevents us from detecting the complete set of fault patterns in the CI acceleration decision-making. However, in principle, the selected mutation tool (i.e., Mutmut) applies mutations to a broad range of statements.

Since our study is conducted only using YourBase, it is possible for the obtained gap mutants to be due to bugs or limitations in the tool itself. We attempt to mitigate this issue by inspecting a sample of 200 mutants that only survive in the accelerated setting and catalogue their causes. However, it is possible that the causes are indeed bound to YourBase's implementation. Future replication of our work in other CI acceleration solutions would be helpful.

It is also possible that the variance in the test coverage of our studied projects may (partially) explain our results (rather than the acceleration setting itself). The test coverage varies broadly from 2% to 94% among our studied projects. However, we believe that the test coverage variance of our studied projects is reflective of the breadth of real-world use-cases. Nevertheless, follow-up studies should control for this confounding factor.

5.3 External Validity

We set an upper bound of mutation time to select projects that can be analyzed within our practical constraints. This constrains the scale of the studied projects. Moreover, we study GitHub-hosted open-source projects that use Python and pytest. Because GitHub is one of the most popular hosts for open-source projects and Python is a popular programming languages, our findings apply to a wide range of projects. Similarly, pytest is one of the most popular unit testing frameworks for Python. Our focus on a single CI acceleration product may constrain the generalizabity of our study; however, the studied product implements a typical PA-based technique and the generalizabity of the observations has been comprehensively discussed. Nonetheless, we encourage extensions of our study using other approaches, projects, languages, and test frameworks.

6 CONCLUSIONS

Users of CI expect to obtain rapid and reliable 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 trustworthiness. However, the trustworthiness of CI acceleration is not guaranteed.

To evaluate the trustworthiness of a commercial-grade CI acceleration product, we apply mutation testing within accelerated and unaccelerated settings across ten projects to measure the gap between accelerated and unaccelerated builds. We make six observations (see Section 4) from which we conclude that:

- 90% of studied projects contain mutants that survive in accelerated settings despite being killed in unaccelerated settings.
- Up to 23.5% of mutants survive only in the accelerated setting (7.24% on average).
- 88.95% of the mutants that only survive in the accelerated setting are traceable within the dependency graph of the studied CI acceleration product.
- 4.38% of gap mutants are not traceable due to incomplete information in the dependency graph.
- While 22.5% of mutants only survive in the accelerated setting because of non-deterministic build behaviour, 69% survive because of deterministic reasons, and can be broadly classified into six categories.

In conclusion, this study shows that while CI acceleration may sacrifice trustworthiness, it is possible to use techniques such as mutation testing to evaluate this sacrifice and identify improvements for current approaches. For future work, our study demonstrates that the following improvements for PA-based CI acceleration approaches would improve their trustworthiness: (1) depending on the size and complexity of the codebase, it may be necessary to manually refine the dependency graph, especially by concentrating on class properties, global variables, and constructor components; and (2) solutions should be added to detect and bypass flaky test during CI acceleration to minimize the impact of flakiness.

7 DATA AVAILABILITY

The online appendix is available, under an open license, using the following link: https://doi.org/10.5281/zenodo.10076515

REFERENCES

[1] Rabe Abdalkareem, Suhaib Mujahid, and Emad Shihab. 2020. A machine learning approach to improve the detection of ci skip commits. *IEEE Transactions on Software Engineering* (2020).

[2] Bram Adams. 2009. Co-evolution of source code and the build system. In *2009 IEEE International Conference on Software Maintenance*. IEEE, 461–464.

[3] Amal Akli, Guillaume Haben, Sarra Habchi, Mike Papadakis, and Yves Le Traon. 2022. Predicting Flaky Tests Categories using Few-Shot Learning. *arXiv preprint arXiv:2208.14799* (2022).

[4] Hirohisa Aman, Sousuke Amasaki, Tomoyuki Yokogawa, and Minoru Kawahara. 2020. A comparative study of vectorization-based static test case prioritization methods. In *2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE, 80–88.

[5] Richard Baker and Ibrahim Habli. 2012. An empirical evaluation of mutation testing for improving the test quality of safety-critical software. *IEEE Transactions on Software Engineering* 39, 6 (2012), 787–805.

[6] Lívia Barbosa and Andre Hora. 2022. How and Why Developers Migrate Python Tests. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 538–548. https://doi.org/10.1109/SANER53432.2022.00071

[7] Jonathan Bell, Gail Kaiser, Eric Melski, and Mohan Dattatreya. 2015. Efficient dependency detection for safe Java test acceleration. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 770–781.

[8] Moritz Beller, Chu-Pan Wong, Johannes Bader, Andrew Scott, Mateusz Machalica, Satish Chandra, and Erik Meijer. 2021. What it would take to use mutation testing in industry—a study at facebook. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 268–277.

[9] Jean-Marcel Belmont. 2018. *Hands-On Continuous Integration and Delivery: Build and release quality software at scale with Jenkins, Travis CI, and CircleCI.* Packt Publishing Ltd.

[10] Antonia Bertolino, Antonio Guerriero, Breno Miranda, Roberto Pietrantuono, and Stefano Russo. 2020. Learning-to-Rank vs Ranking-to-Learn: Strategies for Regression Testing in Continuous Integration. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/3377811.3380369

[11] Qi Cao, Ruiyin Wen, and Shane McIntosh. 2017. Forecasting the Duration of Incremental Build Jobs. In *Proc. of the International Conference on Software Maintenance and Evolution (ICSME)*. 524–528.

[12] Cagatay Catal and Deepti Mishra. 2013. Test case prioritization: a systematic mapping study. *Software Quality Journal* 21, 3 (2013), 445–478.

[13] An Ran Chen, Tse-Hsun Peter Chen, and Shaowei Wang. 2023. T-Evos: A Large-Scale Longitudinal Study on CI Test Execution and Failure. *IEEE Transactions on Software Engineering* 49, 4 (2023), 2352–2365. https://doi.org/10.1109/TSE.2022.3218264

[14] Junjie Chen, Yiling Lou, Lingming Zhang, Jianyi Zhou, Xiaoleng Wang, Dan Hao, and Lu Zhang. 2018. Optimizing Test Prioritization via Test Distribution Analysis. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 656–667. https://doi.org/10.1145/3236024.3236053

[15] Frédéric Dadeau, Pierre-Cyrille Héam, and Rafik Kheddam. 2011. Mutation-based test generation from security protocols in HLPSL. In *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*. IEEE, 240–248.

[16] Frédéric Dadeau, Pierre-Cyrille Héam, Rafik Kheddam, Ghazi Maatoug, and Michael Rusinowitch. 2015. Model-based mutation testing from security protocols in HLPSL. *Software Testing, Verification and Reliability* 25, 5-7 (2015), 684–711.

[17] Richard A DeMillo, Richard J Lipton, and Frederick G Sayward. 1978. Hints on test data selection: Help for the practicing programmer. *Computer* 11, 4 (1978), 34–41.

[18] Lin Deng, Nariman Mirzaei, Paul Ammann, and Jeff Offutt. 2015. Towards mutation analysis of android apps. In *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 1–10.

[19] Anna Derezińska and Konrad Hałas. 2014. Analysis of mutation operators for the python language. In *Proceedings of the Ninth International Conference on Dependability and Complex Systems DepCoS-RELCOMEX. June 30–July 4, 2014, Brunów, Poland*. Springer, 155–164.

[20] Anna Derezinska and Konrad Halas. 2014. Experimental evaluation of mutation testing approaches to python programs. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops*. IEEE, 156–164.

[21] Daniel Di Nardo, Nadia Alshahwan, Lionel Briand, and Yvan Labiche. 2015. Coverage-based regression test case selection, minimization and prioritization: A case study on an industrial system. *Software Testing, Verification and Reliability* 25, 4 (2015), 371–396.

[22] Stefan Dösinger, Richard Mordinyi, and Stefan Biffl. 2012. Communicating continuous integration servers for increasing effectiveness of automated testing. In *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 374–377.

[23] Thomas Durieux, Rui Abreu, Martin Monperrus, Tegawendé F Bissyandé, and Luís Cruz. 2019. An analysis of 35+ million jobs of Travis CI. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 291–295.

[24] Kathleen M Eisenhardt. 1989. Building theories from case study research. *Academy of management review* 14 (1989), 532–550.

[25] Matheus Ferreira, Lincoln Costa, and Francisco Carlos Souza. 2020. Search-based Test Data Generation for Mutation Testing: a tool for Python programs. In *Anais da IV Escola Regional de Engenharia de Software*. SBC, 116–125.

[26] Keheliya Gallaba, John Ewart, Yves Junqueira, and Shane McIntosh. 2021. Accelerating Continuous Integration by Caching Environments and Inferring Dependencies. *IEEE Transactions on Software Engineering* (2021), To appear.

[27] Keheliya Gallaba, Maxime Lamothe, and Shane McIntosh. 2022. Lessons from Eight Years of Operational Data from a Continuous Integration Service: An Exploratory Case Study of CircleCI. In *Proc. of the International Conference on Software Engineering (ICSE)*. 1330–1342.

[28] Keheliya Gallaba and Shane McIntosh. 2020. Use and Misuse of Continuous Integration Features: An Empirical Study of Projects that (mis)use Travis CI. *IEEE Transactions on Software Engineering* 46, 1 (2020), 33–50.

[29] Alessio Gambi, Jonathan Bell, and Andreas Zeller. 2018. Practical test dependency detection. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 1–11.

[30] Taher Ahmed Ghaleb, Daniel Alencar Da Costa, and Ying Zou. 2019. An empirical study of the long duration of continuous integration builds. *Empirical Software Engineering* 24 (2019), 2102–2139.

[31] Milos Gligoric, Rupak Majumdar, Rohan Sharma, Lamyaa Eloussi, and Darko Marinov. 2014. Regression test selection for distributed software histories. In *International Conference on Computer Aided Verification*. Springer, 293–309.

[32] Martin Gruber, Stephan Lukasczyk, Florian Kroiß, and Gordon Fraser. 2021. An empirical study of flaky tests in python. In *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 148–158.

[33] Sarra Habchi, Maxime Cordy, Mike Papadakis, and Yves Le Traon. 2021. On the use of mutation in injecting test order-dependency. *arXiv preprint arXiv:2104.07441* (2021).

[34] Mary Jean Harrold. 2000. Testing: a roadmap. In *Proceedings of the Conference on the Future of Software Engineering*. 61–72.

[35] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. 2016. Usage, costs, and benefits of continuous integration in open-source projects. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 426–437.

[36] Toshiki Hirao, Shane McIntosh, Akinori Ihara, and Kenichi Matsumoto. 2019. The review linkage graph for code review analytics: a recovery approach and empirical study. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 578–589.

[37] Dominik Holling, Sebastian Banescu, Marco Probst, Ana Petrovska, and Alexander Pretschner. 2016. Nequivack: Assessing mutation score confidence. In *2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 152–161.

[38] Mostafa Jangali, Yiming Tang, Niclas Alexandersson, Philipp Leitner, Jinqiu Yang, and Weiyi Shang. 2022. Automated generation and evaluation of JMH microbenchmark suites from unit tests. *IEEE Transactions on Software Engineering* (2022).

[39] Yue Jia and Mark Harman. 2010. An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering* 37, 5 (2010), 649–678.

[40] Xianhao Jin and Francisco Servant. 2022. Which builds are really safe to skip? Maximizing failure observation for build selection in continuous integration. *Journal of Systems and Software* 188 (2022), 111292.

[41] René Just, Darioush Jalali, Laura Inozemtseva, Michael D Ernst, Reid Holmes, and Gordon Fraser. 2014. Are mutants a valid substitute for real faults in software testing?. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 654–665.

[42] Samuel J Kaufman, Ryan Featherman, Justin Alvin, Bob Kurtz, Paul Ammann, and René Just. 2022. Prioritizing mutants to guide mutation testing. In *Proceedings of the 44th International Conference on Software Engineering*. 1743–1754.

[43] Eero Kauhanen, Jukka K. Nurminen, Tommi Mikkonen, and Matvei Pashkovskiy. 2021. Regression Test Selection Tool for Python in Continuous Integration Process. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 618–621. https://doi.org/10.1109/SANER50967.2021.00077

[44] Remo Lachmann, Sandro Schulze, Manuel Nieke, Christoph Seidl, and Ina Schaefer. 2016. System-level test case prioritization using machine learning. In *2016 15th IEEE International Conference on Machine Learning and Applications (ICMLA)*. IEEE, 361–368.

[45] Jingjing Liang, Sebastian Elbaum, and Gregg Rothermel. 2018. Redefining Prioritization: Continuous Prioritization for Continuous Integration. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 688–698. https://doi.org/10.1145/3180155.3180213

[46] Christian Macho, Shane McIntosh, and Martin Pinzger. 2016. Predicting Build Co-Changes with Source Code Change and Commit Categories. In *Proc. of the International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. 541–551.

[47] Christian Macho, Shane McIntosh, and Martin Pinzger. 2017. Extracting build changes with builddiff. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 368–378.

[48] Christian Macho, Shane McIntosh, and Martin Pinzger. 2018. Automatically repairing dependency-related build breakage. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 106–117.

[49] Shane McIntosh, Bram Adams, Meiyappan Nagappan, and Ahmed E Hassan. 2014. Mining co-change information to understand when build changes are necessary. In *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 241–250.

[50] Ade Miller. 2008. A hundred days of continuous integration. In *Agile 2008 conference*. IEEE, 289–293.

[51] Peter Miller. 1998. Recursive make considered harmful. *AUUGN Journal of AUUG Inc* 19, 1 (1998), 14–25.

[52] Ioannis K Moutsatsos, Imtiaz Hossain, Claudia Agarinis, Fred Harbinski, Yann Abraham, Luc Dobler, Xian Zhang, Christopher J Wilson, Jeremy L Jenkins, Nicholas Holway, et al. 2017. Jenkins-CI, an open-source continuous integration system, as a scientific data and image-processing platform. *SLAS DISCOVERY: Advancing Life Sciences R&D* 22, 3 (2017), 238–249.

[53] Armin Najafi, Weiyi Shang, and Peter C. Rigby. 2019. Improving Test Effectiveness Using Test Executions History: An Industrial Experience Report. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. 213–222. https://doi.org/10.1109/ICSE-SEIP.2019.00031

[54] Milos Ojdanic, Ezekiel Soremekun, Renzo Degiovanni, Mike Papadakis, and Yves Le Traon. 2023. Mutation Testing in Evolving Systems: Studying the Relevance of Mutants to Code Evolution. *ACM Trans. Softw. Eng. Methodol.* 32, 1, Article 14 (feb 2023), 39 pages.

[55] Milos Ojdanic, Ezekiel Soremekun, Renzo Degiovanni, Mike Papadakis, and Yves Le Traon. 2023. Mutation Testing in Evolving Systems: Studying the Relevance of Mutants to Code Evolution. *ACM Trans. Softw. Eng. Methodol.* 32, 1, Article 14 (feb 2023), 39 pages.

[56] Rongqi Pan, Mojtaba Bagherzadeh, Taher A. Ghaleb, and Lionel Briand. 2021. Test case selection and prioritization using machine learning: a systematic literature review. *Empirical Software Engineering* 27, 2 (14 Dec 2021), 29. https://doi.org/10.1007/s10664-021-10066-6

[57] Mike Papadakis, Christopher Henard, and Yves Le Traon. 2014. Sampling program inputs with mutation analysis: Going beyond combinatorial interaction testing. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*. IEEE, 1–10.

[58] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. 2019. Mutation testing advances: an analysis and survey. In *Advances in Computers*. Vol. 112. Elsevier, 275–378.

[59] Mike Papadakis, Donghwan Shin, Shin Yoo, and Doo-Hwan Bae. 2018. Are mutation scores correlated with real fault detection? a large scale empirical study on the relationship between mutants and real faults. In *Proceedings of the 40th International Conference on Software Engineering*. 537–548.

[60] Goran Petrović and Marko Ivanković. 2018. State of mutation testing at google. In *Proceedings of the 40th international conference on software engineering: Software engineering in practice*. 163–171.

[61] Goran Petrović, Marko Ivanković, Gordon Fraser, and René Just. 2021. Does mutation testing improve testing practices?. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 910–921.

[62] Goran Petrović, Marko Ivanković, Gordon Fraser, and René Just. 2021. Practical mutation testing at scale: A view from Google. *IEEE Transactions on Software Engineering* 48, 10 (2021), 3900–3912.

[63] Goran Petrovic, Marko Ivankovic, Bob Kurtz, Paul Ammann, and René Just. 2018. An industrial application of mutation testing: Lessons, challenges, and research directions. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 47–53.

[64] Gustavo Pinto, Fernando Castor, Rodrigo Bonifacio, and Marcel Rebouças. 2018. Work practices and challenges in continuous integration: A survey with Travis CI users. *Software: Practice and Experience* 48, 12 (2018), 2223–2236.

[65] Alessandro Viola Pizzoleto, Fabiano Cutigi Ferrari, Jeff Offutt, Leo Fernandes, and Márcio Ribeiro. 2019. A systematic literature review of techniques and metrics to reduce the cost of mutation testing. *Journal of Systems and Software* 157 (2019), 110388.

[66] Jackson A. Prado Lima and Silvia R. Vergilio. 2020. Test Case Prioritization in Continuous Integration environments: A systematic mapping study. *Information and Software Technology* 121 (2020), 106268. https://doi.org/10.1016/j.infsof.2020.106268

[67] Chuck Rossi, Elisa Shibley, Shi Su, Kent Beck, Tony Savor, and Michael Stumm. 2016. Continuous deployment of mobile software at facebook (showcase). In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 12–23.

[68] Gregg Rothermel, Roland H Untch, Chengyun Chu, and Mary Jean Harrold. 1999. Test case prioritization: An empirical study. In *Proceedings IEEE International Conference on Software Maintenance-1999 (ICSM'99).'Software Maintenance for Business Change'(Cat. No. 99CB36360)*. IEEE, 179–188.

[69] Hyunmin Seo, Caitlin Sadowski, Sebastian Elbaum, Edward Aftandilian, and Robert Bowdidge. 2014. Programmers' build errors: a case study (at google). In *Proceedings of the 36th International Conference on Software Engineering*. 724–734.

[70] Daniel Ståhl and Jan Bosch. 2013. Experienced benefits of continuous integration in industry software product development: A case study. In *The 12th IASTED International Conference on Software Engineering*. 736–743.

[71] Daniel Ståhl and Jan Bosch. 2014. Modeling continuous integration practice differences in industry software development. *Journal of Systems and Software* 87 (2014), 48–59.

[72] Daniel Ståhl and Jan Bosch. 2016. Industry application of continuous integration modeling: a multiple-case study. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 270–279.

[73] Mohsen Vakilian, Raluca Sauciuc, J David Morgenthaler, and Vahab Mirrokni. 2015. Automated decomposition of build targets. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 123–133.

[74] 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 2015 10th joint meeting on foundations of software engineering*. 805–816.

[75] Carmine Vassallo, Sebastian Proksch, Harald C Gall, and Massimiliano Di Penta. 2019. Automated reporting of anti-patterns and decay in continuous integration. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 105–115.

[76] Moshi Wei, Yuchao Huang, Jinqiu Yang, Junjie Wang, and Song Wang. 2022. Cocofuzzing: Testing neural code models with coverage-guided fuzzing. *IEEE Transactions on Reliability* (2022).

[77] Tao Xiao, Dong Wang, Shane McIntosh, Hideaki Hata, Raula Gaikovina Kula, Takashi Ishio, and Kenichi Matsumoto. 2021. Characterizing and mitigating self-admitted technical debt in build systems. *IEEE Transactions on Software Engineering* 48, 10 (2021), 4214–4228.

[78] Yunwen Ye and Kouichi Kishida. 2003. Toward an understanding of the motivation of open source software developers. In *25th International Conference on Software Engineering, 2003. Proceedings*. IEEE, 419–429.

[79] Zhe Yu, Fahmid Fahid, Tim Menzies, Gregg Rothermel, Kyle Patrick, and Snehit Cherian. 2019. TERMINATOR: Better Automated UI Test Case Prioritization. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3338906.3340448

[80] Fiorella Zampetti, Carmine Vassallo, Sebastiano Panichella, Gerardo Canfora, Harald Gall, and Massimiliano Di Penta. 2020. An empirical characterization of bad practices in continuous integration. *Empirical Software Engineering* 25, 2 (2020), 1095–1135.

[81] Yuecai Zhu, Emad Shihab, and Peter C. Rigby. 2018. Test Re-Prioritization in Continuous Testing Environments. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 69–79. https://doi.org/10.1109/ICSME.2018.00016