Developer-Applied Accelerations in Continuous Integration

A Detection Approach and Catalog of Patterns

Mingyang Yin
Software REBELs
University of Waterloo
Canada
mingyang.yin@uwaterloo.ca

Yutaro Kashiwa
Nara Institute of Science and Technology
Japan
yutaro.kashiwa@is.naist.jp

Keheliya Gallaba
Centre for Software Excellence, Huawei
Canada
keheliya.gallaba@huawei.com

Mahmoud Alfadel
Department of Computer Science
University of Calgary
Canada
mahmoud.alfadel@ucalgary.ca

Yasutaka Kamei
POSL Lab
Kyushu University
Japan
kamei@ait.kyushu-u.ac.jp

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

ABSTRACT

Continuous Integration (CI) provides a feedback loop for the change sets that developers produce. It is crucial that CI processes change sets quickly to provide timely feedback to developers and enable teams to release software updates rapidly. Prior work has made several advances in proposing automated approaches to speed up CI builds. While these approaches have been broadly adopted, CI platforms are flexible enough to enable teams to produce custom strategies to optimize or omit unnecessary or redundant tasks (i.e., developer-applied accelerations). Exploring developer-applied accelerations and identifying recurrent patterns within them may enable broader reuse and can inform recommendations to enhance software development efficiency.

In this paper, we set out to detect and catalog developer-applied CI accelerations. First, we propose clustering, rule-based, and ensemble approaches to detect developer-applied accelerations in a dataset of 2,896 CircleCI build jobs, which achieve an F1-score of up to 0.64. We then conduct a qualitative analysis of the detected developer-applied accelerations to create a detailed catalog of 14 patterns spanning four categories of purposes, 16 patterns spanning five categories of mechanisms, and three categories of magnitudes, from which we infer actionable implications for both the consumers and the providers of CI platforms. Developers can leverage our identified patterns to audit their CI pipelines for inefficiencies, such as redundant invocations of costly external services and rebuilds triggered by minor corrections. Additionally, developers can use our identified patterns to create templates that detect non-impactful changes to specific files, such as `.yml` and `.json`.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ASE '24, October 27-November 1, 2024, Sacramento, CA, USA
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-1248-7/24/10
https://doi.org/10.1145/3691620.3695533

KEYWORDS

Continuous Integration, Build Systems, Empirical Software Engineering

ACM Reference Format:
Mingyang Yin, Yutaro Kashiwa, Keheliya Gallaba, Mahmoud Alfadel, Yasutaka Kamei, and Shane McIntosh. 2024. Developer-Applied Accelerations in Continuous Integration: A Detection Approach and Catalog of Patterns. In *39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24), October 27-November 1, 2024, Sacramento, CA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3691620.3695533

1 INTRODUCTION

Continuous Integration (CI) is a software development practice involving frequent integration of code changes into a shared repository [6]. Its main goal is to provide developers with timely feedback about whether the change sets that they produce integrate cleanly with the existing codebase and the change sets that other team members have produced concurrently [8]. Prior work shows that the adoption of CI is associated with improvements in developer productivity [12, 29] and software quality [14, 30, 32].

Using a CI process that is suboptimally configured can delay feedback and waste computational resources [10, 33], which can frustrate developers and increase the effective cost of operating the service. Indeed, Widder et al. [36] found that developers often complained about slow feedback caused by long CI processes.

To reduce time-to-feedback and computation costs, CI acceleration approaches have been proposed (e.g., [2, 3, 17, 20, 25, 31]). They reduce time-to-feedback and computation costs by skipping jobs, phases, or steps that are deemed unnecessary or unlikely to provide value based on the change set or its characteristics. Since a failure signal from CI is richer (providing log data that can aid in diagnosis) and of greater importance (typically calling for the immediate attention of team members), acceleration techniques focus on skipping jobs, phases, and steps that are likely to pass.

While off-the-shelf CI acceleration approaches exist, popular CI platforms like CircleCI are flexible enough to allow teams to implement their own. Indeed, it is common for CI platforms to allow consumers (e.g., developers) to specify the behaviour of their CI process using YAML configuration files that denote script commands to be invoked in a shell. As such, developers may leverage the full

```
test-go-race-remote-docker:
  ...
  steps:
  - run:
      command: |
        # If the branch being tested starts with ui/ or docs/ we want to exit the job without failing
        [[ "$CIRCLE_BRANCH" = ui/* || "$CIRCLE_BRANCH" = docs/* ]] && {
          # stop the job from this step
          circleci-agent step halt
        }
        # exit with success either way
        exit 0
      name: Check branch name
      working_directory: ~/
```

Listing 1: A real-world example of a developer-applied acceleration from the `hashicorp/vault` project.

flexibility of the shell to conditionally invoke/skip commands or terminate the CI process early when conditions are or are not met. Such acceleration practices with flexibility in controlling the workflow based on contextual conditions or requirements are what we refer to as *developer-applied accelerations*. For instance, Listing 1 provides an example of such a developer-applied acceleration in the CircleCI specification of the `hashicorp/vault` project, which checks the branch on which the change set has landed, skipping steps of the process if the branch is focused on the development of the user interface or product documentation.

Studying developer-applied accelerations can yield insights for enhancing the efficiency and effectiveness of CI practices. For example, CI platforms, such as CircleCI can consider promoting recurrent accelerations to built-in acceleration features. Moreover, a catalog of recurrent developer-applied acceleration patterns can serve as a useful reference to promote reuse among CI consumers. Therefore, in this paper, we set out to detect and catalog developer-applied accelerations. Specifically, we perform an empirical study of developer-applied accelerations in a corpus of 2,896 CircleCI jobs. To structure our study, we formulate and address the following two research questions:

***RQ*$_1$: To what extent can developer-applied accelerations be detected automatically?**

Motivation: Developer-applied accelerations may include clever solutions to recurring problems. Developer-applied accelerations, unlike platform-provided accelerations, are rarely documented and are hard for developers from other projects to discover. Therefore, to foster reuse, we set out to automatically discover developer-applied accelerations.

Results: We propose a clustering-based solution, a rule-based solution, and an ensemble solution that combines the two. When applied to a manually curated set of 2,896 jobs, our ensemble approach achieves the best performance, yielding a maximum F1-score of 0.64 with a recall of 0.67 and a precision of 0.62 when the tunable acceleration threshold is optimized.

***RQ*$_2$: Are there recurring patterns of developer-applied accelerations?**

Motivation: To further foster reuse, we set out to identify acceleration patterns within the set of accelerated examples. By inspecting a large sample of developer-applied accelerations, we set out to catalog recurrent patterns, which may inform recommendations about when accelerations are likely to be beneficial.

Results: Our catalog documents the purpose, mechanism, and magnitude of each pattern. We discover 14 patterns spanning four categories in purposes, 16 patterns spanning five categories in mechanisms, and three categories in magnitudes of developer-applied accelerations. For example, steps are often skipped because the change set being built includes or excludes specific files or based on the availability of external packages (or lack thereof) and tools in the build environment (*purpose*). Environment variable settings, Git commands, and even external APIs are used to make acceleration decisions (*mechanism*). While it is typical for all subsequent job steps to be skipped, step or sub-step skipping behaviour is not uncommon (*magnitude*).

Contributions. In summary, this paper makes the following contributions: **(1)** *empirical evidence* of the phenomenon of developer-applied CI accelerations; **(2)** an *approach to detecting* developer-applied acceleration; and **(3)** a *catalog of emergent patterns* that summarizes the characteristics of developer-applied accelerations.
Data Availability. To aid in future replication of our results, we make a replication package publicly available.[1]

2 CORE CONCEPTS IN CIRCLECI

In this paper, we choose to focus on the community of users who adopt CircleCI as their CI platform. According to GitHub Marketplace,[2] CircleCI has the largest number of installations on GitHub. In this section, we describe the basic concepts of the CircleCI platform and how its users configure their CI processes.

CircleCI uses YAML[3] as their **CI specification** format. CI specifications contain a series of **workflows**, which each have triggers (i.e., when to run the workflow), jobs to run, and other configuration data (e.g., arguments passed to the jobs). Each workflow may define the dependencies between jobs, so that order-sensitive jobs can be appropriately orchestrated by the provider.

A build job, or simply **job**, consists of multiple **steps**. Each step could invoke a command from an orb,[4] execute a script, or perform any other shell operation. All steps within a job run consecutively in the same environment. Hence, any data (e.g., files) produced during one step are automatically available in the subsequent steps. When a job runs, data such as the start time and the logs of each step are stored. Each run of a job is called a build invocation, or simply an **invocation**, and all of the data produced by an invocation is called a build **record**.

[1] https://github.com/software-rebels/Developer-Applied-Accelerations-in-Continuous-Integration
[2] https://github.com/marketplace?category=continuous-integration&query=sort%3Apopularity-desc
[3] https://yaml.org/
[4] https://circleci.com/orbs/

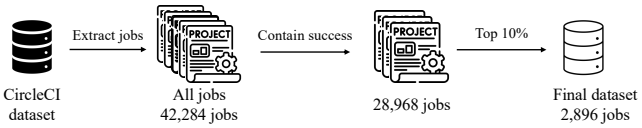


Figure 1: An overview of data collection and data filtering.

3 STUDY DESIGN

In this section, we present our approaches to data collection (Section 3.1) and filtering (Section 3.2).

3.1 Data Collection

To address our research questions, we require a rich and diverse set of real-world CI specifications. As CI has been widely adopted as a practice, there are many CI services available. While GitHub Actions may have a larger installation base on GitHub, it is used to perform any sort of routine action, and is not limited to the CI use case. Thus, considering the popularity of CircleCI, and its explicit focus on CI workflows, we choose to focus our analysis on it.

We begin with the dataset that was collected by Gallaba et al. [9]. They retrieve the dataset by crawling historical build records using the CircleCI API.[5] The primary data collection process involved retrieving all GitHub repositories, locating CircleCI configurations (.circleci/config.yml files), and querying the CircleCI API to obtain build records. After this process was finished, a dataset of 42,284 build jobs and 21,597,023 build records was created. The stored build records occurred during the period from June 2017 to January 2022, spanning 7,795 repositories. The dataset we used comprises 7,795 projects, written in several languages (e.g., Python, JavaScript, C++).

3.2 Data Filtering

The goal of the Gallaba et al. [9] study was to analyze the CI provider's perspective. Therefore, the dataset that was collected aimed to be as complete as possible. Our study aims to focus on the perspective of the invested CI consumer. As such, we apply filters to the Gallaba et al. dataset to identify meaningful and well-tested accelerations that are ready for reuse. Figure 1 shows the overview of our data filtering process. Below, we explain each step.

3.2.1 Unsuccessful invocations. Since build invocations may terminate prior to completion (e.g., failed or canceled invocations), build records will not always be complete. These prematurely terminated build records will not accurately reflect the duration of a complete build. Hence, we first filter out all build records that do not have a result setting of success. 16,970,889 records spanning 28,968 build jobs survive this filter. In the remainder of the paper, we refer to the build records that survive the filter simply as records.

3.2.2 Jobs with too few records. A job with few records may not have established a clear trend in execution time. Moreover, it is difficult to reason about the impact of an acceleration on such jobs with low cardinality. Figure 2 plots the cardinality of jobs in the records that survive our prior filter. The figure shows a clearly skewed distribution with a long tail.

[5]https://circleci.com/docs/api/v2/index.html

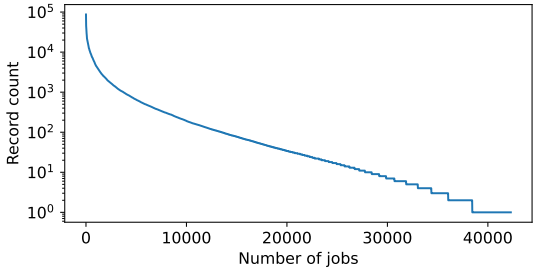


Figure 2: The long-tail distribution of build jobs. This figure shows that most build jobs only have a small number of build invocations.

To exclude the jobs with low cardinality, we select for future analysis only those jobs in the top 10% in terms of build record count.[6] The filter reduces our dataset to 2,896 jobs with at least 1,391 invocations. Finally, 13,292,279 records (78%) survive our filter.

4 RQ_1: DEVELOPER-APPLIED ACCELERATION DETECTION

In this section, we present our approach to detecting developer-applied acceleration and the results of an empirical evaluation. We propose a clustering-based solution, a rule-based solution, and an ensemble solution that combines the two (Sections 4.1, 4.2, and 4.3), and present the results of the evaluation that compares the accuracy of the proposed approaches (Section 4.4).

4.1 Clustering-Based Solution

The key intuition behind this solution is that when a build job is accelerated, build durations will naturally cluster around accelerated and non-accelerated central points, respectively. When the acceleration is substantial, the two groups should be distinct.

The clustering-based solution divides build records into two groups according to build duration. Then, a ratio is computed between the sum of the two within-group standard deviations and the distance between the centers of these groups. When the ratio is small, it suggests that the records truly belong to two distinct groups in terms of build duration, and that an acceleration is likely being employed. To group records by their duration, we apply k-means—an unsupervised approach that splits examples into k mutually exclusive clusters.

4.1.1 Pre-processing Prior to Clustering. Before applying k-means, we need to control for the tendency of build duration to grow over time (i.e., data drift), and the impact of outliers.

(1) **Data drift.** The duration of a build record tends to grow as projects age [22]. As a result, accelerated records that occur later in a project's history may spend as much time as (or even more time than) the non-accelerated records that occur earlier in the project's history. Figure 3 shows an example of this phenomenon in the `spotify/scio` project. To control for this confounding factor, we calculate the clusters individually in monthly periods. Some months

[6]A sensitivity analysis shows that the F1-score drops from 0.73 to 0.64 if we increase the top 5% to 10%. The F1-score is presented in later sections.

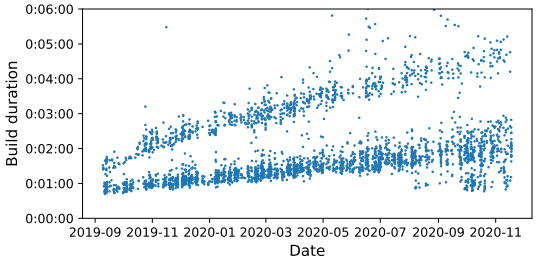


Figure 3: Evolution of build duration. Note that later accelerated invocations can take more time than earlier non-accelerated invocations.

have abnormal ratio values because these months only have too few records to produce stable k-means clusters. To reduce the impact of extreme magnitudes, we compute the median ratios weighted by the record count for each month.

(2) **Outliers.** In extreme cases, the duration can appear to be misleadingly long (e.g., due to transient network issues) [4, 19]. This creates outliers that impact the performance of the k-means approach. For example, k-means may be misled to treat outliers as one group and all other invocations as another. Through experimentation, we find that excluding the top 5% of records substantially improves the performance of the k-means approach. We also find outliers that have a far shorter duration than other invocations, but they are much fewer in number than slow outliers. Thus, only excluding the fastest 1% of invocations is sufficient to counteract that noise.[7]

4.1.2 Computing Clusters. Figure 4 shows an overview of the clustering-based solution that we apply. We apply k-means with $k = 2$ to attempt to split build durations into accelerated and non-accelerated clusters. The only feature that we provide to the k-means algorithm is the build duration, because it is the most distinct feature between accelerated and non-accelerated invocations. To determine whether the two clusters are distinct, we use the ratio between the standard deviations of build duration in each group and the distance between the two cluster centers. We use `sklearn.cluster.KMeans`, a widely used machine learning library for Python [24], to implement our approach.

More precisely, let A and B denote the collection of slower and faster invocations divided by k-means. σ_A denotes the standard deviation of build duration in A, and so for B. C_A denotes the cluster center of A, which is build duration because that is the only feature that we use, and so for C_B. Then, we define inter-cluster ratio as:

$$Ratio = \frac{\sigma_A + \sigma_B}{|C_A - C_B|}$$

This ratio intuitively measures the proximity of two clusters relative to their standard deviations. When two distinct groups indicate acceleration, they typically yield a low ratio. To detect whether acceleration is present or not, we need to apply a threshold t to the ratio measurement. The build job is considered accelerated if and only if $Ratio < t$. Arbitrary selection of t is unlikely to produce useful results. Thus, we perform a preliminary analysis to select t empirically (see Section 4.4 for details).

Like any learning-based solution, our clustering-based solution will produce false positives. In our setting, false positives are likely because the approach cannot distinguish between the developer-applied accelerations (which are desired) and platform-provided accelerations, such as built-in caching (which are not desired). Nonetheless, our goal is not to develop an approach with perfect discriminatory power, but rather to aid in detecting acceleration candidates among the 2,896 jobs, and to facilitate our further manual inspection to build a catalog of developer-applied accelerations.

4.2 Rule-Based Solution

The key intuition behind this solution is that developer-applied accelerations can often be identified by their use of certain commands or keywords within the CI specification file. In fact, CircleCI provides developers with a suite of commands to craft custom acceleration scripts. For instance, developers might use the `circleci halt` command to skip unnecessary steps in the CI workflow. By locating specifications that include such commands or keywords, we can identify jobs that are likely to be accelerated by the developer.

Parsing built-in service provider commands within CI specifications presents challenges due to the YAML-based script structure. YAML includes features like anchors and aliases, which enable developers to reuse configuration logic blocks by expanding stored statements at dereference locations within the CI specification. To handle this complexity, our approach performs a dynamic analysis that leverages the unfolded CI specification (still in the same YAML format but with all anchors and aliases dereferenced) that is stored in the CircleCI logs.

Listing 2 shows an example of an unfolded CircleCI specification. To extract the executed commands within a job, we traverse the list of values associated with the `command` key (line 10). This produces a list of strings, which may each span multiple lines. For each entry in the list, we analyze each line to determine whether any of them start with the keyword `circleci`. When such lines are detected, we label the job as having been accelerated.

4.3 Ensemble Solution

The key intuition behind this solution is that the clustering-based and rule-based solutions can complement each other by capturing different types of developer-applied accelerations. For example, the clustering-based solution can detect accelerations that do not use CircleCI commands, such as Bash scripts, while the rule-based solution can detect accelerations that use CircleCI commands. By combining both solutions, we may improve the completeness of our detection of developer-applied accelerations.

Our ensemble approach classifies a CI job as accelerated if it is flagged as such by the rule-based approach, regardless of the finding produced by the clustering-based approach. In cases where the rule-based solution indicates that the job is not accelerated, the ensemble approach relies on the ratio generated by the clustering-based approach. If $Ratio < t$, the ensemble approach labels the job as accelerated.

[7] Note that excluding the outliers improves the F1-score from 0.44 to 0.64.

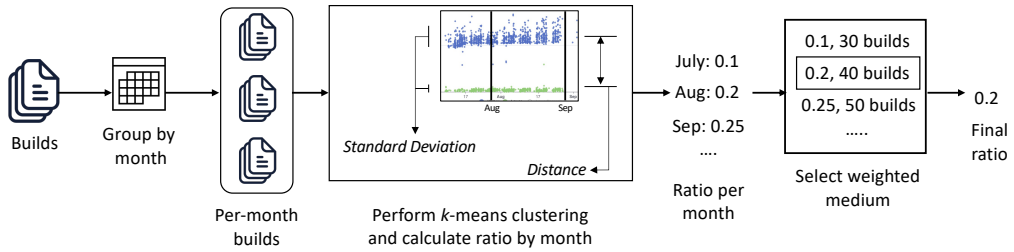


Figure 4: An overview of the *k*-means approach and ratio measurement.

```
# Orb 'circleci/shellcheck@1.3.16' resolved to 'circleci/shellcheck@1.3.16'
version: 2
jobs:
  docker-pre-publish:
    ...
    steps:
    ...
    - run:
        name: should pre build docker images (targeting a release branch)?
        command: |
          eval `.circleci/get_pr_info.sh -b`
          if  [[ ! "$TARGET_BRANCH" =~ "^release-[0-9|.]+$" ]] && [[ ! "$TARGET_BRANCH" =~ "^test-[0-9|.]+$" ]] ; then
            echo Targeting branch $TARGET_BRANCH will not publish docker images.
            circleci step halt
          fi
    ...
```

Listing 2: Example of unfolded YAML file from the `diem/diem` project.

```
1 if git log --name-status --exit-code --format=oneline "$GIT_BASE_REVISION..$CIRCLE_SHA1" -- .circleci/config.yml home package.json yarn.lock; then
2   echo "No changes found in watched paths.  Ending job."
3   circleci-agent step halt
4 else
```

Listing 3: An example of developer-applied acceleration.

4.4 Evaluation

We conduct an evaluation to compare the performance of our three approaches in identifying jobs with developer-applied accelerations. The evaluation requires a ground truth to which the solutions can be compared. Hence, we first manually label a sample of jobs to determine if they contain developer-applied accelerations (Section 4.4.1). Then, we evaluate our solutions (Sections 4.4.2 and 4.4.3).

4.4.1 Ground Truth Establishment. We begin with our initial dataset of 2,896 jobs (Section 3.2). Since manual inspection of all 2,896 jobs is too onerous, we select a random sample of 339 build jobs for inspection. The size of this sample ensures that we have 95% confidence that the measures we compute will generalize to the population of 2,896 jobs within a ± 5% margin of error.

For each sampled job, we first plot its build duration over time (see Figure 3 for an example) for visual indications of multimodality. We also inspect the CI specification file of the project to check if the job contains further evidence of developer-applied acceleration.

Given the potential for frequent changes to the CI specification, we retrieve the most up-to-date version of the CI specification file by identifying the branch where the majority of CI build invocations take place. To achieve this, we mine the build records of a job and retrieve the most recent 1,000 invocations based on their dates of invocation. If there are fewer than 1,000 invocations available, we consider all of them. From these 1,000 invocations, we analyze the branch on which each invocation occurred and compute the number of invocations associated with each branch. The branch with the highest count of invocations is selected. We then extract and inspect the CI specification corresponding to the latest invocation on the most active branch.

Listing 3 shows an example of a job that we label as accelerated.[8] The script in this listing examines file changes (as indicated in line 1) and, based on the outcome, may skip subsequent steps by invoking `circleci-agent step halt` (line 3). This behaviour leads us to label the job as accelerated. Note that there are accelerated jobs that do not have clear indications of acceleration in the specification. To counteract this, we conduct an in-depth manual analysis of the scripts that are linked to the job specification. For instance, consider the `integration_tests_artifacts` job in the `gatsbyjs/gatsby` repository. The CI specification in the footnote[9] indicates that the job consists of only one step (line 271). However, this step invokes a developer-applied command.[10] An inspection of the command definition (line 149) reveals that this command invokes external scripts (line 171). Upon reviewing the external scripts, we discover an example with indications of acceleration (lines 29–34).[11]

Finally, to mitigate potential bias in our manual inspection, two authors independently label the build jobs. In cases of disagreement (where one author labels a job as "accelerated" and the other as "non-accelerated"), a third author casts the deciding vote. Upon resolving all disagreements and excluding five jobs because their

[8] https://github.com/expo/expo/blob/009ab81e2f27b6b79a1a18c895acb0d40c921bfa/.circleci/config.yml#L71
[9] https://github.com/gatsbyjs/gatsby/blob/14d3be3fd4009a7f24ea9ed9c92b26ae5e8e4940/.circleci/config.yml
[10] https://circleci.com/docs/configuration-reference/#commands
[11] https://github.com/gatsbyjs/gatsby/blob/14d3be3fd4009a7f24ea9ed9c92b26ae5e8e4940/scripts/assert-changed-files.sh

original repositories are inaccessible, we label 24 (7.1%) jobs as accelerated with developer-applied accelerations and 310 (91.4%) as non-accelerated.[12]

4.4.2 Performance Measurement. Upon establishing the ground truth, we evaluate our detection approach based on the established ground truth and compute the accuracy.

Let J denote all sampled 339 jobs, g_j denote the ground truth of a job j. $g_j = 1$ when the job is accelerated; $g_j = 0$ otherwise. Similarly, d_j denotes the detection result of a job j. $d_j = 1$ when the job is detected by our approach as accelerated; $d_j = 0$ otherwise. We perform our evaluation using precision, recall, and F1-score metrics, which are calculated as follows:

$$Precision = \frac{|\{j \in J : g_j = 1 \text{ and } d_j = 1\}|}{|\{j \in J : d_j = 1\}|}$$

$$Recall = \frac{|\{j \in J : g_j = 1 \text{ and } d_j = 1\}|}{|\{j \in J : g_j = 1\}|}$$

$$F1\text{-}score = 2 \times \frac{Precision \times Recall}{Precision + Recall}$$

Additionally, we calculate the F1-score of Zero-R —a naïve classification approach that labels all jobs as accelerated— to serve as a baseline.

4.4.3 Results. In Figure 5, we present the performance metrics for our approaches. The x-axis shows the settings of the threshold t that is used within the clustering-based approach. Note that the rule-based solution is equivalent to the ensemble solution with $t = 0$.

Starting with a threshold of 0, the recall of the ensemble approach is already at 0.33 due to the rule-based solution that can alone detect 33% of the truly accelerated jobs. As we increase the threshold, recall steadily rises until it reaches 1. Conversely, precision begins at 0.53 and reaches its peak of 0.63 at $t = 0.19$. These results highlight the tradeoff between more liberal thresholds that improve recall at the cost of precision.

The F1-score, which balances precision and recall, reaches its highest point of 0.64 at $t = 0.27$, indicating an optimal setting of t (assuming that precision and recall are of equal importance in the deployment setting).

In contrast, the clustering-based solution reaches its highest F1-score at $t = 0.55$ with a recall of 0.54 and a precision of 0.57; the rule-based solution reaches an F1-score of 0.41 with a recall of 0.33 and a precision of 0.53. The ensemble solution combines the advantages of both solutions and is superior to them.

Answer to RQ_1: Our ensemble approach achieves the strongest performance of the evaluated solutions for identifying developer-applied accelerations based on the selected threshold. The approach yields a maximum F1-score of 0.64 with a recall of 0.67 and a precision of 0.62 when the threshold for the inter-cluster ratio is set to 0.27.

[12]The repositories of 5 (1.5%) jobs were no longer accessible at the time of inspection.

5 RQ_2: DEVELOPER-APPLIED ACCELERATION PATTERNS

The goal of RQ_2 is to identify recurring patterns in the accelerations that developers apply. To do this, we first apply the ensemble detection approach introduced in RQ_1 to collect jobs with potential developer-applied accelerations (Section 5.1). Then, we use an open-coding process to (a) confirm that accelerations are truly being applied; and (b) manually categorize them to construct a catalog of developer-applied accelerations (Section 5.2). Finally, we present the constructed catalog of developer-applied acceleration patterns (Section 5.3).

5.1 Collection of Job Candidates

We use our ensemble approach to identify candidate build jobs that might have developer-applied accelerations. After evaluating our approach in RQ_1 (as discussed in Section 4.4.3), we set a threshold of 0.27 for the ratio measurement — jobs with a ratio below 0.27 are flagged as potentially having accelerations. Figure 6 shows how the studied build jobs are distributed with respect to the ratio measurement. When threshold $t = 0.27$, our approach detects 280 candidate jobs with potential accelerations, which we plan to inspect manually. Considering that the precision of our ensemble approach is 0.62, we expect that roughly 174 of the candidate jobs will contain developer-applied accelerations; however, considering that the recall of our ensemble approach is 0.67, we also expect that we are missing roughly 86 accelerated jobs. The 86 potential accelerations are dispersed among the remaining 2,616 jobs, making them impractical to identify without excessive effort and cost. To enhance our dataset with jobs featuring developer-applied accelerations, we include an additional 24 jobs. These jobs were identified during manual inspections in RQ_1.

In total, we have 304 candidate build jobs to inspect — 280 detected by our ensemble approach and 24 identified through RQ_1. It is important to note that some jobs overlap between these two sets, i.e., jobs labeled as accelerated in RQ_1 may also be detected as accelerated by the ensemble approach. After removing 17 duplicates, the total count of unique build job candidates is 287.

5.2 Categorization of Developer-Applied Accelerations

After collecting the set of candidate jobs, we first validate if these candidates truly include developer-applied accelerations. Then, we categorize the confirmed accelerations to build a catalog of developer-applied acceleration patterns. To achieve this goal, we adopt an open-coding approach. Figure 7 provides an overview of the approach, which includes (i) labeling candidate jobs (Section 5.2.1), (ii) assessing agreement and resolving conflicts (Section 5.2.2), and (iii) constructing a catalog of developer-applied acceleration patterns (Section 5.2.3). Below, we describe each step in the approach.

5.2.1 Label job candidates. Our open-coding approach is composed of procedure discovery and code discovery steps. Procedure discovery is required because of the lack of pre-existing categories for job classification. During this phase, we inspect each of the 287 jobs under consideration in two passes. In the first pass, we inspect a

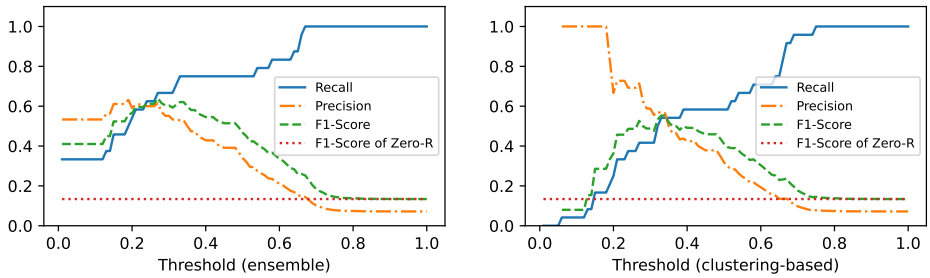


Figure 5: Comparison of detection performance between approaches. The rule-based solution equals the ensemble solution with $t = 0$.

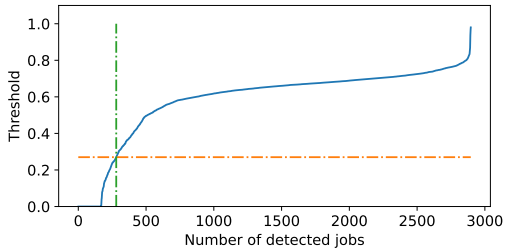


Figure 6: The vertical axis shows the threshold ratio and the horizontal axis counts jobs detected as accelerated per threshold. A solid line displays jobs our solution detects, while dash-dotted lines indicate the chosen threshold and job numbers.

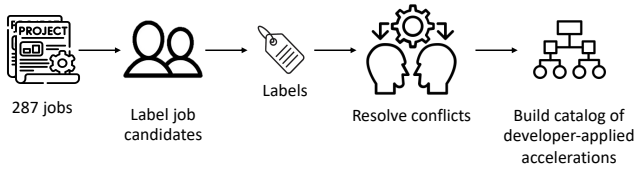


Figure 7: An overview of the open-coding approach.

randomly selected subset of 50 jobs. The first two authors of the paper independently analyze the sampled jobs, focusing on relevant artifacts that could potentially explain patterns of developer-applied acceleration within the jobs. The inspectors begin with the CircleCI specifications to understand its logic and determine if it contains any developer-applied acceleration logic. If a decision cannot be made solely based on the specification, the inspectors also check related issues, pull requests, discussions, etc.

Following this, the two inspectors meet with the third author to establish a clear procedure for subsequent inspections of each job. All three inspectors agree on a procedure that includes the following steps when checking each job:

(1) **Validate acceleration.** The goal of this step is to confirm whether the job under examination contains any form of acceleration. To achieve this, we follow the same approach that we use for labeling accelerations in RQ_1 (Section 4).
(2) **Assign labels.** If we label the job as having developer-applied acceleration (from the previous step), we assign labels describing (a) the objective of the acceleration, (b) the mechanism used for inducing acceleration, and (c) the extent or degree of the acceleration. We describe each label below.
   (a) *Purpose.* This label explains the reason why developers chose to use acceleration for the job being studied. For instance, developers might decide to skip certain tasks in the CI pipeline if a new commit only alters project metadata, such as documentation.
   (b) *Mechanism.* This label explains the method that developers use to perform acceleration.
   (c) *Magnitude.* This label explains the extent to which the job is accelerated.

   To assign each of these labels, we first inspect the CircleCI specification, focusing on the job definition section. Often, the combination of comments and implementation in this section is sufficient for us to understand the rationale of the developer-applied acceleration in the job. When the specifications refer to other scripts, we also inspect those scripts to understand their roles in and mechanisms of performing the acceleration. Moreover, we inspect referenced issues, pull requests, and discussions to gain a more complete understanding of the acceleration context. Additionally, our dataset incorporates the duration of each job step, allowing us to draw comparisons between shorter and longer invocations. This allows us to identify the step(s) that contribute the bulk of the savings, helping to understand the varying magnitude of acceleration present within the jobs.

Then, in the second pass, the first two authors independently apply the above procedure to each of the 287 candidate jobs. They each provide a detailed summary of each label to facilitate subsequent discussions and the resolution of conflicts related to the produced labels.

5.2.2 Compute agreement and resolve conflicts. Conflicts may arise in the independently produced labels for the studied jobs. More specifically, in our context, validation-type or category-type conflicts may arise. Validation-type conflicts occur when one inspector labels a job as having developer-applied acceleration, while the other labels it as non-acceleration. To assess the rate of validation-type conflicts, we calculate the Cohen's Kappa coefficient [21, 23] and obtain a Kappa score of 0.79, which is considered to be a "substantial" level of agreement. Category-type conflicts occur when

the two inspectors apply semantically different labels to a job.[13] To assess the rate of category-type conflicts, we calculate the Cohen's Kappa coefficient again and obtain a Kappa score of 0.74, which is also considered to be a "substantial" level of agreement.

To resolve both types of conflicts, the two inspectors meet with the third to inspect all conflicting labels. During the meeting, the authors articulate the rationale behind their assigned labels and incorporate feedback from the other author. All three inspectors discuss to determine the final labels based on the records that were produced during the initial inspection, such as build duration data, annotated CI specifications, and referenced scripts. Using this process, we achieve an agreement for each of the final labels.

The final set of labels identifies 169 (59%) jobs as having developer-applied accelerations. This rate is roughly in line with the precision of our ensemble approach in RQ_1 (0.62), substantiating the usefulness of our detection approach and showcasing its prospective utility in identifying developer-applied accelerations.

5.2.3 Build catalog. Similar to prior work [16, 28], we apply open-card sorting of the consolidated list of labels to construct a catalog of patterns of developer-applied acceleration. We first group patterns based on label similarity and then assign descriptive names to each of these groups to describe the higher-level category to which they belong.

5.3 Catalog of Developer-Applied Acceleration Patterns

In this section, we present the constructed catalog of developer-applied acceleration patterns. Specifically, we present the catalog of patterns in terms of the *purpose* (Section 5.3.1), *mechanism* (Section 5.3.2), and *magnitude* (Section 5.3.3) of developer-applied accelerations.

5.3.1 ***Purposes.*** Table 1 presents an overview of the 14 patterns in the *purpose* of developer-applied accelerations that we discover. Those patterns span four high-level categories. Note that a single job may include more than one developer-applied acceleration, which may span more than one pattern and/or category. As a result, the total number of jobs in a given category may not align with the sum of jobs within those patterns. Below, we describe each category in more detail.

P1. File changes (102 jobs). This category represents the most prevalent set of *purpose* patterns. CI jobs need not be triggered when a change set solely consists of a specific set of (non-code) files. When modifications are limited to files unrelated to the primary task at hand, it is probable that a CI job will yield identical results to previous runs. Executing such jobs repetitively not only wastes computational resources but could also distract developers, especially if the jobs generate (failing) outcome notifications.

This category comprises two distinct patterns. The first pattern skips CI jobs when there are no changes to specific files, whereas the second pattern skips CI jobs when change sets solely consist of changes to a list of specified files. The choice between these patterns is often contingent upon the CI workflow being applied and its requirements. Listing 3 provides an example of the pattern "skip build, test, publish, etc., when specific files are not changed." This listing scans the change set for a set of "watched files", skipping the subsequent steps if those files remain unchanged.

[13] Note that semantically similar labels, albeit phrased differently, are not deemed conflicts. For example, one job was labeled as "skip creating cache when the cache has already been made" by one author and "skip if the cache to build already exists" by the other.

Table 1: Catalog of *purpose* patterns.

Purpose	# Jobs
P1: File changes	102
Skip build, test, publish, etc., when specific files are not changed.	53
Skip build, test, publish, etc., when only specific files are changed.	52
P2: Externals	45
Skip build, test, publish, etc., when a branch/fork/PR does not satisfy specific conditions.	19
Skip build, test, publish, etc., when explicitly specified by developers.	15
Skip builds that have specific schedules.	5
Skip full packaging in specific branches.	3
Skip tasks under specific build-related conditions.	3
Skip full packaging in specific tags.	1
P3: Environmentals	38
Skip build if the artifact/cache/target already made.	23
Skip commands using external compilation cache.	10
Skip build, test, etc. when infrastructure is not ready.	4
Skip installing software if it is already installed.	1
P4: Miscellaneous	2
Skip frequent invocations.	1
Skip multiple parallel tests.	1

```
if [[ "$( git log --format=oneline -n 1 $CIRCLE_SHA1 | grep -i -E '\[skip[ _]?docs\]' )" != "" ]]; then
  echo "Skipping doc building job"
  circleci step halt
fi
```

Listing 4: Example of a job for the skip-related keyword pattern (P2).

P2. Externals (45 jobs). This category represents the second most prevalent set of purpose patterns. The predominant pattern within this category is "skip build, test, publish, etc., when a branch/-fork/PR does not satisfy specific conditions." While the examples of this pattern target different development events, they share a common practice. For instance, certain examples skip CI jobs that are triggered by fork invocations. Other examples target skipping on designated branches, i.e., there are specific branches (e.g., `dev` branches) within the code repository where certain CI jobs or steps within the job are carried out, which do not need to be performed on other branches. Other examples target skipping on pull requests (PRs) that meet specific conditions, e.g., PRs that are submitted to forks (i.e., not the main repository), can be skipped.

Another pattern is "skip build, test, publish, etc., when explicitly specified by developers." This pattern provides developers with flexibility to specify whether each triggered CI job is necessary using a keyword in the commit message. Listing 4 shows an example of this pattern, which checks whether the Git commit message matches a pattern, and if so, the job skips documentation steps.

Table 2: Catalog of *mechanism* patterns.

Mechanism	# Jobs
M1: Environmental variables and parameters	82
Check branch name, PR number, repository name, etc. from CircleCI environmental variables.	77
Check developer-applied environmental variables.	3
Check pipeline parameters.	3
Use command arguments to control the acceleration.	1
M2: Git commands	68
Check file changes using Git commands.	56
Check metadata using Git commands.	12
M3: External tools and APIs	47
Check GitHub APIs.	38
Use external compilation cache tools.	8
Check Docker image tag from API.	1
M4: File commands	36
Check flag files.	34
Check target files.	11
Use shell commands.	3
Use commands defined in package manager files.	1
M5: Logs/outputs	4
Check trace log.	2
Check test log.	1
Check program output.	1

P3. Environmentals (38 jobs). This category encompasses patterns in which jobs are skipped based on the setting in which the CI job is being executed. Acceleration of CI jobs is contingent on the presence of output or dependencies within the CI environment. For example, developers often skip CI steps when the output of the step can be reused from a prior CI job. The most common pattern in this category involves skipping steps or entire jobs when the artifact, cache, or target already exists. For instance, CI jobs may create dependencies that are used by other jobs. developer-applied accelerations that fit this purpose pattern can determine if these dependencies are up-to-date, potentially allowing them to skip the build phase.

P4. Miscellaneous (2 jobs). We categorize the remaining two jobs under the miscellaneous category. Such jobs have context-specific conditions that make them unique to certain platforms or situations. The "skip frequent invocations" pattern involves a condition where a CI job is skipped if it is not its first invocation of the day. Similarly, the pattern "skip multiple parallel tests" is specific to a feature offered by CircleCI.[14]

5.3.2 ***Mechanisms.*** Table 2 presents an overview of our catalog of 16 patterns in the *mechanisms* that are applied to perform developer-applied accelerations, which span five categories. As previously noted, multiple patterns may apply to a single CI job. Below, we present each mechanism category.

M1. Environmental variables and parameters (82 jobs). This category (i.e., the most common mechanism category) incorporates mechanisms that review environmental variables, including those listed in CircleCI documentation and those created by developers, as well as pipeline and program parameters. An example of this mechanism is when jobs extract job-relevant details from environmental variables that are created by the CircleCI platform. These variables hold details about the CI job and its context.[15] Additionally, CircleCI users have the option to create their own variables within CI and/or build specifications.[16] Users also can use pipeline parameters[17] to shape the behaviour of a CI job that spans multiple pipelines. Listing 2 provides an example of this category. The specification in the listing examines the name of the branch. If the branch name, retrieved from environmental variables, does not begin with "release" or "test," the subsequent steps are skipped.

```
1  if [ "$USE_SCCACHE" == "true" ]; then
2    # https://github.com/mozilla/sccache
3    SCCACHE_PATH="$PWD/src/electron/
       external_binaries/sccache"
4    echo 'export SCCACHE_PATH="'"$SCCACHE_PATH"'"'
       >> $BASH_ENV
5    if [ "$CIRCLE_PR_NUMBER" != "" ]; then
6      #if building a fork set readonly access to
       sccache
7      echo 'export SCCACHE_BUCKET="electronjs-
       sccache-ci"' >> $BASH_ENV
8      echo 'export SCCACHE_TWO_TIER=true' >>
       $BASH_ENV
9    fi
10 fi
```

Listing 5: Example of a job accelerated using cache tools (M3).

M2. Git commands (68 jobs). This category (i.e., the second most prevalent mechanism category) encapsulates two patterns. The first pattern analyzes change sets using Git commands, such as `git diff`. Those in the second pattern analyze commit metadata, such as the commit message. Listings 3 and 4 provide examples of this category of mechanism.

M3. External tools and APIs (47 jobs). These accelerations use external tools and APIs to decide whether to skip or accelerate CI steps within a job. An example of this mechanism involves using external caching tools like `ccache`.[18] Listing 5 illustrates how `ccache` is initialized. First, the job examines if the environmental variable is analyzed on line 1 to determine whether the CI job should use `sccache`, followed by its setup on lines 2–9.

M4. File commands (36 jobs). To perform the acceleration, this category of mechanism relies on an inspection of file properties using Bash commands on Linux and PowerShell commands on Windows. The most frequently occurring pattern in this category is "check flag files." These flag files act like on/off switches, determining whether the acceleration should be enabled. Listing 6 provides an example of this pattern. In lines 16–20 of the listing, the build date is stored in a file, which is persisted using the cache feature of the CI platform. In future CI job invocations, the file is retrieved from the cache (lines 1–3), and its content is checked. If the date stored in the file matches the current date, the job is skipped (lines 4–15). In the "check target files" pattern, jobs are associated with specific files. For example, in certain tasks, the "target file" is the `dist` folder. If the `dist` folder already exists, it means the required

[14] https://circleci.com/docs/parallelism-faster-jobs/

[15] https://circleci.com/docs/variables/#built-in-environment-variables
[16] https://circleci.com/docs/env-vars/
[17] https://circleci.com/docs/pipeline-variables/
[18] https://ccache.dev/

```
- restore_cache:
    name: restore daily latch cache file.
    key: code-coverage-daily
- run:
    name: Halt job if already built code coverage
    today.
    command: |
      NOW=`date +%Y-%m-%d`
      LAST=`cat /home/circleci/lastbuild` || true
      if [[ "$LAST" == "$NOW" ]]; then
        echo Last build occured today, halting.
        circleci step halt
      else
        echo Last build occured $LAST, building.
        date +%Y-%m-%d > /home/circleci/lastbuild
      fi
- save_cache:
    name: store updated daily latch file.
    key: code-coverage-daily-{{ epoch }}
    paths:
    - /home/circleci/lastbuild
```

Listing 6: Example of a job accelerated under pattern (M4).

```
SHOULD_LINT_ALL=$(./Tests/scripts/should_lint_all.
    sh)
mkdir ./unit-tests
if [ -n "$SHOULD_LINT_ALL" ]; then
  echo -e  "----------\nLinting all because:\n${
    SHOULD_LINT_ALL}\n----------"
  demisto-sdk lint -p 8 -a -q --test-xml ./unit-
    tests --log-path ./artifacts --failure-report
    ./artifacts
else
  demisto-sdk lint -p 8 -g -v --test-xml ./unit-
    tests --log-path ./artifacts --failure-report
    ./artifacts
fi
```

Listing 7: Example of a job that skips a partial step.

distribution artifacts (e.g., the generated website files) are ready, and steps within the job can be skipped.

M5. Logs/outputs (4 jobs). Jobs classified under this category perform acceleration according to the outputs or logs of certain build procedures, i.e., by analyzing standard output/error streams[19] or the logs that are generated during job execution. For instance, a step within a job might review previous step logs to decide whether to skip tests, particularly when a test log already exists.

5.3.3 ***Magnitudes.*** Our inspection reveals that jobs can be classified into three categories based on their acceleration magnitude:

- **Remaining steps (78%)**: This is the most prevalent category. Jobs under this category skip all subsequent steps.
- **Current step (12%)**: Jobs under this category skip the remainder of the step that is currently being executed.
- **Partial step (10%)**: Jobs under this category skip certain tasks within the current step.

[19]https://en.wikipedia.org/wiki/Standard_streams

In the "remaining steps" category, CircleCI command-line tools, such as using "`circleci step halt`", are commonly used. An example of this acceleration can be seen in Listing 4, where the halt command appears on line 3. The "current step" category typically involves operations that terminate the currently executing script. For instance, the `exit` command can be used to skip the remaining statements in a script. The "partial step" pattern involves executing specific commands only on a particular branch. Listing 7 is an example from the `demisto/content` project, where conditions are checked to decide whether to run the full linting procedure (line 3) or a subset (specified using the arguments on lines 5 & 7).

Answer to *RQ*$_2$: Emergent patterns that explain developer-applied accelerations include *file changes (P1)*, where specific job steps are skipped due to changes made to specific files, as well as *externals (P2)* and *environmentals (P3)*, where job acceleration occurs based on external and environmental factors. Developers use various mechanisms to perform these accelerations, such as analyzing the state of *environmental variables (M1)*, issuing *commands (M2, M4)* and integrating *external tools and APIs (M3)*. In the majority of examined jobs (78%), developer-applied accelerations result in the complete skipping of all subsequent job steps; however, step (12%) or sub-step (10%) skipping behaviour is not uncommon.

6 RELATED WORK

Below, we situate our work with respect to the literature on CI.

CI provides numerous benefits to the software teams that adopt it [15, 32]. For example, Hilton et al. [15] explored the adoption and implications of CI across 34,544 open-source projects, revealing that 40% of these projects adopt CI. Surveys of developers within these projects indicated that early bug detection capabilities and preventing contributors from releasing breaking builds are major motivations for CI adoption. Additionally, Vasilescu et al. [32] studied a dataset of 246 open-source projects that use CI and found that CI was associated with increased developer productivity.

Adopting CI presents its challenges. Several studies have discussed the hurdles that developers face [11–13]. For example, Hilton et al. [12] surveyed 523 developers to identify barriers that developers face, with 50% of the respondents reporting problems troubleshooting CI builds, and 52% of them preferring simplified configuration options for CI tools. Ghaleb et al. [11] modeled long build durations using a dataset of 104,442 builds that span 67 projects.

Other studies proposed approaches to reduce CI build time by skipping unnecessary builds or steps within builds [1, 2, 7, 18, 34, 35]. For example, Gallaba et al. [7] proposed a language-agnostic approach (Kotinos) to infer data from which build acceleration decisions can be made. They found that at least 87.9% of the 14,364 studied CI build records contained at least one Kotinos acceleration in their production setting. Abdalkareem et al. [2] examined 1,813 commits where developers requested for CI builds to be skipped. This analysis revealed reasons for skipping CI builds (e.g., skip builds when changes are made to documentation). Based on those reasons, they proposed a rule-based method (CI-Skipper) that automatically identifies commits that could be CI skipped. In another

study, Abdalkareem et al. [1] trained a decision tree classifier to detect CI-skippable commits. Jin and Servant [18] proposed PreciseBuildSkip—an enhanced CI-Skipper that considers additional rules to maximize the rate of build failure observation.

Other studies explored ways to speed up CI using testing methods. They focused on reducing the number of tests being invoked (i.e., test case selection), yet ensuring the most likely-to-fail tests are run [5, 26, 27, 37]. For example, Shi et al. [26] proposed a tool (iFixFlakies) to mitigate the inefficiency of build rerunning, specifically in the context of flaky tests. Durieux et al. [5] analyzed a dataset of 3,286,773 builds, and found that 56,522 of those builds were restarted due to timeouts and flaky tests.

Previous research has primarily focused on the benefits of and obstacles to CI adoption, as well as proposing approaches to accelerate CI processes. However, to the best of our knowledge, the concept of developer-applied accelerations remains unexplored. Inspired by these prior studies, we bridge that gap by proposing a detection approach, which we use to identify developer-applied accelerations. Our inspection of detected examples yields a catalog of reusable patterns of developer-applied accelerations. Furthermore, our study enhances the current understanding of CI acceleration by providing empirical evidence that goes beyond the anecdotal insights typically found in grey literature. While blog posts offer practical tips for accelerating CI processes,[20] they do not typically include the kind of systematic validation across diverse real-world scenarios that our paper includes. Our study not only documents but also evaluates CI acceleration techniques that developers have implemented within the CircleCI platform in open-source projects. One such example is the strategy of skipping unaffected steps. While this technique is mentioned both in our research and in various blog posts,[21] there is a lack of clarity regarding its actual adoption in practice. For example, specific patterns, such as categories that are related to skipping builds when particular files remain unchanged, are commonly adopted by developers (31.4%). In contrast, patterns that are related to skipping builds on specific schedules (3.0%) are not yet widely adopted.

7 THREATS TO VALIDITY

Construct Validity. Construct validity is concerned with the degree to which our measurements capture what we aim to study. Our procedure for producing the catalog of developer-applied accelerations is based on the opinions of the inspectors, and as such, is subject to inspector bias. We address this by having multiple authors label job samples, achieving a substantial Cohen's Kappa score of +0.79. We mitigate this threat by having multiple authors label a sample of jobs, yielding a Cohen's Kappa agreement score of +0.79 (considered "substantial" agreement), suggesting that our catalog is robust to inspector bias.

Internal Validity. We categorize the accelerations based on 280 acceleration-prone jobs and the 24 accelerated jobs from samples. Since the majority is from detection, the categorization may be biased because some categories may be more likely to be caught by our detection. We mitigate this threat by involving the 24 samples. Involving the 24 samples is the way we mitigate this threat.

External Validity. Our study is based on projects that adopt CircleCI. Hence, our results may not generalize to projects that use other CI providers. Nonetheless, CI providers typically use YAML-based configurations that are highly similar to CircleCI (e.g., `.circleci/config.yml` for CircleCI, `.travis.yml` for Travis CI), and hence, our results may likely generalize to other CI providers.

8 CONCLUSION & LESSONS LEARNED

This paper proposes approaches to detect developer-applied CI accelerations. We evaluate our approaches using a large-scale dataset of 2,896 CircleCI jobs, observing that our ensemble approach achieves an F1-score of 0.64 with a recall of 0.67 and a precision of 0.62. Having achieved reasonable performance scores, we conduct a qualitative analysis of the detected developer-applied accelerations to create a detailed catalog of patterns. We conjecture that this catalog will provide the following benefits for CI consumers and providers:

- ***CI consumers* can leverage the patterns in our catalog to access tailored strategies for CI accelerations.** Our study reveals a catalog of patterns that developers leverage to enhance the CI process in their projects, especially less recognized patterns (e.g., P2 & P3 in Section 5.3). For example, developers can use our patterns to accelerate their CI by removing costly redundancies, such as unnecessary external service calls or rebuilds triggered by minor typos. They can also explore beyond standard CI documentation by inspecting unique aspects of build metadata, like timestamps and commit messages.
- ***CI providers* (platforms) can leverage the patterns in our catalog to promote built-in acceleration features.** Our catalog can serve as a resource for enhancing CI platforms by aligning with identified patterns and introducing new features. For example, CI platforms can use branch filters to decide if a job should run on specific branches, thus reducing unnecessary resource use and feedback delays. Additionally, CI platforms can develop new acceleration features inspired by our catalog. Features such as allowing developers to skip builds based on environmental variable analysis and integration with APIs could be essential. For example, developers could specify job skipping in commit messages, a feature not currently offered by CircleCI, despite a clear demand for such capabilities.[22]

DECLARATIONS

Funding. This work was supported by the Natural Sciences and Engineering Research Council (NSERC) of Canada, the financial support of JST for PRESTO grant JPMJPR22P3, the financial support of JSPS for the Bilateral Program grant JPJSBP120239929, and the Inamori Research Institute for Science via the InaRIS Fellowship.

Disclaimer. The findings and opinions in this paper belong solely to the authors, and are not necessarily those of Huawei. Moreover, our results do not in any way reflect the quality of Huawei products.

[20] https://dev.to/zenika/gitlab-ci-optimization-15-tips-for-faster-pipelines-55al
[21] https://claroty.com/blog/engineering-speed-up-your-ci-cd-pipeline
[22] https://circleci.canny.io/cloud-feature-requests/p/add-ability-to-filter-by-commit-messages-in-workflows

REFERENCES

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

[2] Rabe Abdalkareem, Suhaib Mujahid, Emad Shihab, and Juergen Rilling. 2019. Which commits can be CI skipped? *Transactions on Software Engineering* 47 (2019).

[3] Bihuan Chen, Linlin Chen, Chen Zhang, and Xin Peng. 2020. Buildfast: History-aware build outcome prediction for fast feedback and reduced cost in continuous integration. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 42–53.

[4] Jürgen Cito, Gerald Schermann, John Erik Wittern, Philipp Leitner, Sali Zumberi, and Harald C Gall. 2017. An empirical analysis of the docker container ecosystem on github. In *Proceedings of the 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR'17)*. 323–333.

[5] Thomas Durieux, Claire Le Goues, Michael Hilton, and Rui Abreu. 2020. Empirical study of restarted and flaky builds on Travis CI. In *Proceedings of the 17th International Conference on Mining Software Repositories*.

[6] Paul M Duvall, Steve Matyas, and Andrew Glover. 2007. *Continuous integration: improving software quality and reducing risk*.

[7] Keheliya Gallaba, John Ewart, Yves Junqueira, and Shane Mcintosh. 2020. Accelerating continuous integration by caching environments and inferring dependencies. *Transactions on Software Engineering* (2020).

[8] 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 *Proceedings of the 44th International Conference on Software Engineering*.

[9] 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 *Proceedings of the 44th International Conference on Software Engineering (ICSE'22)*. 1330–1342. https://doi.org/10.1145/3510003.3510211

[10] Keheliya Gallaba and Shane McIntosh. 2018. Use and misuse of continuous integration features: An empirical study of projects that (mis) use Travis CI. *IEEE Transactions on Software Engineering* 46, 1 (2018), 33–50.

[11] 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.

[12] Michael Hilton, Nicholas Nelson, Danny Dig, Timothy Tunnell, Darko Marinov, et al. 2016. Continuous integration (CI) needs and wishes for developers of proprietary code. (2016).

[13] Michael Hilton, Nicholas Nelson, Timothy Tunnell, Darko Marinov, and Danny Dig. 2017. Trade-offs in continuous integration: assurance, security, and flexibility. In *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering*.

[14] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. 2016. Usage, costs, and benefits of continuous integration in open-source projects. In *Proceedings of the 31st International Conference on Automated Software Engineering*.

[15] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. 2016. Usage, Costs, and Benefits of Continuous Integration in Open-Source Projects. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE'16)*. 426–437. https://doi.org/10.1145/2970276.2970358

[16] 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 (FSE'19)*. 578–589. https://doi.org/10.1145/3338906.3338949

[17] Xianhao Jin and Francisco Servant. 2020. A cost-efficient approach to building in continuous integration. In *Proceedings of the 42nd International Conference on Software Engineering*. 13–25.

[18] 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).

[19] Eero Laukkanen, Juha Itkonen, and Casper Lassenius. 2017. Problems, causes and solutions when adopting continuous delivery—A systematic literature review. *Information and Software Technology* 82 (2017), 55–79.

[20] Mateusz Machalica, Alex Samylkin, Meredith Porth, and Satish Chandra. 2019. Predictive test selection. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*.

[21] Mary L McHugh. 2012. Interrater reliability: the kappa statistic. *Biochemia medica* 22, 3 (2012), 276–282.

[22] Shane McIntosh, Bram Adams, and Ahmed E. Hassan. 2012. The evolution of Java build systems. *Empirical Software Engineering* 17, 4-5 (2012), 578–608.

[23] Mahtab Nejati, Mahmoud Alfadel, and Shane McIntosh. 2023. Code Review of Build System Specifications: Prevalence, Purposes, Patterns, and Perceptions. In *Proceedings of the 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE'23)*. 1213–1224. https://doi.org/10.1109/ICSE48619.2023.00108

[24] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[25] Islem Saidani, Ali Ouni, Moataz Chouchen, and Mohamed Wiem Mkaouer. 2021. Bf-detector: an automated tool for ci build failure detection. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1530–1534.

[26] August Shi, Wing Lam, Reed Oei, Tao Xie, and Darko Marinov. 2019. iFixFlakies: A framework for automatically fixing order-dependent flaky tests. In *Proceedings of the 27th Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*.

[27] August Shi, Peiyuan Zhao, and Darko Marinov. 2019. Understanding and Improving Regression Test Selection in Continuous Integration. In *Proceedings of the 2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE'19)*. 228–238. https://doi.org/10.1109/ISSRE.2019.00031

[28] Mini Shridhar, Bram Adams, and Foutse Khomh. 2014. A Qualitative Analysis of Software Build System Changes and Build Ownership Styles. In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM'14)*. Article 29, 10 pages. https://doi.org/10.1145/2652524.2652547

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

[30] Daniel Ståhl and Jan Bosch. 2016. Industry application of continuous integration modeling: a multiple-case study. In *Proceedings of the 38th International Conference on Software Engineering Companion*.

[31] Gengyi Sun, Sarra Habchi, and Shane McIntosh. 2024. RavenBuild: Context, Relevance, and Dependency Aware Build Outcome Prediction. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 996–1018.

[32] 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 10th joint meeting on foundations of software engineering*.

[33] Carmine Vassallo, Sebastian Proksch, Harald C. Gall, and Massimiliano Di Penta. 2019. Automated Reporting of Anti-Patterns and Decay in Continuous Integration. In *Proceedings of the 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE'19)*. 105–115. https://doi.org/10.1109/ICSE.2019.00028

[34] Nimmi Weeraddana, Mahmoud Alfadel, and Shane McIntosh. 2024. Characterizing Timeout Builds in Continuous Integration. *IEEE Transactions on Software Engineering* 50, 6 (2024), 1450–1463.

[35] Nimmi Rashinika Weeraddana, Mahmoud Alfadel, and Shane McIntosh. 2024. Dependency-Induced Waste in Continuous Integration: An Empirical Study of Unused Dependencies in the npm Ecosystem. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 2632–2655.

[36] David Gray Widder, Michael Hilton, Christian Kästner, and Bogdan Vasilescu. 2019. A Conceptual Replication of Continuous Integration Pain Points in the Context of Travis CI. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE'19)*. 647–658. https://doi.org/10.1145/3338906.3338922

[37] C. Zhu, O. Legunsen, A. Shi, and M. Gligoric. 2019. A Framework for Checking Regression Test Selection Tools. In *Proceedings of the 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE'19)*. 430–441. https://doi.org/10.1109/ICSE.2019.00056