Lessons from Eight Years of Operational Data from a Continuous Integration Service

An Exploratory Case Study of CircleCI

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

Maxime Lamothe
Polytechnique Montréal
Montréal, Canada
maxime.lamothe@polymtl.ca

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

ABSTRACT

Continuous Integration (CI) is a popular practice that enables the rapid pace of modern software development. Cloud-based CI services have made CI ubiquitous by relieving software teams of the hassle of maintaining a CI infrastructure. To improve these CI services, prior research has focused on analyzing historical CI data to help service consumers. However, finding areas of improvement for CI service providers could also improve the experience for service consumers. To search for these opportunities, we conduct an empirical study of 22.2 million builds spanning 7,795 open-source projects that used CircleCI from 2012 to 2020.

First, we quantitatively analyze the builds (i.e., invocations of the CI service) with passing or failing outcomes. We observe that the heavy and typical service consumer groups spend significantly different proportions of time on seven of the nine build actions (e.g., dependency retrieval). On the other hand, the compilation and testing actions consistently consume a large proportion of build time across consumer groups (median 33%). Second, we study builds that terminate prior to generating a pass or fail signal. Through a systematic manual analysis, we find that availability issues, configuration errors, user cancellation, and exceeding time limits are key reasons that lead to premature build termination.

Our observations suggest that (1) heavy service consumers would benefit most from build acceleration approaches that tackle long build durations (e.g., skipping build steps) or high throughput rates (e.g., optimizing CI service job queues), (2) efficiency in CI pipelines can be improved for most CI consumers by focusing on the compilation and testing stages, and (3) avoiding misconfigurations and tackling service availability issues present the largest opportunities for improving the robustness of CI services.

KEYWORDS

Automated Builds, Build Systems, Continuous Integration

ICSE '22, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9221-1/22/05…$15.00
https://doi.org/10.1145/3510003.3510211

ACM Reference Format:
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 *44th International Conference on Software Engineering (ICSE '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3510003.3510211

1 INTRODUCTION

Continuous Integration (CI) [9] is a software development practice in which development events (e.g., the creation of and updates to pull requests, pushes to central code repositories) trigger build, test, and reporting routines automatically. The main goal of CI is to provide fast feedback to developers, allowing them to verify whether their changes cleanly integrate with other changes. Indeed, the benefits of CI, such as increases in developer productivity and improved software quality, have been observed by the software development community [42]. Both open source [4] and proprietary [10, 37] software organizations have dedicated resources to maintaining and operating CI pipelines for this purpose.

Dedicated cloud-based CI providers, such as CircleCI and Travis CI offer CI services for software organizations. These services provide consumers with the benefits of CI without the hassle of provisioning, operating, and maintaining CI infrastructure.

The broad adoption of CI services has facilitated research on CI. Researchers have interpreted the outcome [13] and duration [15] of the builds run by these CI providers from the perspective of the CI consumers, discussing the challenges and benefits of adopting CI [47]. However, the CI providers' perspective has remained largely unexplored. Focusing on build data from the perspective of CI service providers could uncover opportunities to holistically improve the CI solutions that swaths of software teams rely upon. For example, focusing optimization effort on slow CI stages could drive down service costs for the CI service providers and simultaneously provide fast feedback to their consumers.

To that end, we conduct an exploratory case study of CircleCI—one of the most popular CI service providers for projects hosted on GitHub. Our observations are likely to generalize to other popular cloud-based CI/CD providers (e.g., Travis CI, GitHub Actions) with similar features (e.g., YAML-based configuration, social coding platform integration, container-based orchestration, support job-based parallelism for projects that target multiple execution platforms). Our analysis includes 22.2 million builds spanning 7,795 open source projects that used CircleCI during the period of 2012–2020. This data enables us to address the following research questions:

RQ1: How does the usage of a CI service change over time?
Motivation: Studying the growth of CircleCI in terms of active consumers and productivity heuristics may help to identify specific areas that need attention to improve resource allocation and the overall user experience.
Results: During the last eight years, the CircleCI service has grown, in terms of both monthly consumers and total number of builds that are invoked every month. However, this growth has stagnated since mid-2020. At least 14% of projects that were inactive on CircleCI during 2020 have started using another CI service. Throughput and build duration have been increasing (up to 25 minutes per build and 900 builds per month) for the platform's heaviest consumers, while median values have remained stable. Inequality in terms of build execution rates and resource consumption has steadily increased to Gini coefficients of 95% and 98%.
Recommendations: Additional resource usage incurred by growth could be tackled by applying build acceleration and skipping approaches from the literature. The high success rate shows that there is a large pool of builds that could likely be safely skipped. Automated repair techniques can be used to further reduce MTTR.

RQ2: How is time spent during signal-generating builds?
Motivation: Signal-generating builds terminate with either a pass or fail outcome. By understanding the time consumption of different steps in the CI pipeline, service providers can identify resource bottlenecks and estimate operational costs. Identifying stages that slow down the CI pipeline can allow researchers and developers to target the most impactful stages.
Results: Compared to other stages in the CI process, a larger proportion (median 35%) of the CI runtime is spent on the compilation and testing stages.
Recommendations: Focusing research efforts on accelerating testing and compilation steps in the CI pipeline will yield the largest reductions to CI workload costs for CI providers and feedback delays for CI consumers. The heaviest consumers will benefit most from the allocation of additional network bandwidth during dependency installation or the deployment of local mirrors of dependency archives.

RQ3: Why are some builds unable to provide a signal?
Motivation: When a build does not provide a signal, developers are not provided with feedback about the changes that they have submitted. Since receiving early feedback is a key feature of CI, these non-signal-generating builds are a concern. Studying why builds fail early without providing a signal can help service providers and researchers to develop approaches to mitigate such instances, yielding more robust and available CI services.
Results: Most non-signal-generating builds occur due to user cancellation (46.4%), availability issues (22.0%), configuration errors (1.8%), and exceeding time limits (1.1%).
Recommendations: Promising directions for research include the analysis of why consumers cancel builds, and the development of approaches to improve traceability of orphaned builds and identify builds that are likely to time out.

2 CORE CONCEPTS IN MODERN CI

The **consumers** of CI services may subscribe with multiple **projects**, each with its own CI **workflow**. A workflow is comprised of one or more **jobs**. The result of executing a workflow is referred to as a **build**. CI services typically support specifying workflows in a configuration file (e.g., `.circleci/config.yml` for CircleCI, `.travis.yml` for Travis CI). Although this configuration file is primarily used to specify the sequence of commands to be executed, other options can be customized like:

Parallelism: The number of parallel instances of a job to run.
Environment: Environment variables to be set for each build.
Resource Class: The compute resources allocated to each job.

Workflows can be configured to invoke builds based on development events (e.g., when a pull request is created or updated or when a push to a central repository is performed), on a schedule (e.g., nightly), or manually (e.g., programmatically via an API request or on-demand to retry a build without changing the code). Once a build request is received, depending on the subscription of the consumer and the workflow configuration of the project, a build environment (e.g., a set of physical machines, virtual machines, and/or containers) is allocated to execute the build.

2.1 CI Build Outcomes

CI builds are executed with the expectation that they will produce a signal indicating whether changes to the codebase are ready to be integrated. However, in practice, build outcomes are not always conclusive. Therefore, we categorize build outcomes as either:

Signal-generating. Builds that execute until a passing or failing outcome is produced. If a build passes, CI consumers know that the associated changes to the codebase have at least passed the baseline checks. If a build fails, CI consumers can diagnose the problems with their changes while design decisions are still fresh in their minds.

Non-signal-generating. Builds that are terminated before completion. A build could prematurely terminate due to a user aborting the build, configuration errors, or infrastructure provisioning issues. These builds do not provide consumers with a meaningful signal about their changes.

In an ideal CI pipeline, all builds are signal generating. If non-signal-generating builds occur often, service consumers will lose faith in a provider's ability to deliver a meaningful CI signal.

2.2 CI Indicators

CI providers have proposed indicators to track performance in CI pipelines. CircleCI[1] has recently proposed four such indicators:

Build Duration. The time taken for a build to execute. A long duration may force developers to switch contexts—a costly action for knowledge workers like software engineers [30]. A short duration may indicate inadequate testing is included in the CI workflow.

Mean Time to Recovery (MTTR). The average length of the time interval between the end times of failing and subsequent passing builds. A long MTTR suggests that failures

[1] https://circleci.com/resources/data-driven-ci/

are difficult to diagnose or resolve, or a lack of build status monitoring. A short MTTR suggests that consumers are quickly resolving build failures without too much disruption.

Success Rate. The proportion of signal-generating builds with passing outcomes. The importance of the success rate is context sensitive. The success rate in the main branch where a large team collaborates should be kept high to avoid impeding development progress. On the other hand, the success rate in a developer-specific feature branch may be irrelevant.

Throughput. The number of builds that are performed during a given period of time. Throughput may vary across projects based on the development model and the team size. This metric gives an indication of expected total server load and network bandwidth usage for the CI service provider.

These four indicators provide insight into the maturity of CI adoption among CircleCI consumers. Individual software organizations also may find these indicators useful to understand how their CI usage compares with others. Recently, CircleCI started providing these metrics to their users on a per-workflow basis within their web interface. In this study, we use these indicators to characterize the growth of CI usage over time.

3 STUDY DESIGN

In this section, we provide our rationale for studying CircleCI (Section 3.1), and describe our data extraction (Section 3.2) and data analysis approaches (Section 3.3).

3.1 Subject CI Provider

With the popularity of CI as a software development practice, many cloud-based providers offer CI services. A Forrester market report[2] identified five leaders in the cloud-native CI area by comparing their current product offerings, strategy, and market presence, namely: (1) Google Cloud Build, (2) AWS Code Build, (3) Azure DevOps Service, (4) GitLab, and (5) CircleCI. Among these area leaders, according to GitHub marketplace statistics,[3] CircleCI has the largest number of installs (748k). Therefore, considering its popularity and that CI build data for a large number of its users is openly available for a span of eight years, we choose to focus our analysis on CircleCI. As a leading CI platform, CircleCI has served over one million developers during its nine years of operation.[4]

3.2 Data Extraction and Filtering

Figure 1 provides an overview of our data extraction and filtering approach. We describe each step below.

To arrive at reliable conclusions representing the workload of a typical CI service provider, it is important that we access all publicly available build data for projects that use CircleCI. We start by querying for projects that use CircleCI in the public GitHub dataset on Google BigQuery,[5] one of the largest publicly available datasets of software repositories. For this purpose, we check for projects that have a `.circleci/config.yml` configuration file in their version control system (see DE1 of Figure 1). This query identifies 10,170 projects with a CircleCI configuration.

Having a CircleCI configuration file in its version control system is a necessary but not sufficient condition to conclude that a project uses the CircleCI service. Even with a CI configuration file, it is possible that the CI service was never activated or that no CI builds were run for a particular project. Therefore, we query the CircleCI API for projects that have run at least one CI build in their lifetime (see DE2 of Figure 1). A corpus of 8,259 projects survive this filter.

Next, via the CircleCI API, we retrieve the metadata of the builds that are associated with the surviving projects. A total of 23,330,690 build records were retrieved (see DE3 of Figure 1). We remove builds that have missing values for mandatory fields (i.e., `platform`, `vcs_url`, `build_num`) because we, as external observers, cannot determine why these fields are incomplete. We also remove builds that were started after December 31st, 2020 (see DE4 of Figure 1). We chose this cutoff date to allow only completed calendar years into our dataset for analysis. We use the 22,238,413 unique builds spanning 7,795 projects for further analysis.

3.3 Data Analysis

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

We first extract the outcome of each build and label each one as signal-generating or non-signal-generating. We label builds with an outcome of *success* or *failed* as signal-generating builds because these builds provide a conclusive signal to the user, reporting whether their proposed changes can be integrated into the mainline of development or not. All builds with other outcomes are categorized as non-signal-generating because they were prematurely terminated without providing a signal.

Next, we identify projects that are heavy CI consumers. We consider projects that consume a large proportion of the build time as heavy consumers and determine a threshold for this consumption. To identify these heavy CI consumers, we first calculate the total monthly build time consumption for all projects. Then, we consider a monthly build time consumption threshold value, above which projects are considered heavy CI consumers. We then modify this threshold value to verify its impact on the size of the heavy CI user sample that we obtain. We consider a threshold value acceptable if fluctuations in the threshold value present little variance in the number of heavy CI consumers.

We use all build activity data to answer RQ1. To answer RQ2 and RQ3, we use signal-generating and non-signal-generating builds, respectively.

4 STUDY RESULTS

In this section, we present the results of our study with respect to our research questions. For each research question, we describe the approach used to address it, and the results that we observe.

(RQ1) How does the usage of a CI service change over time?

RQ1: Approach. For each month throughout the studied period from 2012 to 2020, we plot the number of projects that use CircleCI

[2] https://www.forrester.com/report/The+Forrester+Wave+CloudNative+Continuous+Integration+Tools+Q3+2019/-/E-RES148217

[3] https://github.com/marketplace?category=continuous-integration&query=sort%3Apopularity-desc

[4] https://circleci.com/milestones/

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

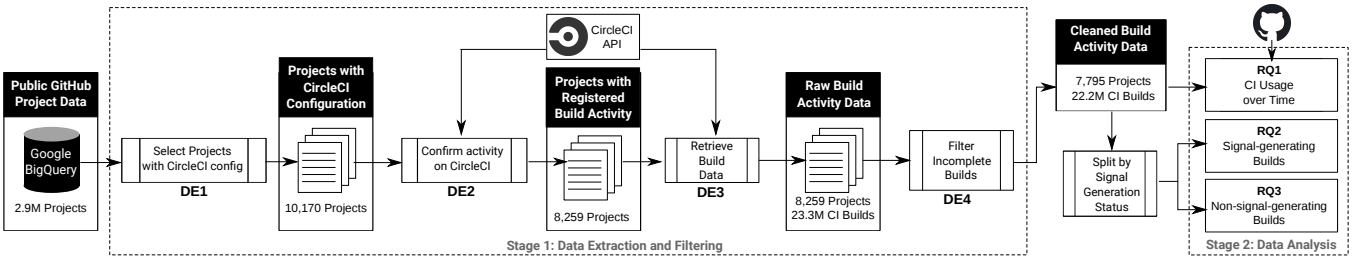


Figure 1: An overview of the approach we followed for data analysis

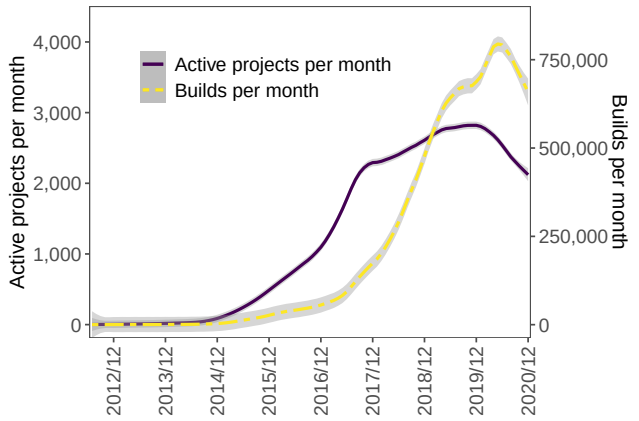


Figure 2: The growth of CircleCI usage during the period of 2012–2020. The number of projects that used CircleCI in each month is shown in purple. The number of CircleCI builds of these projects in each month is shown in yellow. Both lines are Loess-smoothed curves with gray shaded areas indicating the 95% confidence interval.

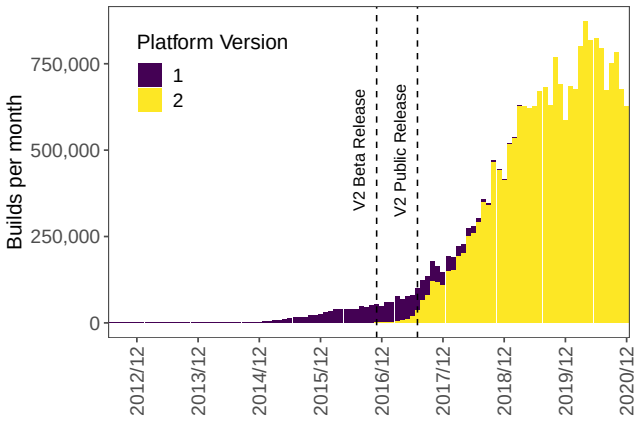


Figure 3: Number of builds on CircleCI platforms 1 and 2 during the 2012–2020 time period.

and the number of total builds that are run on CircleCI. Then we plot the growth of CircleCI's usage by computing the project-level values of: (a) Build Duration, (b) Mean Time to Recovery (MTTR), (c) Success Rate, and (d) Throughput (see Section 2.2).

RQ1: Results. Figure 2 shows the growth of CircleCI usage during the studied period, in terms of the number of projects that are using CircleCI (shown in blue) and the number of total builds that are executed on CircleCI (shown in red).

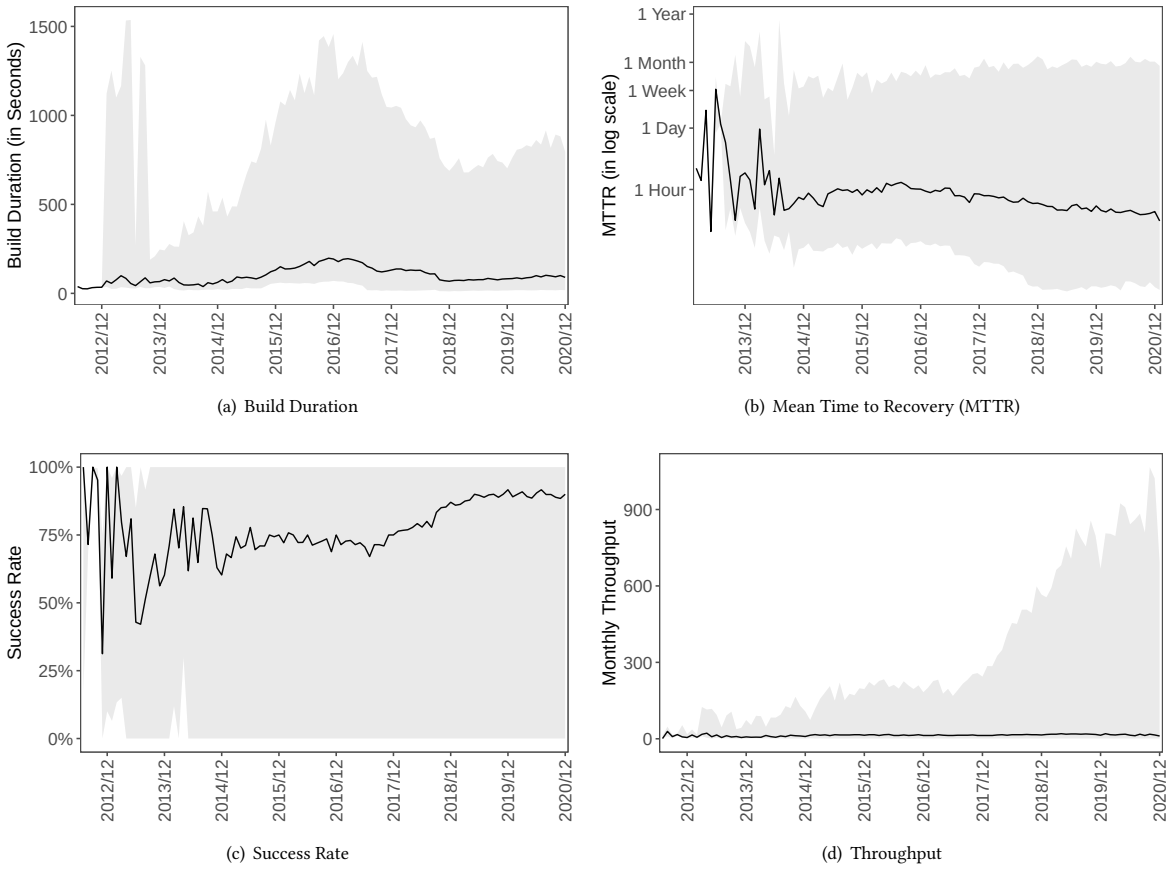
Observation 1*: The number of builds per month across all studied projects grew over the years, reached a peak of 872,842 builds during the month of April 2020, and then declined.* Competitive pricing and new features offered by other service providers such as GitLab CI and GitHub Actions—GitHub's own automation service—may have contributed to a user exodus from CircleCI. For instance, compared to the 250 free minutes of build time per month in CircleCI, GitHub Actions provides 20 free parallel builds and unlimited build minutes for every open source project. There are other community and technical factors at play. For example, Widder *et al.* [46] observed that projects with more pull requests tend to be less likely to abandon a CI service. This suggest that the projects derive value from CI by using it to evaluate pull requests. Similarly, Widder *et al.* found that projects with longer build durations are less likely to abandon CI, suggesting that projects with more complex builds are better able to adapt the CI service to fit their needs.

To explore whether other CI service providers are attracting users away from CircleCI, we investigate the CI usage of projects that have stopped using CircleCI. For this purpose, we focus on projects that have not executed any CI builds on CircleCI during 2020, the last year of our analysis. We find that 39% (3,074 of 7,795) projects match this criteria. Then, we query the GitHub API to determine if these projects have reported the result of a build from any other CI service during the year 2020 or if they have configured GitHub Actions to execute any CI workflows.

Observation 2*: At least 14% (425 of 3,074) projects that were inactive on CircleCI during the year 2020 have started using another CI service.* We found that projects that became inactive on CircleCI migrated to other CI providers such as Travis CI (50%), GitHub Actions (35%), AppVeyor (16%), Scrutinizer (2%), and SemaphoreCI (2%). The inactive projects on CircleCI had configured at least 20 different CI services to report results back to GitHub. Some of these projects were configured to use more than one CI service, therefore the sum of percentages exceeds 100%.

Figure 3 shows the number of builds on the two different CircleCI platforms during the 2012–2020 time period. The second version of the CircleCI platform[6] provided users with additional

[6]https://circleci.com/blog/say-hello-to-circleci-2-0/



(a) Build Duration

(b) Mean Time to Recovery (MTTR)

(c) Success Rate

(d) Throughput

Figure 4: The evolution of four CI indicators during the period of 2012–2020 in CircleCI. The median value of each metric across the subject systems is shown by the black line. The 90% confidence interval is shown by the gray band.

capabilities like native support for Docker images, flexible resource allocation, customizable images, and SSH access. The release of this updated platform may have contributed to the rapid growth during the 2016–2018 period. The CircleCI 2.0 platform was available as a closed beta in November 2016 and was later made publicly available for all CircleCI consumers in July 2017. Figure 3 shows that these dates coincide with sharp changes in the CircleCI build activity.

Observation 3*: While median project-level indicator values have remained stable, indicators like throughput and build duration have been increasing among the platform's heaviest consumers.* Figure 4 shows the evolution of the four project-level indicators over the studied period. Median values are computed to aggregate individual builds to the level of a project. The black lines show the provider-level medians and gray bands show 90% confidence intervals of each metric across the projects.

Figure 4(a) shows that the median build duration stayed below 200 seconds throughout our data. However, 5% of studied projects took at least 25 minutes to build. Similarly, a study conducted by CircleCI[7] between August 1 and August 30, 2020 found 50% of the studied workflows ran to completion in under four minutes while 5% of the workflows took more than 35 minutes. These large projects might benefit from the CI acceleration techniques proposed in prior research [12].

Figure 4(b) shows a fluctuating median MTTR before December of 2014 because of the low number of observations. However, after this, the median MTTR stayed in the range of 12–87 mins. Similarly, the CircleCI study[7] reported that 50% of the workflows recovered in 55 minutes. For the slowest 5% of projects, the MTTR was at least one week. As shown in prior research [13], these projects could be taking a long time to recover from build failures because the software teams are not taking the CI signal seriously. Only a small proportion of software teams may be focusing on fixing broken builds. For the benefit of software teams who are truly struggling to repair builds, research in automatically repairing build breakage [19, 26], providing developer assistance for build breakage resolution [44], and more broadly automated program repair [24] can be incorporated into CI services to fix build breakages quickly thereby reducing the MTTR.

The large confidence interval in Figure 4(c) shows that the success rate varies widely between projects. Although the median success rate fluctuates before December of 2014, the rate gradually

[7]https://circleci.com/landing-pages/assets/2020-State-of-Software_Metrics-Report_Final.pdf

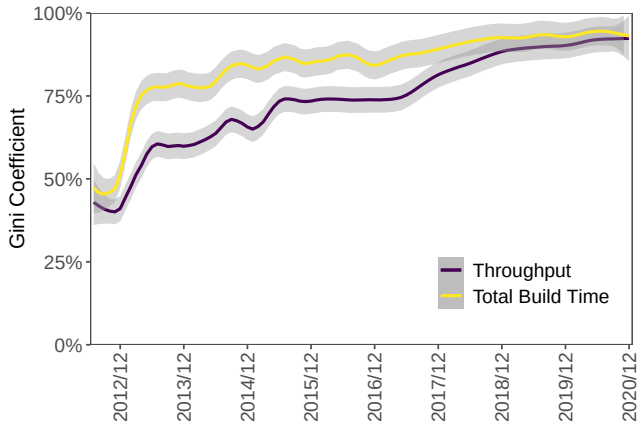


Figure 5: Gini coefficient for throughput and total build time during 2012–2020. Inequality of CI consumers is increasing.

increases later in the studied period and ranges between 67%–92% of all builds. Similarly, the CircleCI study[7] reported that median success rates were 80% for the default branch and 58% for the non-default branch. This demonstrates that, increasingly, in the majority of CI builds, the newly introduced changes are not causing any build failures. If these changes that are not causing any build failures can be identified in advance, the execution of such CI builds can be completely skipped, saving time and compute resources. Abdalkareem *et al.* [1] have proposed a machine learning approach to identify such commits that can be skipped. Jin and Servant [23] have proposed to reduce the high cost of CI by running fewer builds, while running as many failing builds as early as possible. Similarly, Ananthanarayanan *et al.* [3] have proposed to use a probabilistic model, powered by logistic regression to select builds that are most likely to succeed, and speculatively execute them in parallel. These approaches to skip builds can be used by CI providers to prioritize builds that can uncover faults early without wasting computing resources on the growing proportion of passing builds.

Figure 4(d) shows that the median throughput remained under 30 throughout the studied period (roughly one build per day). However, for 5% of the studied projects, the throughput grew rapidly and reached a peak of 900 builds per month. Confirming our observations, the CircleCI study[7] reported that 50% of workflows were invoked fewer than one time per day (0.7), while in the top 5%, workflows were invoked over 35 times per day.

Observation 4*: Inequality of build execution and resource consumption has steadily increased over time.* To further investigate the imbalance of CI usage across users during the studied period, we compute the Gini coefficient [16], a popular measure of inequality. We calculate the Gini coefficient of the throughput and total build time. The Gini coefficient of the throughput estimates inequality in the number of builds being executed, while the Gini coefficient of total build time provides a finer grained perspective. We do not consider projects with no builds for the calculation of Gini index in a given month. Therefore, the reported values should be interpreted as a lower bound on the true inequality.

Figure 5 shows the evolution of the Gini coefficients of throughput and total build time as Loess-smoothed curves with gray shaded areas indicating the 95% confidence interval. Increasingly high Gini coefficients indicate that there is a smaller number of projects that generate a larger proportion of builds and consume a greater proportion of the build time per month. Since these heavy consumers run CI builds frequently, and put a heavy burden on a CI provider's resources, it is important for CI providers like CircleCI to invest in approaches that optimize the workloads of their heaviest consumers.

> *Although the demand for CI has rapidly grown over the years, CI providers have managed to provide a consistent service for the regular consumers. However, to cater to the heaviest consumers who account for a growing proportion of the build activity and resources, CI providers may benefit from research breakthroughs in the areas of build acceleration, and automated build and program repair.*

(RQ2) How is time spent during signal-generating builds?

RQ2: Approach. To answer RQ2, we focus our analysis on signal-generating builds. The CircleCI API response provides start and end times for each step in a given build. Furthermore, each step has an action type which is *machine, infrastructure, checkout, dependencies, compile, test, database, deployment,* or *teardown.* We use the action type and runtime of each build step to compute the percentage of runtime spent executing each action type in a build. Then, we apply the Scott-Knott Effect Size Difference (ESD) test [39] to cluster the action types of build steps into statistically distinct ranks based on the proportion of the build time spent on each action type.

Moreover, we investigate whether the time spent during the signal-generating builds of heavy consumers is different from typical consumers. After identification of heavy consumers, we apply the Mann–Whitney U test [27] and Cliff's Delta [7] statistical instruments. We use those tests because they allow for comparison of the runtime distributions of build steps in heavy and typical consumer categories without an assumption of normality.

In this RQ, note that we are investigating how time is spent performing different actions during builds and not the amount of time spent on each action. Therefore, we focus on relative values instead of absolute values with regard to durations.

RQ2: Results. The top three rows of Table 1 show the distribution of signal-generating builds used to answer RQ2.

Observation 5*: Compiling source code and running tests take up the greatest proportion of the build runtime.* Figure 6 shows the run time percentage for each action in signal-generating builds. The median runtime percentages of the compilation and testing stages are 33.2% and 32.5%, respectively. The next largest action type is downloading dependencies, with a median of 19.5% and is a statistically distinct rank lower than compilation and testing. Focusing efforts to reduce the time taken for compilation and testing stages during the build will provide the most value for CI providers.

An attentive reader may think that the results of RQ2 are influenced by aggregating builds from different projects. For example, based on the results of RQ1, it is likely that Figure 6 is influenced by

Table 1: Distribution of Build Outcome. The global percentage of each category is shown in brackets.

	Outcome	#	%
Signal-generating	Success	18,831,874	80.72%
	Failed	2,806,795	12.03%
	Sub Total	**(21,638,666)**	**92.75%**
Non-signal-generating	Canceled	850,033	3.64%
	Infrastructure fail	10,993	0.05%
	Timedout	17,917	0.08%
	No tests	36,184	0.16%
	Null	776,964	3.33%
	Sub Total	**(1,692,024)**	**7.25%**

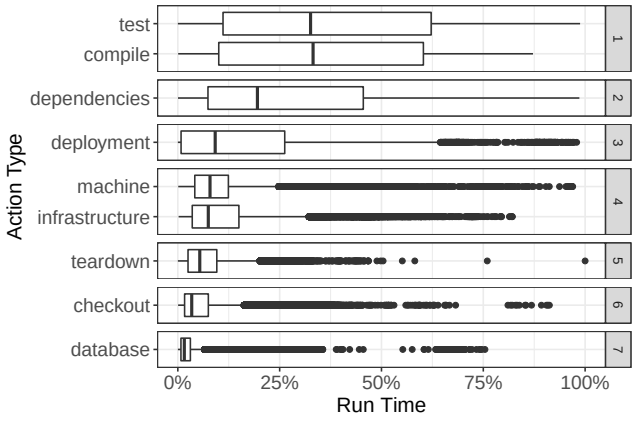


Figure 6: Runtime percentage of each action type in signal-generating builds. The numbered rows indicate the distinct ranks discovered by the Scott-Knott ESD ranker.

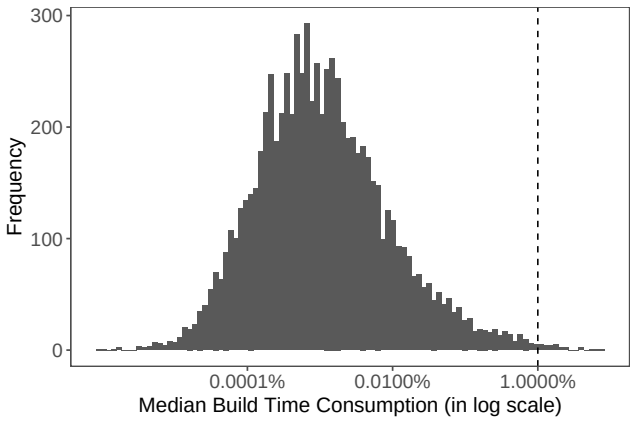


Figure 7: Distribution of median monthly build time consumption. Dashed black line at 1% marks the threshold for selecting heavy CI consumers.

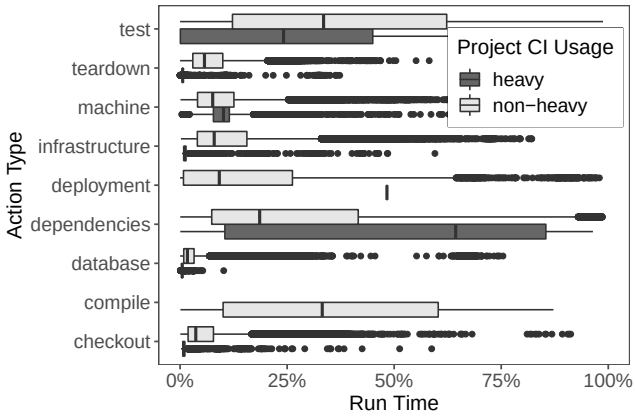


Figure 8: Runtime percentage of each action type in signal-generating builds in heavy CI consumers vs others.

projects that generate many builds/many action types. Because we chose to focus on the overall impact of each action type on the CI service providers, we believe that the influence of aggregate build data is relevant for the service providers.

However, as we observe in RQ1, the heaviest consumers exhibit different behaviour than others. Figure 7 shows the distribution of monthly build time consumption across all studied projects. Guided by this, we select 1% of total monthly build time consumption as the threshold for identifying projects that heavily use CI. Based on this threshold, we identify 27 of 7,795 projects as heavy CI consumers. We conservatively selected this threshold based on the distribution of consumption data which follows a lognormal distribution (Anderson-Darling normality test [38] for log-transformed values, $\alpha = 0.05$). In the log-transformed distribution, our 1% threshold is 5.7 standard deviations away from the mean and therefore projects above the threshold are certainly outliers (typically 2–3 SDs away from the mean).

To check if the selected threshold for identifying heavy CI consumers is suitable, we change the threshold value and see how many projects survive. A more lenient threshold of 0.9% only categorizes five more of the 7,795 projects as heavy consumers. A more strict threshold of 1.1% removes only four more projects from the group of heavy consumers. This indicates that the chosen threshold value will not heavily impact the sample size of heavy CI consumers.

Furthermore, note that the group of heaviest consumers (n=27) identified in this RQ and the 27 projects with highest values for each indicator in RQ1 may not be the same set of projects. While there is considerable overlap between the group of heaviest consumers (n=27) identified in this RQ and the 27 projects with the highest throughput (15 projects = 55%) of RQ1, there is no overlap between the heaviest consumers and the 27 projects with the largest build durations, MTTRs, and success rates of RQ1. We do not believe this is a concern because the largest outliers of these measures do not represent the same concept.

Observation 6*: For all action types except deployment and compile, there were statistically significant differences between heavy user builds and other builds, in terms of the runtime percentage.* Figure 8 compares the runtime percentage of each action type in the signal-generating builds of heavy and typical consumer groups. We apply

the Mann-Whitney U test to the runtime percentage of each action type for the builds of the two user groups. The results show that there are significant differences in the usage patterns and resource consumption of heavy CI consumers compared to typical consumers ($p < 2.2 \times 10^{-16}$) for all action types except *deployment* and *compile*. The *deployment* and *compile* action types had insufficient evidence (i.e., one and zero observations respectively) among the builds of heavy consumers to reach any conclusions. Based on Cliff's delta, effect sizes are large for *infrastructure, checkout, database,* and *teardown* action types. The effect size is medium for the *dependencies* action type, while for the *machine* and *test* action types the effect sizes are small.

As heavy consumers account for an increasing portion of resource consumption, catering to their needs will have a substantial effect on CI providers as well. For example, heavy consumers are spending more time downloading dependencies during the CI process compared to typical consumers. Therefore, if accelerating those builds is a priority, the CI service can allocate additional network bandwidth during the dependency installation of the heavy consumers. Caching the build environment after a CI build is run and reusing that environment in subsequent builds could help to reduce the bandwidth demand, as proposed in prior work [12], because it will eliminate the need to download dependencies for some builds.

> *Approaches to make testing and compiling faster will benefit a large proportion of CI consumers. The heaviest consumers (and CI providers as a consequence) will benefit most from approaches to optimize dependency installation.*

(RQ3) Why are some builds unable to provide a signal?

RQ3: Approach. First, we use an open coding approach on a randomized sample to study the reasons why builds failed to generate a signal. Next, we evaluate the identified set of reasons (i.e., codes) by developing scripts to label example builds automatically. We also manually code a sample of the script-classified builds to evaluate correctness. Finally, we group codes according to common themes. Below, we describe our sampling, discovery, validation, and grouping procedures in more detail.

Sampling. The bottom six rows of Table 1 show the outcomes of 1.69 million non-signal-generating builds. Since coding all of these builds is impractical, we select a random sample for coding.

Our goal is to discover as complete of a set of reasons for why signals could not be generated for builds as possible. Therefore, we strive to achieve *saturation* with our codes. Similar to prior work [22], we code randomly selected non-signal-generating builds until no new codes are discovered for 50 consecutive builds.

We aim to achieve saturation separately for each outcome type (provided by CircleCI) of non-signal-generating builds. We reach saturation after coding 53, 53, 50, 50 and 116 builds for `canceled`, `infrastructure_fail`, `timedout`, `no_tests`, and `NULL` outcome types, respectively.

Discovery. Code discovery was performed by all three authors during remote coding sessions. During the coding sessions, the authors jointly analyzed CircleCI API responses, build logs, and git commits of the sampled non-signal-generating builds to identify the circumstances that led to the particular outcome type.

Validation. To ensure that the identification of the codes is consistent and repeatable, we synthesize our manual coding behaviour into classification scripts. We run these scripts on a sample of 100 non-signal-generating builds. The sample is also manually coded by one author and the inter-rater agreement (Cohen's κ) is computed to measure the agreement between automatic and manual labels.

Grouping. We apply open card sorting to construct a taxonomy of codes to help us to discover latent themes in our detailed coded data. We present these themes to help guide decisions to improve future research and practice in the context of CI services. This card sorting activity was performed collaboratively by all authors.

RQ3: Results. Table 2 provides an overview of the coded reasons for why builds were unable to provide signals. In the validation phase, we observe a perfect inter-rater agreement (Cohen's $\kappa = 1$) between automatic and manual labels, demonstrating a consistent and repeatable coding. Below, we describe the discovered codes in detail according to the four themes that we discovered.

***Observation 7**: At least 371,647 builds failed to generate a signal due to availability issues of the CI service and the supporting services.*

A CI service that is available and reliable is essential to providing prompt feedback to consumers. However, we observe multiple instances where the CI service or the supporting services were unavailable. We observe that 10,993 builds which were unable to provide a signal had an `infrastructure_fail` outcome. According to CircleCI forum discussions,[8] builds can terminate with the `infrastructure_fail` outcome due to faults in CircleCI internal infrastructure or other services that are contacted during the builds (e.g., GitHub, AWS). Although builds that terminate due to infrastructure failures are restarted automatically, it wastes time and resources.

Table 2: Reasons why build signals could not be generated.

Category	#	%
Availability issues	**371,647**	**1.59%**
Orphaned builds	359,857	1.54%
Github unreachable	797	0.00%
Infrastructure fail	10,993	0.05%
Configuration Errors	**31,238**	**0.13%**
Missing or outdated configuration file	30,065	0.13%
Unsupported XCode version	26	0.00%
Github missing CircleCI SSH key	23	0.00%
User Cancellation	**785,877**	**3.37%**
Timed out	**17,917**	**0.08%**
Other	**485,345**	**2.08%**

[8] https://support.circleci.com/hc/en-us/articles/360007792373-There-was-an-issue-while-running-this-container-and-it-was-rerun-The-most-recent-run-is-shown-

Of the builds with a NULL outcome, 359,857 builds were not executed and no explanation was provided. Further investigation revealed that there were no references to these build requests from the GitHub side. Therefore, we code such builds as orphaned builds. An additional 797 builds with NULL outcomes had not started because fetching information from GitHub was unsuccessful. Moreover, incident reports[9] in the CircleCI status dashboard confirmed the occurences of occasional service outages that prevented builds from executing.

To put these rates of failure in context, let us presume that CI providers like CircleCI are striving to achieve industry-standard levels of availability like five nines (99.999%). These availability levels are often achieved by cloud services [31]. In total, at least 371,647 (1.59%) failed to generate a signal due to availability issues of the CI service and the supporting services. This suggests that CircleCI falls a little bit short of this standard (98.41% = 100% - 1.59%), but it is a goal that is not too far out of reach. Since we do not have access to the internals of CircleCI service, we cannot pinpoint the specific architectural shortcomings that lead to availability issues. However, since prior research has identified that investing effort in disaster recovery, fast failure detection, and eliminating single points of failure [29] help to improve availability in cloud services, we recommend CI services in general to direct their attention to such best practices.

***Observation 8**: The vast majority of cancelled builds (92%) were cancelled by user request.* We observe that 50% of the non-signal-generating builds (849,954 of 1,692,024) were abruptly terminated with a build outcome of `cancelled`. Our coding reveals two reasons for cancellation; user requests or automatically by CircleCI if the build is determined to be redundant. If the *Auto-cancel redundant builds* feature is enabled, CircleCI cancels any queued or running builds when a newer build is triggered on that same branch. We find that only 64,077 of the cancelled builds were cancelled automatically. The vast majority (785,877) were cancelled by user request. Since no further reason is provided, future user studies may help to better understand why so many builds are being cancelled by consumers.

***Observation 9**: 17,917 builds timed out before completion.* If there is no output from any of the commands during build execution for ten minutes, CircleCI terminates the build. We observe that builds time out at all stages of the CI process (e.g., setting up the environment, installing dependencies, testing). Currently, consumers can extend the timeout in the CI configuration if they expect a build step to continue longer than ten minutes without output. However, programmatically determining if a program will terminate is a form of undecidable problem known as the halting problem [40]. Therefore, using only the source code changes to be built, CI service cannot determine if a build will eventually terminate if given more time or will hang forever. Although termination analysis research has proposed automatic tools to determine whether some builds will eventually terminate, implementing such solutions in a CI service to support a multitude of programming languages and build tools is infeasible. Instead, CI services can use recent past build outcomes to speculate that a build will time out, similar to how SmartBuildSkip [23] operates based on the hypothesis that failing builds in CI happen consecutively after another build failure.

***Observation 10**: At least 31,238 builds failed to generate a signal due to misconfigurations.* There are three codes that are associated with misconfigurations that lead to signals not being generated. At least 31,238 builds with NULL outcomes did not complete because the CI process was not properly configured (e.g., 30,065 builds with missing or outdated CircleCI configuration files, 26 builds specifying an unsupported Xcode version, 23 builds where GitHub was missing a CircleCI SSH key). Adoption of tools such as CI-Odor [43] and Hansel & Gretel [14] proposed in prior research to identify and fix configuration errors may reduce future occurrences of the builds terminating due to misconfiguration.

Furthermore, some configuration issues are resolved over time by the service provider. For example, in platform version 1.0 of CircleCI, if a command was not defined for the testing phase of the CI process, the build terminated with the special outcome of `No tests`. To prevent this behaviour from interfering with their workflow, CircleCI consumers had to include an explicit no-op command in the testing phase of their CI configuration. We found that 36,184 builds terminated with a build outcome of `No tests`. In version 2.0 of the CircleCI platform, this requirement was relaxed such that a test section was not required for build job execution.

> *Availability issues, configuration errors, user cancellation, and exceeding time limits are key reasons that lead to non-signal-generating builds. Approaches to increase the availability and improve the robustness of CI configuration will likely yield the largest reductions in non-signal-generating builds.*

5 THREATS TO VALIDITY

This section describes the threats to the validity of our study.

5.1 Construct Validity

We use the mapping of commands and action types provided by CircleCI API to determine the CI stage for each command and then compute the time distribution for each signal-generating build. The accuracy of this mapping depends on CircleCI's labelling. To mitigate this threat, we manually inspected a sample of 50 commands and their assigned action types for consistency.

Some CI consumers may have configured CI builds that finish quickly and return a successful build outcome in seconds without executing any useful tests. In such situations, metrics such as build duration and success rate will not provide value as indicators of the effectiveness of CI. Therefore, we do not promote these metrics as goals for software teams. Instead, we use them as indicators of service usage from the CI providers' perspective.

5.2 Internal Validity

We manually analyze API responses, build logs, and source code changes to characterize reasons that cause builds to terminate without providing a signal. To discover as complete of a set of reasons as is possible, we set out to achieve saturation in our samples of manually analyzed examples of non-signal-generating builds. Although we set what we believe to be a conservative saturation criterion (50

[9] https://status.circleci.com/incidents/kxw3dyhykqkb

consecutively labelled examples without discovering a new label), our approach is not exhaustive. There may be other reasons that caused the builds to abruptly terminate, which have not yet been uncovered. Nonetheless, we were able to discover a set of reasons that can be (a) linked to actionable implications for CI service providers; and (b) automatically discovered, enabling further characterization and the assessment of mitigation strategies.

5.3 External Validity

Although we studied a large sample of projects (7,795) and builds (23.3 million) spanning a long period (eight years), our study focuses on the context of a single CI provider. As a consequence of the choice of the research method, we aim to provide deeper insights on a single case and does not aim at statistical generalizability. On the other hand, modern CI providers share a similar configuration interface (simple YAML-based DSLs) and largely overlapping feature sets (e.g., social coding platform integration, container-based orchestration, job-based parallelism support for projects that target multiple execution platforms, and multiple programming languages). Therefore, we suspect that similar conclusions would be drawn from other service providers. Nevertheless, replication studies in other contexts might prove fruitful.

6 RELATED WORK

In this section, we situate our work in the context of the literature on the analysis of CI data, the challenges associated with CI, and the service provider perspective in other DevOps contexts.

6.1 Analysis of CI Datasets

The analysis of large collections of historical CI data is not uncommon in the literature. For example, the Travis CI service has been the target of several research efforts. Beller et al.'s *TravisTorrent* [6] provides a curated set of data about 2.6 million builds from the Travis CI service. Their initial analysis indicates that testing is the single dominant reason for builds to fail [5]. Durieux et al. [8] have curated an even larger set of 35 million Travis CI jobs. Using independent collections of Travis CI data, Hilton et al. [21] studied how developers use CI and Rausch et al. [33] performed a targeted analysis of build failures in the context of Java open source systems. Similarly, Zhang et al. [50] have studied 6.9 million CI builds to identify the ten most common compiler error types. Complementing the work on build categorizations based on failures, in our work, we categorize the build outcome based on whether a signal was provided to the consumer at the end of the build.

Prior work has demonstrated that these corpora, made of large collections of operational data, are not free of noise. Gallaba et al. [13] found that, on average, there is at least one build with an inaccurate outcome in every eleven builds that they analyze. Zolfagharinia et al. [52] described the build inflation problem in an analysis of 30 million builds from the Perl ecosystem. Felidré et al. [11] identified four characteristics of projects that use CI technology without adopting CI principles. Since our focus is to provide the CI provider's perspective of the overall usage of the CI service, we do not remove noisy builds from our analysis.

A line of work on anti-patterns has emerged from the studies of CI data. Gallaba et al. [14] formulate four anti-patterns that impact CI specifications, and propose Hansel & Gretel to detect and repair them. Vassallo et al. [43] identified four additional anti-patterns that hinder the benefits associated with CI, and propose CI-Odor to detect them. Zampetti et al. [49] studied Q&A forums to identify 79 CI smells. Since slow build durations hinder the pace of development, Ghaleb et al. [15] characterize builds with long durations. While these prior works adopt the CI consumers' perspective, we focus on the CI providers' perspective to facilitate all users amidst the presence of anti-patterns.

Collections of build data have also been used to study the links between the adoption of CI and other project characteristics. Vasilescu et al. [41, 42] observe an increase in developer-reported bugs after the introduction of CI, suggesting that CI helps developers discover more defects. Zhao et al. [51] report that the introduction of CI technology is associated with a higher rate of successful pull requests; however, pull requests tend to take longer to arrive at an outcome. Instead of focusing on individual project characteristics, we focus on the overall state of the CI service over time.

6.2 Challenges in CI

Researchers have also studied the challenges faced when software teams are adopting CI. By conducting a qualitative study, Hilton et al. [20] found that when using CI, developers face trade-offs between speed and certainty, better access and security, and more configuration options and ease of use. Pinto et al. [32] surveyed 158 CI users about the benefits and problems of CI systems and found that developers are unsure about what constitutes a successful build, due to reasons such as flaky tests or misconfigured CI jobs. Widder et al. [47], identify information overload, organizational pain points, configuration, slow feedback, and testing deficiencies as the main CI pain points. Using the CI provider's perspective, our work confirms that slow feedback due to long-running builds and configuration issues causing non-signal-generating builds are common in practice, strengthening the conclusions from prior work.

6.3 DevOps Service Providers' Perspective

There have been several studies on the design and operation of large-scale DevOps services. Schermann and Leitner [36] propose using genetic algorithms for scheduling experiments when continuous deployment is used by software organizations. Similarly, Günalp et al. [17] present Rondo, a tool suite for continuous deployment of service-oriented applications, which aims for a deterministic and idempotent deployment process.

Going beyond research experiments, practitioners also report on state-of-the-art DevOps platforms developed at large-scale software organizations. Esfahani et al. [10] describe how Microsoft's internal distributed build service was designed to speed up the CI workflow of their existing projects. Gupta et al. [18] report on how a large-scale online experimentation platform was designed at Microsoft to provide scalable and trustworthy results for internal users in their controlled experiments. An experience report by Savor et al. [35] presents observations from following the continuous deployment process of cloud-based software at Facebook and OANDA. Similarly, Rossi et al. [34] describe how continuous deployment is practiced during mobile software development at Facebook.

While prior work focuses on designing and maintaining DevOps infrastructure for internal users, exposing the service to external users may also present unique challenges. We therefore look into the challenges in providing CI services for external users.

7 CONCLUSIONS & LESSONS LEARNED

Through an empirical study of 23.3 million builds from the popular CircleCI service, we set out to better understand the challenges that CI service providers face and the opportunities that are present. We make ten observations (see Section 4) from which we conclude that prior research innovations are well-suited to address current limitations in CI services:

The rapidly growing build throughput and build durations of heavy consumers (Observation 3) suggest that build acceleration approaches are needed to stem this rising tide. Large software organizations like Microsoft and Google have produced internal solutions to accelerate builds [10]; however, their adoption in other settings presents challenges. Approaches like Kotinos [12] strive to make accelerated builds accessible to other organizations. The high rate of successful builds suggests that CI providers have a growing pool of candidate builds with which to apply techniques proposed in the literature to skip builds. For example, candidate approaches [1] aim to save resources and time by skipping the execution of builds that are unlikely to fail. Prior work [2] estimates that 18% of successful builds could be skipped, equating to 3.38 million builds in our setting — leading to considerable savings for a provider like CircleCI. Furthermore, CI providers can consider offering suggestions to developers to fix build errors by using automated repair techniques like BuildMedic [26], HireBuild [19], and DeepDelta [28], thereby reducing the MTTR. Vassallo et al. [45] have shown that suggesting public solutions to build breakages, which can be found online, reduces the time to fix breakages by an average of 37%.

> ***Hypothesis:*** *Tools and techniques for accelerating builds will help manage the rapid growth in build throughput and build durations.*

Focusing optimization effort on compilation and testing stages will likely provide the most benefit. Most of the build time is spent in the compilation and testing stages for a large proportion of service consumers (Observation 5). Therefore, approaches to optimize compilation and testing steps effectively reducing their run time or skipping such steps altogether, are well suited to drive down service costs and improve throughput. For example, research in the Facebook context has shown that using one such strategy, predictive test selection [25], can reduce the total infrastructure cost of testing code changes by a factor of two. Yet there are challenges that make adopting such approaches difficult in the (global) context of a CI provider. For example, the plethora of build tools, language toolchains, and testing frameworks makes tool- and language-specific approaches [1, 17, 19, 48] unlikely to yield optimal results. Any provider-side solution will need to operate in the heterogeneous deployment environment in which CI services operate.

> ***Hypothesis:*** *Language-agnostic solutions in CI services to optimize compilation and testing stages will be most beneficial to reduce build durations.*

Providing more transparency regarding orphaned builds may improve the user experience. Orphaned builds constitute a large proportion of the builds that are affected by availability issues (Observation 7). These builds do not provide any details about the internal failures that caused the disruption of the service and are not linked from GitHub. To users, orphaned builds are entirely opaque service failures, which may impact their perception of the service provider. Therefore, providing detailed reporting and traceability for orphaned builds will improve the user experience, helping to retain and attract consumers for the CI service.

> ***Hypothesis:*** *Providing visibility about orphaned builds will improve customer retention and growth.*

Service providers can use recent past build outcomes to identify builds that are likely to timeout. We find 17,917 builds timed out without generating a signal (Observation 9). Timed out builds take the full allocated time reserved to perform a build and do not provide a change status signal. Therefore, although these builds account for a small percentage of all builds, from the perspective of both CI providers and consumers, timed out builds are a waste of the maximum amount of resources and time. However, due to its similarity to the halting problem, identifying builds that will time out is not trivial. Instead, similar to SmartBuildSkip [23], CI service providers can use heuristics (e.g., many time-out builds in CI are followed by more consecutive time-out builds.) to skip builds that are likely to time-out, saving resources.

> ***Hypothesis:*** *Heuristics such as frequent time-out builds in the recent past can be used to identify and skip builds that are likely to time out.*

User research is needed to better understand why builds are being cancelled by users. Since one of the most common reasons for abruptly terminating CI builds is cancellation by the user (Observation 8), future research is needed to characterize this behaviour. For example, while cancellation may indicate that a build was unintentionally triggered (e.g., publishing a PR before it was ready), there may be other cases that more careful user experience engineering could help to mitigate.

> ***Hypothesis:*** *User research will help to understand reasons for user-cancelled builds.*

In order to aid in future replication of our results, we make our data and scripts publicly available online.[10]

Acknowledgements. 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 software products.

[10] https://doi.org/10.6084/m9.figshare.16569630

REFERENCES

[1] R. Abdalkareem, S. Mujahid, and E. Shihab. A machine learning approach to improve the detection of CI skip commits. *IEEE Transactions on Software Engineering (TSE)*, pages 2740–2754, 2020.

[2] R. Abdalkareem, S. Mujahid, E. Shihab, and J. Rilling. Which commits can be CI skipped? *IEEE Transactions on Software Engineering (TSE)*, pages 448–463, 2019.

[3] S. Ananthanarayanan, M. S. Ardekani, D. Haenikel, B. Varadarajan, S. Soriano, D. Patel, and A.-R. Adl-Tabatabai. Keeping master green at scale. In *Proc. of EuroSys Conference*. ACM, 2019.

[4] C. AtLee, L. Blakk, J. O'Duinn, and A. Z. Gasparnian. Firefox release engineering. In A. Brown and G. Wilson, editors, *The Architecture of Open Source Applications: Structure, Scale, and a Few More Fearless Hacks*, chapter 2. Creative Commons, 2012.

[5] M. Beller, G. Gousios, and A. Zaidman. Oops, my tests broke the build: An explorative analysis of travis CI with GitHub. In *Proc. of International Conference on Mining Software Repositories (MSR)*, pages 356–367, 2017.

[6] M. Beller, G. Gousios, and A. Zaidman. TravisTorrent: Synthesizing travis CI and GitHub for full-stack research on continuous integration. In *Proc. of International Conference on Mining Software Repositories (MSR)*, pages 447–450, 2017.

[7] N. Cliff. Dominance statistics: Ordinal analyses to answer ordinal questions. *Psychological Bulletin*, 114(3):494–509, 1993.

[8] T. Durieux, R. Abreu, M. Monperrus, T. F. Bissyandé, and L. Cruz. An analysis of 35+ million jobs of Travis CI. In *Proc. of International Conference on Software Maintenance and Evolution (ICSME)*, pages 291–295, 2019.

[9] P. M. Duvall, S. Matyas, and A. Glover. *Continuous Integration: Improving Software Quality and Reducing Risk*. Pearson Education, 2007.

[10] H. Esfahani, J. Fietz, Q. Ke, A. Kolomiets, E. Lan, E. Mavrinac, W. Schulte, N. Sanches, and S. Kandula. CloudBuild: Microsoft's distributed and caching build service. In *Proc. of International Conference on Software Engineering Companion (ICSE-C)*, pages 11–20, 2016.

[11] W. Felidré, L. Furtado, D. A. da Costa, B. Cartaxo, and G. Pinto. Continuous integration theater. In *Proc. of International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–10, 2019.

[12] K. Gallaba, Y. Junqueira, J. Ewart, and S. Mcintosh. Accelerating continuous integration by caching environments and inferring dependencies. *IEEE Transactions on Software Engineering (TSE)*, 2020.

[13] K. Gallaba, C. Macho, M. Pinzger, and S. McIntosh. Noise and heterogeneity in historical build data: an empirical study of Travis CI. In *Proc. of International Conference on Automated Software Engineering (ASE)*, pages 87–97, 2018.

[14] K. Gallaba and S. McIntosh. Use and misuse of continuous integration features: An empirical study of projects that (mis)use Travis CI. *IEEE Transactions on Software Engineering (TSE)*, 46(1):33–50, 2020.

[15] T. A. Ghaleb, D. A. da Costa, and Y. Zou. An empirical study of the long duration of continuous integration builds. *Empirical Software Engineering (EMSE)*, 24(4):2102–2139, 2019.

[16] C. Gini. On the measure of concentration with special reference to income and statistics. *Colorado College Publication, General Series*, 208(1):73–79, 1936.

[17] O. Günalp, C. Escoffier, and P. Lalanda. Rondo: A tool suite for continuous deployment in dynamic environments. In *Proc. of International Conference on Services Computing (SCC)*, 2015.

[18] S. Gupta, L. Ulanova, S. Bhardwaj, P. Dmitriev, P. Raff, and A. Fabijan. The anatomy of a large-scale experimentation platform. In *Proc. of International Conference on Software Architecture (ICSA)*, 2018.

[19] F. Hassan and X. Wang. HireBuild: An Automatic Approach to History-Driven Repair of Build Scripts. In *Proc. of International Conference on Software Engineering (ICSE)*, pages 1078–1089, 2018.

[20] M. Hilton, N. Nelson, T. Tunnell, D. Marinov, and D. Dig. Trade-offs in continuous integration: assurance, security, and flexibility. In *Proc. of Joint Meeting on European Software Engineering Conference and Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 197–207, 2017.

[21] M. Hilton, T. Tunnell, K. Huang, D. Marinov, and D. Dig. Usage, costs, and benefits of continuous integration in open-source projects. In *Proc. of International Conference on Automated Software Engineering (ASE)*, pages 426–437, 2016.

[22] T. Hirao, S. McIntosh, A. Ihara, and K. Matsumoto. The review linkage graph for code review analytics: a recovery approach and empirical study. In *Proc. of Joint Meeting on European Software Engineering Conference and Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 578–589, 2019.

[23] X. Jin and F. Servant. A cost-efficient approach to building in continuous integration. In *Proc. of International Conference on Software Engineering (ICSE)*, pages 13–25, 2020.

[24] C. Le Goues, M. Pradel, and A. Roychoudhury. Automated program repair. *Communications of the ACM*, 62(12):56–65, Nov. 2019.

[25] M. Machalica, A. Samylkin, M. Porth, and S. Chandra. Predictive test selection. In *Proc. of International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 91–100, 2019.

[26] C. Macho, S. McIntosh, and M. Pinzger. Automatically repairing dependency-related build breakage. In *Proc. of International Conference on Software Analysis, Evolution, Reengineering (SANER)*, pages 106–117, 2018.

[27] H. B. Mann and D. R. Whitney. On a test of whether one of two random variables is stochastically larger than the other. *The Annals of Mathematical Statistics*, pages 50–60, 1947.

[28] A. Mesbah, A. Rice, E. Johnston, N. Glorioso, and E. Aftandilian. Deepdelta: Learning to repair compilation errors. In *Proc. of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 925–936, 2019.

[29] M. R. Mesbahi, A. M. Rahmani, and M. Hosseinzadeh. Reliability and high availability in cloud computing environments: a reference roadmap. *Human-centric Computing and Information Sciences*, 8(1), 2018.

[30] A. N. Meyer, L. E. Barton, G. C. Murphy, T. Zimmermann, and T. Fritz. The work life of developers: Activities, switches and perceived productivity. *IEEE Transactions on Software Engineering (TSE)*, 43(12):1158–1193, 2017.

[31] J. Pavlik, V. Sobeslav, and A. Komarek. Measurement of cloud computing services availability. In *Proc. of the International Conference on Nature of Computation and Communication*, pages 191–201, 2015.

[32] G. Pinto, F. Castor, R. Bonifacio, and M. Rebouças. Work practices and challenges in continuous integration: A survey with Travis CI users. *Software: Practice and Experience*, 48(12):2223–2236, 2018.

[33] T. Rausch, W. Hummer, P. Leitner, and S. Schulte. An empirical analysis of build failures in the continuous integration workflows of java-based open-source software. In *Proc. of International Conference on Mining Software Repositories (MSR)*, pages 345–355, 2017.

[34] C. Rossi, E. Shibley, S. Su, K. Beck, T. Savor, and M. Stumm. Continuous deployment of mobile software at Facebook (showcase). In *Proc. of International Symposium on Foundations of Software Engineering (FSE)*, pages 12–23, 2016.

[35] T. Savor, M. Douglas, M. Gentili, L. Williams, K. Beck, and M. Stumm. Continuous deployment at Facebook and OANDA. In *Proc. of International Conference on Software Engineering Companion (ICSE-C)*, pages 21–30, 2016.

[36] G. Schermann and P. Leitner. Search-based scheduling of experiments in continuous deployment. In *Proc. of International Conference on Software Maintenance and Evolution (ICSME)*, pages 485–495, 2018.

[37] H. Seo, C. Sadowski, S. Elbaum, E. Aftandilian, and R. Bowdidge. Programmers' build errors: a case study (at Google). In *Proc. of International Conference on Software Engineering (ICSE)*, pages 724–734, 2014.

[38] M. A. Stephens. Edf statistics for goodness of fit and some comparisons. *Journal of the American statistical Association*, 69(347):730–737, 1974.

[39] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto. An empirical comparison of model validation techniques for defect prediction models. *IEEE Transactions on Software Engineering (TSE)*, 43(1):1–18, 2017.

[40] A. M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proc. of the London mathematical society*, 2(1):230–265, 1937.

[41] B. Vasilescu, S. van Schuylenburg, J. Wulms, A. Serebrenik, and M. G. J. van den Brand. Continuous integration in a social-coding world: Empirical evidence from github. **updated version with corrections**. *CoRR*, abs/1512.01862, 2015.

[42] B. Vasilescu, Y. Yu, H. Wang, P. Devanbu, and V. Filkov. Quality and productivity outcomes relating to continuous integration in GitHub. In *Proc. of Joint Meeting on European Software Engineering Conference and Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 805–816, 2015.

[43] C. Vassallo, S. Proksch, H. C. Gall, and M. Di Penta. Automated reporting of anti-patterns and decay in continuous integration. In *Proc. of International Conference on Software Engineering (ICSE)*, pages 105–115, 2019.

[44] C. Vassallo, S. Proksch, T. Zemp, and H. C. Gall. Un-break my build: Assisting developers with build repair hints. In *Proc. of International Conference on Program Comprehension (ICPC)*, pages 41–51, 2018.

[45] C. Vassallo, S. Proksch, T. Zemp, and H. C. Gall. Every build you break: developer-oriented assistance for build failure resolution. *Empirical Software Engineering (EMSE)*, 25(3):2218–2257, 2019.

[46] D. G. Widder, M. Hilton, C. Kästner, and B. Vasilescu. I'm leaving you, Travis: a continuous integration breakup story. In *Proc. of International Conference on Mining Software Repositories (MSR)*, pages 165–169, 2018.

[47] D. G. Widder, M. Hilton, C. Kästner, and B. Vasilescu. A conceptual replication of continuous integration pain points in the context of Travis CI. In *Proc. of Joint Meeting on European Software Engineering Conference and Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 647–658, 2019.

[48] X. Xie, L. Ma, F. Juefei-Xu, M. Xue, H. Chen, Y. Liu, J. Zhao, B. Li, J. Yin, and S. See. Deephunter: A coverage-guided fuzz testing framework for deep neural networks. In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 146–157, 2019.

[49] F. Zampetti, C. Vassallo, S. Panichella, G. Canfora, H. Gall, and M. D. Penta. An empirical characterization of bad practices in continuous integration. *Empirical Software Engineering (EMSE)*, 25(2):1095–1135, 2020.

[50] C. Zhang, B. Chen, L. Chen, X. Peng, and W. Zhao. A large-scale empirical study of compiler errors in continuous integration. In *Proc. of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 176–187, 2019.

[51] Y. Zhao, A. Serebrenik, Y. Zhou, V. Filkov, and B. Vasilescu. The impact of continuous integration on other software development practices: A large-scale empirical study. In *Proc. of International Conference on Automated Software Engineering (ASE)*, pages 60–71, 2017.

[52] M. Zolfagharinia, B. Adams, and Y.-G. Guéhéneuc. A study of build inflation in 30 million CPAN builds on 13 Perl versions and 10 operating systems. *Empirical Software Engineering (EMSE)*, 24(6):3933–3971, 2019.