Accelerating Continuous Integration by Caching Environments and Inferring Dependencies

Keheliya Gallaba, John Ewart, Yves Junqueira, and Shane McIntosh, *Member, IEEE*

Abstract—To facilitate the rapid release cadence of modern software (on the order of weeks, days, or even hours), software development organizations invest in practices like Continuous Integration (CI), where each change submitted by developers is built (e.g., compiled, tested, linted) to detect problematic changes early. A fast and efficient build process is crucial to provide timely CI feedback to developers. If CI feedback is too slow, developers may switch contexts to other tasks, which is known to be a costly operation for knowledge workers. Thus, minimizing the build execution time for CI services is an important task.
While recent work has made several important advances in the acceleration of CI builds, optimizations often depend upon explicitly defined build dependency graphs (e.g., make, Gradle, CloudBuild, Bazel). These hand-maintained graphs may be (a) underspecified, leading to incorrect build behaviour; or (b) overspecified, leading to missed acceleration opportunities. In this paper, we propose KOTINOS—a language-agnostic approach to infer data from which build acceleration decisions can be made without relying upon build specifications. After inferring this data, our approach accelerates CI builds by caching the build environment and skipping unaffected build steps. KOTINOS is at the core of a commercial CI service with a growing customer base. To evaluate KOTINOS, we mine 14,364 historical CI build records spanning three proprietary and seven open-source software projects. We find that: (1) at least 87.9% of the builds activate at least one KOTINOS acceleration; and (2) 74% of accelerated builds achieve a speed-up of two-fold with respect to their non-accelerated counterparts. Moreover, (3) the benefits of KOTINOS can also be replicated in open source software systems; and (4) KOTINOS imposes minimal resource overhead (i.e., $<$ 1% median CPU usage, 2 MB – 2.2 GB median memory usage, and 0.4 GB – 5.2 GB median storage overhead) and does not compromise build outcomes. Our results suggest that migration to KOTINOS yields substantial benefits with minimal investment of effort (e.g., no migration of build systems is necessary).

Index Terms—Automated Builds, Build Systems, Continuous Integration

1 INTRODUCTION

Continuous Integration (CI) [10] is the practice of automatically compiling and testing changes as they appear in the version control system (VCS) of a software project. CI is intended to provide quick feedback to developers about whether their changes will smoothly integrate with other changes that team members have submitted. Unlike scheduled (e.g., nightly) builds, CI feedback is received while design decisions and tradeoffs are still fresh in the minds of developers.

With the adoption of CI, software organizations strive to increase developer productivity [41] and improve software quality [31]. Open source [5] and proprietary [12, 35] software organizations have invested in adopting CI. Cloud-based CI services such as CIRCLECI have become popular, since they provide the benefits of CI without the burden of provisioning and maintaining CI infrastructure.

Using a suboptimally configured CI service can slow feedback down and waste computational resources [16, 42, 49]. Indeed, Widder et al. [45] found that developers often complained about slow feedback caused by builds that take too long as a pain point in CI.

Several build tools have been proposed to reduce build duration by executing incremental builds. Google's Bazel, Facebook's Buck, and Microsoft's internal CloudBuild [12] service are prominent examples from large software companies. While these solutions make important contributions, in our estimation, they have two key limitations. First, the build acceleration features rely upon a graph of build dependencies that is specified by developers in build configuration files (e.g., Bazel BUILD files). These manually specified build dependency graphs may drift out of sync with the other system artifacts [25, 27, 46]. Indeed, build dependency graphs may be *overspecified* [28, 40], leading to acceleration behaviour that is suboptimal (i.e., an unnecessary dependency forces potentially parallelizable steps to be executed sequentially), or worse, *underspecified* [6], leading to acceleration behaviour that fails non-deterministically (i.e., a missing dependency may or may not be respected depending on whether or not the acceleration service decides to execute the dependent steps sequentially or in parallel).

Second, the accelerated build tools are designed to replace existing build tools, increasing the barrier to entry. For example, a team that has invested a large amount of effort in designing a build system with an existing tool may be reluctant to migrate their build code to a new language.

To address these limitations, we propose KOTINOS—a build acceleration approach for CI services that disentangles build acceleration from the underlying build tool. KOTINOS accelerates CI by inferring dependencies between build

- *K. Gallaba is with the Department of Electrical and Computer Engineering, McGill University, Canada.*
  E-mail: keheliya.gallaba@mail.mcgill.ca
- *J. Ewart and Y. Junqueira are with YourBase Inc., USA.*
  E-mail: john@yourbase.io, yves@yourbase.io
- *S. McIntosh is with the Cheriton School of Computer Science, University of Waterloo, Canada.*
  E-mail: shane.mcintosh@uwaterloo.ca

Manuscript received date; revised date.

steps. Rather than parsing build configuration files, KOTINOS infers the build dependency graph by tracing system calls and testing operations that are invoked during the execution of an initial *(cold)* build. This inferred dependency graph is then used to reason about and accelerate future *(warm)* CI builds. First, the environment setup is cached for reuse in future builds. Second, by traversing the inferred dependency graph, we identify build steps or tests that can be safely skipped because they are not impacted by the change under scrutiny. Since the KOTINOS approach is agnostic of the programming languages and build tools being used, KOTINOS can yield benefits for teams without requiring considerable build migration effort. Currently, KOTINOS is at the core of a CI service[1] with a growing customer base.

We evaluate KOTINOS by mining 14,364 historical CI build records spanning ten software projects (three proprietary and seven open source) and nine programming languages. Our evaluation focuses on assessing the frequency of activated accelerations, the savings gained by these accelerations, and the computational cost of KOTINOS, and is structured along the following three research questions:

RQ1: How often are accelerations activated in practice?
Motivation: To determine whether an acceleration can be applied to a build, KOTINOS checks what files have changed and how those files impact the previously inferred dependency graph. Therefore, it is important to know how frequently these opportunities for acceleration occur in practice. If such opportunities for acceleration are rare, adopting KOTINOS might not be worthwhile. Therefore, we set out to study how often each type of acceleration (i.e., environment cache, step skipping) is activated in sequences of real-world commits.
Results: We find that in practice, at least 87.9% of builds activate at least one KOTINOS acceleration type. Among the accelerated builds, 100% leverage the build environment cache, while 94% skip unnecessary build steps.

RQ2: How much time do the proposed accelerations save?
Motivation: The primary goal of KOTINOS is to reduce build duration. Therefore, we set out to measure the improvements to build duration that KOTINOS provides.
Results: By mining the CI records of the studied proprietary systems, we observe that, build duration reductions in accelerated builds are statistically significant (Wilcoxon signed rank test, $p < 0.05$; large Cliff's delta, i.e., > 0.474). Moreover, the accelerated builds achieve a clear speed-up of at least two-fold in 74% of the studied builds. By replaying past builds of the studied open source systems, we observe that build durations can be reduced in five out of seven open source subject systems.

RQ3: What are the costs of the proposed accelerations?
Motivation: Build acceleration approaches often increase computational overhead or hinder build correctness. Therefore, it is important to quantify the costs of KOTINOS in terms of resource utilization and correctness.
Results: We observe that KOTINOS can accelerate builds with minimal CPU (median < 1%), memory (median 53 MB), and storage (median 5.2 GB) overhead. Furthermore, 100% of the open source builds that we repeated in the KOTINOS environment report the same build outcome (pass, fail) as the currently adopted CIRCLECI service, suggesting that KOTINOS outcomes are sound.

[1]https://yourbase.io/

Table 1: The duration of build steps in a proprietary system.

Build Step	Duration
Environment Initialization	**1m 14s**
Provision OS	42s
Setup DB Services (MySQL & Redis)	22s
Install Ruby	7s
Install Node.JS	3s
Dependency Installation	**5m 5s**
apt-get update	7s
apt-get install	1m 48s
gem install bundler	8s
bundle install	3m 37s
npm install	25s
Database Population	**1m 23s**
rake db:create	11s
rake db:setup	9s
rake db:migrate	1m3s
Test Execution	**1hr 23m 17s**
npm run test:javascript	4s
rake spec	1hr 23m 13s
Total	**1hr 31m 6s**

2 MOTIVATING EXAMPLE

To demonstrate the reasons for long durations in a typical build, we use the build log of a proprietary software project. The log include diagnostic information about the execution of all jobs belonging to a build. We first classify each command in the build log according to its build phase, which includes: (1) environment initialization; (2) dependency installation; (3) database population; and (4) test execution. Second, for each command, we use its timestamp to estimate the execution duration of the command.

Table 1 shows the durations of these commands and phases. A non-negligible proportion (8.5%) of the time is spent on preparatory steps for build execution (i.e., environment initialization, dependency installation, and database population); however, the vast majority of time (91.5%) is spent (re-)executing tests. This suggest that substantial build acceleration may be achieved by skipping the re-execution of unnecessary tests and by reusing previously prepared build environments.

Table 1 also shows that the build process of this project uses multiple tools (i.e., *apt*, *bundler, npm,* and *rake*) and runs tests written in multiple languages (i.e, *JavaScript/Node.js* and *Ruby*). This suggests that tool-specific acceleration solutions are unlikely to achieve optimal results.

It is observations like these that inform our design of KOTINOS—a programming language and build tool-agnostic approach to accelerate CI builds. KOTINOS addresses the challenge of environmental reuse by building and leveraging a cache of previously established build environment images. Moreover, KOTINOS addresses the challenge of excessive re-execution of test steps (but more broadly, build steps) by reasoning about build dependencies using an inferred, system-level dependency graph.

3 Related Work

The focus of our work is accelerating CI builds. Therefore, we begin by describing the background with respect to challenges associated with CI in general, slow CI builds, and proposed techniques for optimizing slow CI builds.

3.1 Challenges in Continuous Integration

CI provides several bene ts to software teams that adopt it. For example, Vasilescu et al. [41] observed that the adoption of CI coincides with improvements in the productivity of software teams. Practitioners use CI because it helps them to catch bugs early and release software more often [21].

On the other hand, CI also introduces many challenges in the software development process [32, 33]. For example, Hilton et al. [20] observed that practitioners face problems, such as increased complexity, increased time costs, and new security concerns when working with CI. Prior studies have also found that CI speci cations [16] and processes [42, 49] are susceptible to anti-patterns [8] that impact their maintainability, performance, and security.

To tackle these CI adoption and maintenance problems, the research community has provided tools that improve the transparency and maintainability of the CI pipeline. Recent research on identifying reasons for build breakage in CI is one such area [15, 44]. Techniques for automatically xing build breakages have also been proposed [19, 26, 43]. Moreover, tools like Hansel & Gretel [16] and CI-Odor [42] suggest xes for common anti-patterns in CI pipelines.

While the prior work is helpful, the speed at which CI feedback is provided is a primary objective of CI [31]. Below, we discuss the prior work on long CI build durations.

3.2 Slow CI Feedback

In a recent literature survey, Widder et al. [45] summarize multiple studies about the implications of slow CI builds. According to the diverse populations that were studied, slow CI builds is one of the key barriers to adoption of CI for software teams. The latency introduced by slow CI builds can delay pull request assessments [48, 50], hindering the premise of rapid CI feedback [34]. Developers also complain about the cost of computational resources and the dif culty of debugging software with a slow CI cycle [20, 23].

Felidré et al. [14] studied the CI build durations of 1,270 open source projects and found that 16% of the projects have build durations that exceed the 10-minute rule of thumb for which past literature [7, 20] has advocated. Furthermore, 78% of the participants in the study by Hilton et al. [20] stated that they actively allocate resources to reduce the duration of their CI builds. This further illustrates the practical importance of making improvements to CI build speed.

3.3 Accelerating Slow CI Builds

Due to its importance to practitioners, there have been several recent approaches proposed to tackle slow CI builds. Ghaleb et al. [17] studied the reasons behind long build durations, observing that caching content that rarely changes is a cost-effective way of speeding up builds. Cao et al. [9] use a timing-annotated build dependency graph to forecast build duration. Tufano et al. [39] propose an approach to alert developers about the impact that code changes may have on future CI build speeds. While these techniques help developers to cope with slow CI builds, we propose a set of approaches to automatically accelerate CI builds.

In prior work, several approaches have been proposed to reduce the time taken by CI builds. Abdalkareem et al. [1, 2] suggest to skip CI altogether for commits that do not affect source code. However, skipping quality checks and avoiding testing altogether when they were necessary (i.e., false positives) can lead to botched releases. Esfahani et al. [12] describe the CloudBuild distributed build service, which uses content-based caching to save time and compute resources at Microsoft. Li et al. [24] propose test case selection [36] during CI by using static dependencies and dynamic execution rules. Many other approaches have been proposed to reduce test execution time by minimization, selection, and prioritization (e.g., [11, 29, 47]).

The past work highlights the effectiveness of CI build acceleration solutions that skip build steps based on a shared cache of build outputs, as well as the selection of tests that are impacted by a software change. However, a key limitation of prior approaches is a reliance upon developer annotation and/or (largely) manually speci ed build con guration les. The manual speci cation of build dependencies is error-prone [6, 25, 27, 46]. Moreover, since organizations may already have non-trivial build speci cations that were written for existing build tools, migrating to a new build tool requires a large investment of effort [18, 37]. Therefore, in this paper, we strive to accelerate builds without relying on explicitly speci ed build dependencies. To simplify the adoption of our approach, we leverage existing CI pipeline speci cations where available. Broadly speaking, we strive to deliver a language-agnostic solution so that existing code bases can immediately bene t from our approach with minimal investment of migration effort.

4 The Kotinos Approach

A CI build is comprised of jobs, each executing an isolated set of tasks. A typical approach is to have one job for each targeted variant of the programming language toolchain or runtime environment of the project. Once the CI service receives a build request, jobs are created based on the CI con guration le. These jobs are placed into a queue of pending jobs. When job processing nodes become available, they execute jobs from this queue. In this paper, we focus on reducing the duration of the job processing phase.

We propose two acceleration techniques. Listings 1 and 2 provide a running example of the CI con guration of the Wallaby project (https://github.com/reinteractive/wallaby). Listing 1 shows the original steps speci ed for the Travis CI service. We migrated this CI con guration to the format of Kotinos (Listing 2), which simpli es parsing and enables acceleration features; however, the existing build system (Rake and Bundler in this case) remains unmodi ed.

4.1 Caching of the Build Environment (L1)

Before invoking the commands speci ed in a project's CI con guration, build job processing nodes need to be initialized and prepared. First, a programming language

Listing 1: TRAVIS CI Configuration

```
language: ruby
cache: bundler
node_js: '10.5.1'
rvm:
  - 2.6.0
gemfile:
  - gemfiles/Gemfile.rails-5.0

env:
  global:
    - DB=postgresql
    - RAILS_ENV=test

addons:
  postgresql: "9.6"

before_install:
  - gem install bundler

install:
  - bundle install

before_script:
  - psql -c 'CREATE DATABASE dummy_test;' -U postgres

script:
  - bundle exec rake db:setup
  - bundle exec rake db:migrate
  - bundle exec rake spec
```

Listing 2: KOTINOS Configuration

```
1  dependencies:
2    build:
3      - ruby: 2.6.0
4      - node: 10.15.1
5
6  build_targets:
7    - commands:
8        - apt-get update
9        - apt-get install -y postgresql-client libpq-dev
10       - gem install bundler
11       - bundle install
12       - bundle exec rake db:setup
13       - bundle exec rake db:migrate
14       - bundle exec rake spec
15
16     name: daily_ci
17     container:
18       image: yourbase/yb_ubuntu:16.04
19
20     environment:
21       - DB=postgresql
22       - BUNDLE_GEMFILE=gemfiles/Gemfile.rails-5.0
23       - PGUSER=ci
24       - PGPASSWORD=ci
25       - RAILS_ENV=test
```

Figure 1: An example of commits in chronological order. In Commit A, all source code files are added. In Commit B, the README.md file is modified. In commits C and D, source code files that can affect multiple tests are modified.

runtime and basic toolchain need to be installed within an execution environment. Next, the libraries and services (e.g., databases, message brokers, browsers) that are required to build the project are also installed. Since systems rarely migrate from one programming language to another and their dependencies often reach a stable point, we conjecture that these steps for preparing the build processing environment are rarely changed over the lifetime of a project. Repeatedly installing the same runtime and downloading the same dependencies at the start of each build wastes time.

We propose to reuse the environment across builds. We implement this behaviour in KOTINOS by caching a Docker container that is created during the first passing (cold) build, and reusing that image during subsequent (warm) builds. If the environment changes in the later builds (e.g., due to updates in dependency versions), the environment cache will be invalidated and the cold build procedure will be re-executed (i.e., a fresh image will be created and stored).

To illustrate these concepts, consider the series of commits from the Wallaby project (https://github.com/reinteractive/wallaby) that are shown in Figure 1. When the first commit (Commit A) is built, KOTINOS executes all of the initialization and preparation steps because no prior build for the project has been executed. First, an Ubuntu 16.04 build image must be provisioned (line 18 of Listing 2). Next, Ruby and Node.js language runtimes must be installed (lines 3–4). Then, environment variables must be set to specified values (lines 21–25). Finally, the build commands (lines 8–14) can be executed. After the build finishes, a Docker image that encapsulates the initialization and preparation steps is saved to the cache to be reused in subsequent builds. The procedure for Docker image encapsulation is described below.

When later warm builds (for Commits B, C, and D) are requested, KOTINOS checks its cache for build images of ancestors in the version history of the project and selects the most recently created image. This reuse of images avoids repeating installation and preparation steps.

Environment Caching Details. Every request to initiate a CI build is accompanied by build metadata: (1) a reference to a build image; (2) the URL of the source code repository; and (3) the unique identifier (e.g., SHA) of the commit to be built. KOTINOS uses this metadata to look-up previous builds executed for this repository.

If there are no previous builds for the specified repository, a cold build is initiated. Based on the user-specified container build image (e.g., ubuntu:latest) a pre-built image that is hosted in an internal Docker registry is downloaded and a container is created based on this image. This container is used for running the remainder of the build.

Using the source code repository URL and the commit ID, the revision of the code to be built is downloaded within the container. Then, the build steps (e.g., compiling, running tests), which are specified in the configuration file, are executed within the container. Once the build process finishes, if it was successful, the state of the container is saved, and is stored in the environment cache along with the repository name and the commit ID. This image is later used for subsequent warm builds of the same repository, saving the set up time needed before every build. If a cold build fails, the container is retained for debugging purposes, but is not used for accelerating subsequent builds.

4.2 Skipping of Unaffected Build Steps (L2)

Changes for which CI builds are triggered often modify a small subset of the files in a repository. If build steps that a change does not impact can be pinpointed, those steps could be safely skipped. These sorts of incremental builds that only re-execute impacted commands have been at the core of build systems for decades [13]. Typically, build tools create and traverse a Directed Acyclic Graph (DAG) of dependencies to make decisions about which build steps are safe to skip. These DAGs are explicitly specified by developers in tool-specific DSLs (e.g., make files).

To implement step skipping in a tool-agnostic manner at the level of the CI provider, we first collect traces of system calls that are made during build execution. We then mine these system call traces to understand the processes that are created, file I/O operations, and network calls associated with each build step. This information is then used to construct a dependency graph. Later, we traverse this dependency graph to identify skippable steps.

For example, during Commit A of Figure 1, the dependency graph is inferred based on the system call trace log. Later, when the build for Commit B is requested, KOTINOS checks whether files modified in Commit B are part of the dependency graph. In this case, README.md is not part of the dependency graph. Therefore, all build steps (line 8–14) are skipped. This effectively skips an entire CI build like the approach proposed by Abdalkareem et al. [1, 2] without relying on heuristics. On the other hand, in Commits C and D, the modified source code files are part of the dependency graph, and therefore KOTINOS decides to run the tests (line 14: `bundle exec rake spec`).

By default, each command that is specified in the configuration file can be skipped separately. However, users can choose to skip subprocesses at a finer granularity by wrapping invocations using the `skipper` command. If users require an even finer granularity for skipping, KOTINOS provides test-level skipping via plug-ins for popular testing frameworks like RSpec (Ruby) and JUnit (Java).

Below, we define the inferred dependency graph, and the approaches we use to construct, update, and traverse it.

The inferred dependency graph is a directed graph $BDG = (T; D)$ where: (1) nodes represent targets $T = T_f \cup T_s$, T_f is the set of files produced or consumed by the build, T_s is the set of build commands, and $T_f \cap T_s = \emptyset$; and (2) directed edges denote dependencies $d(t; t') \in D$ from target t to target t' of three forms: (a) $read(t_f; t_s)$, i.e., file t_f is read by command t_s; (b) $write(t_s; t_f)$, i.e., command t_s writes file t_f; and (c) $parent(t_{s_1}; t_{s_2})$, i.e., command t_{s_1} is the parent process of command t_{s_2}.

Inferring the Build Dependency Graph. The first build of a project is a cold build As the first step during the cold build, KOTINOS creates a fresh container based on a user-specified container image. Then, the build steps specified by the user in the configuration file are executed inside the freshly-created container. Another process monitors all the system calls being executed during each build step. This monitoring process records: (1) the path of files being read and written; (2) the commands being invoked; (3) the Process ID (PID) of started processes; and (4) the PID of the parents of started processes. Prior to constructing the BDG, we filter the system call traces to remove files that do not appear in the VCS. After the cold build is completed, the BDG is stored for later use in subsequent warm builds.

Updating the Inferred dependency graph. To make correct decisions during the acceleration, the BDG must reflect the state of project dependencies that is relevant to commit C_t. Since C_t represents a unique state of source code at a given time t, we can say that the initial build B_0 is derived from C_0 (i.e., $B_0 \leftarrow f(C_0)$). The BDG inferred from a commit C_t is denoted as $G_t \leftarrow ObservationOf(B_t)$.

After build B_0 completes, there exists a BDG (G_0) that contains the inferred dependencies for that build. Since KOTINOS acceleration strives to skip unnecessary build steps, the BDG from a warm build will be incomplete. We solve this by implementing a pairwise BDG update operation $!(G_{n-1}; G_n)$, which updates the previous BDG with the current observed behaviour of the build.

To ensure that graph G_n is updated to include new dependencies from B_n we:

1) Clone the graph from the parent build (G_{n-1}).
2) Process the recorded observations from the incremental build (B_n) and create a partial dependency graph (G'_n)
3) For each step recorded in G'_n:
   a) Look for the identical step in G_n.
   b) If found, prune its dependencies replacing them with the newly identified nodes using existing nodes as needed (e.g., if another step shares a dependency).
   c) If no step is found, then this step is new and is added to the existing dependency graph in the correct location, also linking to existing nodes where applicable.
   d) Remove the step from G'_n.
4) Store the merged graph, G_n as a new graph so that it can be referenced for subsequent warm builds.

For example, consider a build process that consists of three steps, `npm install`, `npm lint`, and `npm test` These commands will download the dependencies, statically analyze the source code for errors (i.e., linting), and then run the tests. At the first commit, all the build steps will be executed and G_0 will be generated. In the second commit, if a test file is modified, KOTINOS will only run the `npm test` step and generate a partial graph G'_1. If new files are read during this run, they will be recorded as dependencies in G'_1. Then, the subgraph of G_0 that is affected by the `npm test` step will be replaced by G'_1 and this modified G_0 will be saved as G_1.

To avoid incorrect build behaviour, BDG-based acceleration is bypassed for commits that will likely modify the structure of the BDG (e.g., those that add or rename files).

In cases where KOTINOS determines that it cannot confidently make a decision about applying acceleration, it will revert to cold build behaviour to guarantee an accurate build output and dependency graph. For example, in the case where new files are added to a project, prior BDGs are considered invalid and a cold build is performed.

Traversing the Build Dependency Graph. For each build request, KOTINOS first derives the set of files being changed in the changeset to be built (`ChangedFilesList`). KOTINOS determines which build steps should be run, given the `ChangedFilesList` and the inferred build dependency graph (BDG). We first compute the impacted steps using the union of the transitive closures within the BDG for each file

Figure 2: An example of how the Build Dependency Graph is used to identify which steps to skip. S1 and S2 cannot be skipped because they are directly and transitively dependent on the changed le F1, respectively. However, S3 can be skipped because it does not depend on any changed les.

f in ChangedFilesList . For each build step, we check whether it is in the set of impacted steps. If it is, we must re-execute the step, and if not, the step can be safely skipped.

Figure 2 provides an example of a BDG and highlights the behaviour when one le (le F1) is modi ed by a changeset being built. The example illustrates how KOTINOS handles the three types of scenarios that may occur for a step within the graph. We describe each scenario below:

Direct dependency (S1). Step S1 or one of its subprocesses reads from le F1. Therefore, S1 cannot be skipped.

Transitive dependency (S2). Step S1 reads from F1 and writes to le F2. The step S2 reads from F2. Therefore, S2 transitively depends on F1 and cannot be skipped.

No dependency (S3). Step S3 only depends on the le F3, which was not modi ed by the change set being built. Since step S3 does not have a direct or transitive dependency on modi ed les, step S3 can be safely skipped.

5 RQ1: HOW OFTEN ARE ACCELERATIONS ACTIVATED IN PRACTICE?

In this section, we address RQ1 by studying the frequency at which KOTINOS accelerations are activated. We rst introduce the subject systems, then describe our approach to addressing RQ1, and nally, present our observations.

Subject Systems. The top three rows of Table 2 provides an overview of the three proprietary systems that we use to evaluate RQ1. We study a sample of 13,864 passing builds from September 1st, 2019 to December 31st, 2019.

These three subject systems are sampled from the pool of projects that actively use KOTINOS as their primary CI service. We selected these systems for analysis because they are implemented using a variety of programming languages and frameworks, and use a variety of build and test tools.

Approach. During each build execution, KOTINOS prints detailed diagnostic logging messages to an internal datastore. Each KOTINOS build includes a diagnostic log in that datastore. Messages in those diagnostic logs indicate when an acceleration is activated (among other things). To answer RQ1, we analyze the logs of all passing builds in our studied timeframe to determine which levels of acceleration were activated during each build execution. If the log mentions that the build was performed within a container that was based on a previously cached image, that build is labelled as accelerated by caching. If the log mentions that at least one of the build steps was skipped, that build is labelled as accelerated by skipping build steps.

Table 2: Overview of the subject systems.

Project ID	Application Domain	Programming Languages	LOC	# Passing Builds	Non-accelerated Build Duration (Median)
Commercial A	Fintech	Android, Dart, Go, Ruby, Node.js	455,470	5,202	18min 33s
Commercial B	Blockchain	Python, Node.js	208,768	7,273	2min 40s
Commercial C	E-commerce	Ruby, Node.js	1,218,980	1,389	1hr 7min 33s
apicurio-studio	Development Tools	Java, TypeScript, HTML, CSS	84,446	100	8min 28s
forecastie	Weather	Java, Python, HTML, CSS	10,012	96	2min 19s
gradle-gosu-plugin	Development Tools	Groovy, Java	3,773	82	3mins
Robot-Scouter	Robotics	Kotlin	1,442,304	27	16min 36s
aerogear-android-push	Development Tools	Java	2,280	23	1min 34s
magarena	Entertainment	Groovy, Java	165,199	26	1hr 18min 22s
cruise-control	Development Tools	Java, Python	61,426	71	36min 17s

Observation 1: At least 87.9% of the builds in the studied systems are accelerated. Table 3 shows the percentages of studied builds that activate the different types of acceleration. We nd that 87.9%, 98.9%, and 97.6% of the studied builds are accelerated by at least one type of acceleration in the A, B, and C systems, respectively. This indicates that KOTINOS can frequently accelerate CI builds. We delve into why system A has a lower rate of acceleration activation below.

Observation 2: Environment caching is the most commonly activated strategy. Indeed, 87.9%–97.6% of builds leverage the environment cache. In system A, all of the builds that are accelerated by environment caching also skip steps. In systems B and C, 9.2% and 24.8% of the accelerated builds only use the environment cache, respectively.

We follow an iterative process to identify why 548 builds did not use the environment cache. First, we select and inspect a random sample of 30 logs from the builds that missed acceleration. The inspection reveals the root cause for missing acceleration. For each root cause, we implement a detection script to automatically identify occurrences in the other logs. We repeat this sampling, inspection, and scripting process until root causes for all 548 builds. In 2% (twelve of 548) of the cache-missing builds, the user explicitly overrode KOTINOS' decision-making, forcing the cache to be ignored (via a build request parameter). Users may decide to override caching in multiple scenarios. For example, if users want to invalidate an external dependency, ignoring the caching will force a fresh copy of external dependencies to be downloaded. Moreover, if users expect to encounter inconsistencies in the generated dependency graph, they may override KOTINOS' decision-making to reset with a fresh build. This often occurs when new les are added to (a dynamically generated area of) the build graph. Users who choose to override KOTINOS' decision-making are prioritizing the correctness of the build over the speed of feedback—a common trade-off in build systems [4].

Another 0.7% (four of 548) missed the cache because KOTINOS had purged the cache and had started with a fresh container. This purging behaviour triggers when users reach the organizational limit of 100 cached image layers. However, the majority of the cache-missing builds (97%) occurred because KOTINOS was unable to nd a cached image of a predecessor for the commit that is being built.

Observation 3: Although less frequently activated than environ-

Table 3: The frequency of activated build accelerations.

Project ID	# Studied Builds	% Builds Accelerated by Caching	% Builds Accelerated by Skipping Steps	% Builds Accelerated by Any Strategy
A	5,202	87.9	87.9	87.9
B	7,273	98.9	89.7	98.9
C	1,389	97.6	72.8	97.6

Figure 3: Distribution of durations in accelerated and non-accelerated builds across the three subject systems.

ment caching, step skipping accelerations are activated regularly as well. Table 3 shows that 87.9%, 89.7%, and 72.8% of the studied builds skip at least one build step in systems A, B, and C, respectively. This shows that it is relatively rare for the changed files in a commit to impact all of the build steps. Even though steps are being skipped the majority of the time, when they are not skipped, builds are still often accelerated by leveraging the environment cache.

> In practice, a majority of builds (at least 87.9%) activate at least one of the KOTINOS accelerations.

6 RQ2: How much time do the proposed accelerations save?

In this section, we address RQ2 by studying the change in build durations of accelerated and non-accelerated builds.

6.1 Overall Statistical Analysis

First, we conduct a statistical analysis to measure the effect of KOTINOS acceleration on the build durations of the three proprietary subject systems from Section 5. Below, we describe our approach, present our results, and discuss the limitations of such a statistical analysis.

Approach. We extract build durations by using the build_start_time and build_end_time fields in our dataset. Then, we apply Wilcoxon signed rank tests (unpaired, two-tailed, $\alpha = 0.05$) and compute the Cliff's delta to check whether our acceleration strategies reduce build durations to statistically and practically significant degrees.

Observation 4: KOTINOS achieves large, statistically significant reductions to build durations. Figure 3 shows the distributions of build durations. Based on the p-values after applying Holm-Bonferroni correction [22], the null hypothesis that there is no significant difference among the distributions of accelerated and non-accelerated build durations could be rejected in all three subject systems. Moreover, for all three subject systems, the Cliff's Delta values are large (i.e., > 0.58), indicating that the difference between non-accelerated and accelerated builds is practically significant.

Limitations. Although this analysis provides an overview of the benefit of KOTINOS accelerations, the observations may be impacted by (at least) two confounding factors:

1) Differences in jobs. Past builds may have targeted different (groups of) jobs. For example, while most builds of Project A target the entire software system, a subset of past builds only target the backend.
2) Changes in CI configuration over time. Throughout a project's history, build steps may be added or removed from the CI configuration.

Due to the above reasons, build durations from the same project can vary and may not be directly comparable.

6.2 Longitudinal Analysis

To mitigate the limitations of the overall statistical analysis, we conduct a longitudinal analysis. In a nutshell, the approach clusters related builds into streams. Builds within a stream can be more meaningfully compared to one another.

Approach. Since it is unsafe to compare jobs with different definitions, we first group builds according to their unique job names. Next, within each job grouping, we further categorize builds by the set of build steps that are being executed. For this purpose, we extract the set of build steps by parsing each build log for the diagnostic messages that print the set of build steps that were performed. It is important to note that the order of build steps is preserved by this extraction step. We then feed this list of steps into a hash function (i.e., Python hash) to compute a fingerprint for the set of build steps that is easy to compare.

By comparing the hash fingerprints across all of the builds in our sample, we find that there are 25 streams of unique build steps within the build jobs of the three subject systems. Since the hash comparison can be too strict, we manually inspect the 25 sets of build steps for opportunities to merge streams that only differ in minor ways. We find four streams that share similar commands and are unlikely to differ substantially in terms of build duration. These streams were not grouped together because of the overly strict matching of the hashing function. The only differences between these streams are: (1) minor version changes in external dependencies; (2) adding an external dependency that does not affect build steps; and (3) renaming a build script. After merging these minor changes into their respective streams, we are left with 22 streams of builds with durations that may be compared within the streams.

For every warm build b_w, we find the cold build b_c to which it should be compared by searching backwards within the stream to which b_w belongs. The build b_c is the most recent preceding cold build of b_w in the version control

history. Finally, we plot the build duration of warm builds in each stream in comparison to the corresponding cold build.

Observation 5: The vast majority of warm builds are faster than cold ones. Figure 4 shows how the warm build duration changes over the studied period as a percentage of cold build duration in the main job of each subject system. Due to space constraints, we only include the line plot for each project's main job. The plots for all jobs are available in the online appendix. [2] Each line segment with the same shade of gray shows the period in which the project was using the same set of build steps. The black circles show when cold builds were triggered. The red line marks the build duration of the most recent preceding cold build of each warm build. Therefore, gray lines appearing below the red line in Figure 4 illustrate when warm builds are faster than their cold counterparts. The lines with same shade of gray in these figures also show that the build steps are not changed throughout the studied period for the projects A and B.

For project C, some warm builds took longer than cold builds during the first half of the studied time period, as shown by the light gray line segment in Figure 4c. This was due to the overhead of an experimental feature of KOTINOS, which was enabled only for project C. Then, after a change to the build steps that disabled the experimental feature in mid-November, the warm builds appear consistently below the cold builds, as shown in the dark gray segment.

Figure 5 shows the distribution of warm build durations as a percentage of cold builds within the same stream. The solid red line indicates the cold build duration. The builds in red-shaded areas are faster than their cold counterparts. In the best case (Project A), 99.9% of warm builds outperform cold builds. Even in the worst case (Project C), 86.4% of warm builds outperform cold builds. Overall, 93% of warm builds outperform their cold counterparts.

Observation 6: Acceleration often yields substantial build speed improvements. In Figure 5, the builds in dark red-shaded area (below the dashed red line) complete within 50% of the duration of comparable cold builds. Indeed, 99.1%, 53.2%, and 74.5% of the studied warm builds complete within 50% of their cold counterparts in the A, B, and C systems, respectively. Overall, 74% of the studied warm builds complete within 50% of the build duration of similar cold builds.

6.3 Replay Analysis

The historical build records from the studied proprietary systems provide a concrete perspective on build savings, but it is not possible to analyze the impact of each acceleration technique. To enable such an analysis, we expand our study to include subject systems from the open source community.

Approach. We select a sample of seven repositories that use CIRCLECI (a market leader in cloud-native CI [3]). Using the CIRCLECI API, we select the seven systems with the longest median build duration from the set of systems with passing builds in between January and July 2020. The bottom seven rows of Table 2 provide an overview of the subject systems.

We extract the most recent sequence of commits from the master branch of each repository, along with their CI configuration, in the reverse chronological order, until a limit of 100 commits is reached or the CI configuration is modified to the extent that builds start failing. We collect 425 commits across the seven subject systems for further analysis. Then, we migrate the most recent CI configuration file of each subject system to KOTINOS' configuration format. To replay builds following the sequence of development, we build each commit in the order in which they appeared on the master branch (oldest to newest). For each commit, we perform three types of builds: (1) without KOTINOS accelerations; (2) with environment caching enabled (i.e., L1); and (3) with both caching and step-skipping enabled (i.e., L1+L2). To mitigate the impact that fluctuations in the workload on our experimental machines may have in our observations, we repeat each build variant ten times. To check whether the acceleration levels differ in terms of build duration to a statistically significant degree, we rank the build durations of each commit in each acceleration level using the Scott-Knott ESD test [38]. Finally, we compute the likelihood of each acceleration type appearing in the lowest (i.e., fastest) rank across the seven subject systems.

Observation 7: In a majority of open source subject systems, accelerated builds are faster than non-accelerated builds. Figure 6 shows the results of the replay experiment on the seven open source subject systems. The lines indicate the median build performance across ten repetitions, while the error bars indicate the 95% confidence interval. In five of the seven subject systems (i.e., aerogear-android-push, apicurio-studio, forecastie, gradle-gosu-plugin, and Robot-Scouter), all commits except the first one are built faster when accelerations are enabled. The first build of each subject system is slow because an environment cache does not yet exist and the inferred graph needs to be constructed. In the other two subject systems (i.e., cruise-control and magarena), accelerations rarely reduce build duration considerably. Inspection of the source code reveals that all test groups in these systems are invoked by a single process. This resulted in an "all or nothing" re-execution of tests. If code that impacts just one test was changed, all tests would be re-executed. To enable skipping at the finer granularity of individual tests, we plan to implement a language-specific extension capable of decomposing large test groups in future work.

Figure 7 shows the likelihood of each acceleration approach appearing in the top rank of the Scott-Knott ESD test. The builds without acceleration rarely appear in the top rank (median 1%), where as L1 and L1+L2 acceleration levels achieve top-rank performance much more frequently (medians of 47% and 52%, respectively).

> Accelerated builds are statistically significantly faster than non-accelerated builds with large effect sizes. Moreover, the builds accelerated by KOTINOS achieve a clear speed-up of at least two-fold in 74% cases in practice. The benefits of KOTINOS can also be replicated in open source systems that practice process-level test invocation.

[2] https://doi.org/10.6084/m9.figshare.12106845
[3] https://www2.circleci.com/forrester-wave-leader-2019.html

(a) Project A (b) Project B (c) Project C

Figure 4: Warm build duration as a percentage of cold build duration in each project's main job. The red line marks the cold build duration. The black circles indicate when cold builds were triggered. Projects A and B uses only one CI con guration. Project C has modi ed the CI con guration in mid-November as shown by the different shades of gray.

Figure 5: The gray-shaded violin plots show the kernel probability density of warm build durations as a percentage of cold builds across the three subject systems. The solid red line indicates the cold build duration (baseline). The builds in red-shaded area (below the solid red line) are faster than their cold counterparts. The builds in dark red-shaded area complete within 50% of their cold counterparts.

Table 4: CPU and memory usage of KOTINOS during the builds of seven open source systems.

Project ID	Process Type	Normalized CPU Usage (%)		Memory Usage (MB)	
		Median	Max	Median	Max
apicurio	System Call Monitoring	0.30	5.20	53	54
	Graph Creation/Update	0.42	4.32	231	348
	Graph Traversal	0.05	0.10	64	90
forecastie	System Call Monitoring	0.70	1.10	53	54
	Graph Creation/Update	0.64	1.05	129	130
	Graph Traversal	0.01	0.05	2	29
gradle-gosu	System Call Monitoring	0.15	0.90	53	156
	Graph Creation/Update	0.20	0.57	35	182
	Graph Traversal	0.04	0.05	8	9
Robot-Scouter	System Call Monitoring	1.26	2.77	53	105
	Graph Creation/Update	0.11	14.51	2,241	2,592
	Graph Traversal	0.04	6.20	22	62
aerogear	System Call Monitoring	0.50	0.87	53	88
	Graph Creation/Update	0.45	1.04	49	153
	Graph Traversal	0.01	0.04	6	7
magarena	System Call Monitoring	0.02	2.45	53	159
	Graph Creation/Update	0.08	13.34	693	720
	Graph Traversal	0.05	0.06	41	50
cruise-control	System Call Monitoring	0.01	0.54	53	54
	Graph Creation/Update	0.08	1.11	149	190
	Graph Traversal	0.04	0.05	13	20

7 RQ3: WHAT ARE THE COSTS OF THE PROPOSED ACCELERATIONS?

Build speed is often a trade-off with other non-functional build requirements, e.g., computational footprint [12] and build correctness [3]. In prior sections, we quanti ed the bene ts of K OTINOS. In this section, we set out to quantify the costs in terms of resource utilization and correctness.

7.1 Resource Utilization

As KOTINOS relies on cached information and trace logs of build execution, memory, CPU, and storage are consumed in exchange for the performance improvement. Thus, when addressing RQ3, we rst measure the resource overhead.

Approach. We study the seven open source systems from Section 6. We measure memory and CPU usage by collecting process-level metrics from each VM in which the seven systems are built during the execution of three K OTINOS processes: (1) system call monitoring; (2) graph creation/update; and (3) graph traversal. To compute storage usage, we rst download the source code of each subject system and the language toolchain that is required to build that source code into a Docker container. Next, all the build steps are executed without using K OTINOS. The size of the image at this point is recorded as the baseline. Then, each studied commit is built using K OTINOS, after which, the size of the image is computed. The storage overhead of KOTINOS is computed as the difference in the sizes of a post-build KOTINOS image and the post-build baseline image.

Observation 8: KOTINOS does not consume resources heavily. Table 4 shows the CPU and memory usage of the three KOTINOS components. Five of the seven subject systems have minimal CPU (median < 1%) and memory usage (median 2 MB – 231 MB) during their builds. Yet the Robot-Scouterand magarenasystems have greater CPU and memory usage (median 693 MB – 2.2 GB). The logs indicate that these two subject systems made vefold as many system calls as other systems, leading to a larger memory footprint when parsing system calls and inferring the BDG.

Figure 6: Median build time for each acceleration level in the open source subjects. Black vertical bars indicate the 95% con dence interval. (Acceleration Levels: L1 = Caching of the build environment, L2 = Skipping of unaffected build steps).

Figure 7: The likelihood of each acceleration technique appearing in the top rank. Circles indicate the median, while the error bars indicate the 95% con dence interval.

Table 5: Storage overhead of KOTINOS build images.

Project ID	Baseline Image Size (GB)	Median Post-build Image Size (GB)	Effective Post-build Image Size (GB)
apicurio-studio	3.12	49.08	45.96
forecastie	3.45	4.94	1.49
gradle-gosu-plugin	1.53	3.03	1.50
robot-scouter	4.19	7.74	3.55
aerogear-android-push	3.59	3.99	0.40
magarena	0.57	5.81	5.23
cruise-control	1.64	5.87	4.23

Table 5 shows the median image size created by KOTINOS builds. In six of the seven subject systems, KOTINOS introduced between 0.4 GB and 5.23 GB of storage overhead. In the extreme case of apicurio-studio system, median image size was 45.96 GB. By inspecting the builds of this system further, we identi ed that Maven generated a JAR le that included all the transitive dependencies of the nal deliverable. This caused the image to grow in size during each subsequent build. Even in this extreme case, the cost of storing a build image for a month will only be US$1 (50 GB $0.020 per GB) based on cloud storage prices offered by popular service providers. By exploiting the layered le system of Docker, the size of the image on disk is further reduced, effectively minimizing storage costs.

7.2 Correctness

Although build steps are being skipped, it is important that build outcomes are preserved. Therefore, we set out to compare the outcomes provided by KOTINOS and CIRCLECI (i.e., a traditional CI service) for a common set of commits.

Approach. We select 500 commits of the seven studied open source systems, irrespective of the original build outcome in their CIRCLECI builds. To mitigate the risk of non-deterministic (i.e., " aky") build outcomes, we check that the outcomes of two cold builds are identical for each studied commit. We remove 34 commits because the build outcome was inconsistent. We then build the remaining 466 commits with KOTINOS acceleration and compare the outcome with the corresponding CIRCLECI builds.

Observation 9: 100% of the KOTINOS build outcomes are consistent with CIRCLECI builds. All 425 builds that passed originally, resulted in passing KOTINOS builds. The 41 builds that failed in CIRCLECI also failed in KOTINOS.

> KOTINOS can accelerate builds with minimal resource overhead and without compromising build correctness.

8 IMPLICATIONS

KOTINOS saves time and computational resources by accelerating CI builds. By identifying which steps in the CI process can be safely skipped, KOTINOS helps software teams to reduce CI build duration. Since the accelerated CI results are available within minutes, developers will be able to stay focused on their tasks, avoiding costly context switches [30, 51]. Moreover, since unnecessary build steps will not be re-executed, KOTINOS also helps organizations to reduce the computational footprint of their CI pipelines.

Language-agnostic build accelerations allow systems written in various languages using heterogeneous build chains to bene t. As long as a software project has a build script that speci es a series of steps for converting source code into software deliverables, KOTINOS can infer its graph and accelerate future builds. Irrespective of the language runtime or tools that are used in each step, the KOTINOS approach should apply. This allows teams to use the programming languages and build tools that they are comfortable with while still bene ting from modern CI acceleration.

Software teams can immediately bene t by migrating to KOTINOS with minimal disruptions. The accelerations provided by KOTINOS are available to projects without requiring changes to source code or build system speci cations. The users only need to introduce a high-level con guration le (e.g., Listing 2), which invokes steps in existing build les. This approach of not modifying existing project artifacts means that teams are able to immediately derive bene ts from migration to KOTINOS without a substantial initial investment (e.g., migration of build tools [18, 37]) and with minimal disruptions to development activities.

9 THREATS TO VALIDITY

This section describes the threats to the validity of our case study-based evaluations in Sections 5 and 6.

Internal Validity. Threats to internal validity are concerned with (uncontrolled) confounding factors that may offer plausible alternative explanations for the results that we observe. It is possible that factors other than the studied ones may be slowing CI builds down. This work aims to tackle two major causes of slow builds and is not intended to be exhaustive. In our future work, we plan to identify more causes and to add additional acceleration types to KOTINOS.

External Validity. Threats to external validity refer to limitations to the generalizability of our observations to examples outside of our study setting. We analyze and demonstrate our approach on three proprietary and seven open source subject systems. As such, our results may not generalize to all software systems. However, the subject systems that we analyze, use nine different programming languages and nine build tools. By evaluating KOTINOS using these subject systems, we show that the language-agnostic nature of KOTINOS can bene t systems implemented using a broad variety of languages and build tools.

10 CONCLUSION AND FUTURE WORK

A main goal of practicing CI in software teams is to provide quick feedback to developers. While existing build acceleration tools have made important advances, in our estimation, they suffer from two key limitations: (1) reliance upon explicitly speci ed dependencies in build con guration les; and (2) the barrier to entry for adopting a new build tool.

To overcome these limitations, we propose KOTINOS—an approach to build acceleration that is language- and tool-agnostic. At its core, KOTINOS accelerates CI builds by: (1) populating and leveraging a cache of build images to avoid repeating environmental setup steps; and (2) inferring and reasoning about dependencies between build steps by tracing system calls during build execution. Our case study of three consumers of KOTINOS shows that accelerations are regularly triggered (87.9%–97.6% of the time) and when they are triggered, provide signi cant reductions in build time (74% of accelerated builds take at most half of the time of their non-accelerated counterparts). Furthermore, KOTINOS accelerates builds in open source software systems that practice process-level test invocation, with minimal resource overhead and without compromising build outcome.

Future Work. In future work, we will relax the conditions that KOTINOS currently requires in order to achieve robust acceleration. We list these conditions below.

1) Use a build tool that supports deterministic dependency resolution. If the project's build tool relies on an external service to determine the version of dependencies to be used during each build, KOTINOS will not re-invoke this dependency resolution service during builds unless the build speci cation le changes. This can lead to dependencies getting resolved to outdated or missing versions. To mitigate this problem, we currently require projects to pin dependency versions (including transitive dependencies) explicitly, using the lock le mechanisms that are provided by dependency management tools (e.g., Gemfile.lock in Bundler).
2) Test suite is isolated and idempotent. If state is persisted using les or database storage during the test execution and not restored to its original state after the test execution, subsequent builds will have access to the persisted state due to environment caching. This can yield misleading test results. Therefore, KOTINOS users must ensure that the testing environment is reset to its initial state before the test execution.
   Since most modern testing frameworks perform an environment reset during test execution, idempotency and isolation issues have been rarely observed in projects that use KOTINOS. In cases where idempotency issues persist, KOTINOS provides a mechanism that allows users to explicitly exclude (sub)processes from acceleration. For example, if a database needs to be recreated on every test run, then the command can be forced to re-execute rather than short-circuited.
3) Conditional behaviour during the build should be kept to a minimum. KOTINOS relies on system call traces to infer the dependency graph. Like any dynamic analysis, this may yield an incomplete view of the artifact under evaluation (i.e., build dependencies) if there is conditional behaviour. To mitigate this risk, after each build execution, the dependency graph is updated by isolating the steps of the build that were re-executed during warm builds.

Since these are best practices that are recommended for effective and robust automated testing, we believe that software projects should be striving for these build properties whether or not they choose to adopt KOTINOS. Even for the projects that are not currently following these best practices, adopting them will help not only to accelerate CI builds with KOTINOS, but will also yield other bene ts (e.g., preventing false positive or false negative test results). Nonetheless, in future work, we aim to expand the capabilities of KOTINOS so that even projects that do not ful ll the above conditions can bene t from build acceleration.

REFERENCES

[1] R. Abdalkareem, S. Mujahid, and E. Shihab. A machine learning approach to improve the detection of CI skip commits. *IEEE Trans. on Softw. Eng.*, 2020.

[2] R. Abdalkareem, S. Mujahid, E. Shihab, and J. Rilling. Which commits can be CI skipped? *IEEE Trans. Softw. Eng.*, 2019.

[3] B. Adams, K. de Schutter, H. Tromp, and W. de Meuter. MAKAO (demo). In *Proc. of Int. Conf. Softw. Maintenance*, pages 517–518, 2007.

[4] B. Adams, K. de Schutter, H. Tromp, and W. de Meuter. The evolution of the linux build system. *Electron. Commun. of the ECEASST*, 8, 2008.

[5] 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.

[6] C.-P. Bezemer, S. McIntosh, B. Adams, D. M. German, and A. E. Hassan. An empirical study of unspecified dependencies in make-based build systems. *Empirical Softw. Eng.*, 22(6):3117–3148, 2017.

[7] G. Brooks. Team pace keeping build times down. In *Agile 2008 Conference*, pages 294–297, 2008.

[8] W. J. Brown, H. W. McCormick III, and S. W. Thomas. *AntiPatterns and Patterns in Software Configuration Management*. John Wiley & Sons, Inc., 1999.

[9] Q. Cao, R. Wen, and S. McIntosh. Forecasting the duration of incremental build jobs. In *Proc. of Int. Conf. Softw. Maintenance and Evolution*, pages 524–528, 2017.

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

[11] S. Elbaum, G. Rothermel, and J. Penix. Techniques for improving regression testing in continuous integration development environments. In *Proc. of the Int. Symp. Found. Softw. Eng.*, pages 235–245, 2014.

[12] 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 Int. Conf. Softw. Eng. Companion*, pages 11–20, 2016.

[13] S. I. Feldman. Make — a program for maintaining computer programs. *Software: Practice and Experience*, 9(4):255–265, 1979.

[14] W. Felidré, L. Furtado, D. A. da Costa, B. Cartaxo, and G. Pinto. Continuous integration theater. In *Proc. Int. Symp. Empirical Softw. Eng. Measurement*, 2019.

[15] 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 Int. Conf. Autom. Softw. Eng.*, pages 87–97, 2018.

[16] K. Gallaba and S. McIntosh. Use and misuse of continuous integration features: An empirical study of projects that (mis)use Travis CI. *IEEE Trans. Softw. Eng.*, 2018.

[17] T. A. Ghaleb, D. A. da Costa, and Y. Zou. An empirical study of the long duration of continuous integration builds. *Empirical Softw. Eng.*, 2019.

[18] M. Gligoric, W. Schulte, C. Prasad, D. van Velzen, I. Narasamdya, and B. Livshits. Automated migration of build scripts using dynamic analysis and search-based refactoring. In *Proc. of Int. Conf. Object Oriented Programming Systems Languages & Applications*, pages 599–616, 2014.

[19] F. Hassan and X. Wang. HireBuild: An Automatic Approach to History-Driven Repair of Build Scripts. In *Proc. of Int. Conf. Softw. Eng.*, 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. Joint Meeting Eur. Softw. Eng. Conf. Int. Symp. Found. Softw. Eng.*, 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 Int. Conf. Autom. Softw. Eng.*, pages 426–437, 2016.

[22] S. Holm. A simple sequentially rejective multiple test procedure. *Scandinavian Journal of Statistics*, 1979.

[23] C. Lebeuf, E. Voyloshnikova, K. Herzig, and M.-A. Storey. Understanding, debugging, and optimizing distributed software builds: A design study. In *Proc. of Int. Conf. Softw. Maintenance and Evolution*, pages 496–507, 2018.

[24] Y. Li, J. Wang, Y. Yang, and Q. Wang. Method-level test selection for continuous integration with static dependencies and dynamic execution rules. In *Int. Conf. Softw. Quality, Reliability and Security*, pages 350–361, 2019.

[25] C. Macho, S. McIntosh, and M. Pinzger. Predicting build co-changes with source code change and commit categories. In *Int. Conf. on Softw. Analysis, Evolution, Reengineering*, 2016.

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

[27] S. McIntosh, B. Adams, M. Nagappan, and A. E. Hassan. Mining co-change information to understand when build changes are necessary. In *Int. Conf. Softw. Maintenance Evolution*, 2014.

[28] S. McIntosh, B. Adams, M. Nagappan, and A. E. Hassan. Identifying and Understanding Header File Hotspots in C/C++ Build Processes. *Autom. Softw. Eng.*, 23(4):619–647, 2015.

[29] A. Memon, Z. Gao, B. Nguyen, S. Dhanda, E. Nickell, R. Siemborski, and J. Micco. Taming google-scale continuous testing. In *Proc. of Int. Conf. Softw. Eng.: Softw. Eng. in Practice Track*, 2017.

[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 Trans. Softw. Eng.*, 43(12):1178–1193, 2017.

[31] A. Miller. A hundred days of continuous integration. In *Proc. of the Agile Conf.*, pages 289–293, 2008.

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

[33] G. Pinto, M. Reboucas, and F. Castor. Inadequate testing, time pressure, and (over) confidence: A tale of continuous integration users. In *Proc. of the Int.*