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 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) gap mutants account for 0.11%–23.50% of the studied mutants; (2) 88.95% of projects uncover 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.

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. 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. 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
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, CircleCI and GitHub Actions) 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.

1[https://www.travis-ci.com](https://www.travis-ci.com)
2[https://circleci.com](https://circleci.com)
3[https://github.com/features/actions](https://github.com/features/actions)
CI acceleration approaches are not without pitfalls. Indeed, the misuse of CI services can cause additional failures [15, 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, a mature mutation tool for Python with 12 mutation strategies [25], to generate mutants 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?

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).
3.1 Data Filtering

As shown in Figure 1, we begin by retrieving a dataset of GitHub repositories from Google BigQuery.\textsuperscript{4} 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.\textsuperscript{5} We select the pytest framework because previous research demonstrated that it has a more stable performance profile than other testing frameworks \textsuperscript{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 \textsuperscript{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.
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.\[^{16}\] 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;

\[^{16}\]https://doi.org/10.5281/zenodo.10076515
Table 1: Confusion Matrix for Gap Rate Calculation

<table>
<thead>
<tr>
<th></th>
<th>Accelerated</th>
<th>Non-accelerated</th>
</tr>
</thead>
<tbody>
<tr>
<td>Survived</td>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>Killed</td>
<td>c</td>
<td>d</td>
</tr>
<tr>
<td>Timeout</td>
<td>e</td>
<td>f</td>
</tr>
<tr>
<td>Suspicious</td>
<td>g</td>
<td>h</td>
</tr>
</tbody>
</table>

• **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).

\[
\text{Gap Rate} = \frac{|a \cap d|}{|a \cup c \cup e \cup g|}
\]

(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.
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.

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].

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 (95.3%) 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, 0.02% 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.

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.
(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.

**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.

**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.
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.

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 `@classmethod`, 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.

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.

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.

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 predict 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 language, 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 generalizability of our study; however,
the studied product implements a typical PA-based technique and the generalizability 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


