Department of Electrical and Computer Engineering
McGill University, Montréal

Improving the Robustness and Efficiency of Continuous Integration and Deployment

Ph.D. Thesis

Keheliya Gallaba

November 2021

A thesis submitted to McGill University in partial fulfillment
of the requirements of the degree of Doctor of Philosophy

Abstract

Modern software is developed at a rapid pace. To sustain that rapid pace, organizations rely heavily on automated build, test, and release steps. To that end, Continuous Integration and Continuous Deployment (CI/CD) services take the incremental codebase changes that are produced by developers, compile, link, and package them into software deliverables, verify their functionality, and deliver them to end users.

While CI/CD processes provide mission-critical features, if they are misconfigured or poorly operated, the pace of development may be slowed or even halted. To prevent such issues, in this thesis, we set out to study and improve the *robustness* and *efficiency* of CI/CD processes.

First, we present two empirical studies that focus on *robust configuration* of CI/CD processes. To understand the ways in which CI/CD features are being used, we analyze a curated sample of 9,312 open source projects that are hosted on GitHub and have adopted the popular Travis CI service. We find that explicit deployment code is rare. Then, to analyze feature misuse, we propose Hansel—an anti-pattern detection tool for Travis CI specifications. We define four anti-patterns and Hansel detects anti-patterns in the Travis CI specifications of 894 projects (10%) in the corpus. Furthermore, we propose Gretel—an anti-pattern removal tool for Travis CI specifications, which can remove 70% of the most frequently occurring anti-pattern automatically.

Our third empirical study focuses on *robust CI/CD outcome data*. In this work, we use openly available project metadata and CI/CD results of 1,276 GitHub projects that use Travis CI, to better understand the extent to which noise and heterogeneity are present in CI/CD outcome data. We find that: (1) 12% of passing builds have an actively ignored failure; (2) 9% of builds have a misleading or incorrect outcome on average; and (3) at least in 44% of the broken builds, the breakage is local to a subset of build variants.

Our fourth empirical study focuses on improving the *efficiency of CI/CD services*. We propose a programming 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. To evaluate our approach, we mine 14,364 historical CI build records spanning three proprietary and seven open-source software projects. We find that accelerated builds achieve a substantial speed-up (two-fold in 74% of accelerated builds) with 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).

Our final empirical study identifies *opportunities for service providers* to improve *robustness and efficiency* of CI/CD processes by analyzing signal-generating builds (i.e., builds that pass or fail due to project factors) and non-signal-generating builds (e.g., incompleted builds due to provider infrastructure issues). In this study, we analyze 23.3 million builds spanning 7,795 open source projects that used the CircleCI service from 2012 to 2020. Our observations demonstrate the ways in which existing research breakthroughs (e.g., build acceleration, automated program repair) may benefit CI/CD providers, as well as the ways in which these approaches should be tailored to generate the most value. For example, since the heaviest users account for a growing proportion of the build activity and resources over the studied time period (measures of inequality like the Gini coefficient growing from 14% to 98%), approaches that are catered to optimizing these projects will likely generate more value for service providers than blanket solutions. Furthermore, *efficiency* in CI pipelines can be improved by reducing bottlenecks in the compilation and testing stages of signal-generating builds. Addressing configuration and resource allocation issues will reduce the number of non-signal-generating builds, increasing the *robustness* of CI pipelines.

Abrégé

Les logiciels modernes sont développés à un rythme rapide. Pour maintenir ce rythme, les organisations qui développe des logiciels s'appuient fortement sur l'automatisation de la construction, des tests, et de la diffusion de logiciels. à cette fin, les services d'intégration continue et de déploiement continu (CI/CD) prennent les modifications de code incrémentielles produites par les développeurs, les compilent, les relient et les regroupent dans des logiciels livrables, vérifient leur fonctionnalité, et les livrent aux utilisateurs.

Bien que les processus CI/CD fournissent des fonctionnalités critiques, s'ils sont mal configurés ou mal exploités, le rythme de développement peut être ralenti, voire arrêté. Pour éviter de tels problèmes, dans cette thèse, nous avons entrepris d'étudier et d'améliorer la robustesse et l'efficacité des processus CI/CD.

Premièrement, nous présentons deux études empiriques qui se concentrent sur la configuration robuste des processus CI/CD. Pour comprendre la manière dont les fonctionnalités CI/CD sont utilisées, nous analysons un échantillon de 9,312 projets à code source ouvert, hébergés sur GitHub, et ayant adopté le service Travis CI. Nous constatons que le code de déploiement est rarement explicite. Ensuite, pour analyser les erreurs dans l'utilisation des fonctionnalités de Travis CI, nous proposons Hansel, un outil de détection d'anti-modèle pour les spécifications de Travis CI. Nous définissons quatre anti-modèles. Hansel détecte ces anti-modèles dans les spécifications Travis CI de 894 projets (10%) du corpus. De plus, nous proposons Gretel, un outil de suppression d'anti-modèle pour les spécifications Travis CI, qui peut supprimer automatiquement 70 % des anti-modèles les plus fréquents.

Notre troisième étude empirique se concentre sur les données de CI/CD qui produisent des résultats robustes. Ce travail utilise des métadonnées de projet librement disponibles et les résultats CI/CD de 1,276 projets GitHub qui utilisent Travis CI. Nous utilisons ces projets afin de mieux comprendre dans quelle mesure le bruit de fond et l'hétérogénéité sont présents dans les données de résultats CI/CD. Nous constatons que : (1) 12% des construction de logicielle réussis ont un échec qui est activement ignoré ; (2) en moyenne, 9% des construction de logicielle ont un résultat trompeur ou incorrect; et (3) dans au moins 44% des construction

de logicielle défectueuses, la défectuosité est locale et dans un sous-ensemble de variantes de construction.

Notre quatrième étude empirique porte sur l'amélioration de l'efficacité des services CI/CD. Nous proposons une approche indépendante du langage de programmation pour déduire des données à partir desquelles des décisions d'accélération de construction de logiciels peuvent être prises sans se fier aux spécifications de construction de logiciel. Après avoir déduit ces données, notre approche accélère les systèmes de construction de logiciel CI en mettant en cache l'environnement du système de construction et en sautant les étapes qui n'affecte pas le système de construction. Pour évaluer notre approche, nous extrayons 14,364 archives de construction de logiciels de systèmes CI couvrant trois projets logiciels propriétaires et sept logiciels à code source ouvert. Nous constatons que les construction logiciel accélérés atteignent une accélération substantielle (deux fois plus rapides dans 74% des construction logiciel accélérés) avec une surcharge de ressources minimale (c. < 1% d'utilisation médiane du processeur, 2 Mo à 2.2 Go d'utilisation médiane de mémoire, et 0.4 Go à 5.2 Go de surcharge médiane de stockage).

Notre étude empirique finale identifie des opportunités pour les fournisseurs de services pour améliorer la robustesse et l'efficacité de leurs processus CI/CD en analysant les construction logiciel générant des signaux (c. construction logiciel qui réussissent ou échoue en raison de facteurs inhérents aux projets des utilisateurs). Dans cette étude, nous analysons 23,3 millions de constructions de logiciel couvrant 7,795 projets à code source ouvert qui ont utilisé le service CircleCI de 2012 à 2020. Nos observations démontrent les façons dont les percées de recherche existantes (par exemple, l'accélération des systèmes de construction logiciel, la réparation automatisée des programmes) peuvent bénéficier les fournisseurs de CI/CD, ainsi que les manières dont ces approches devraient être adaptées pour générer le plus de valeur possible pour ces fournisseurs. Par exemple, étant donné que les plus gros utilisateurs représentent une proportion croissante de l'activité de construction de logiciel et des ressources au cours de la période étudiée (mesures d'inégalité comme le coefficient de Gini passant de 14% à 98%), les approches conçues pour optimiser ces projets génèrent plus de valeur pour les fournisseurs de services que les solutions globales. De plus, l'efficacité des pipelines CI peut être améliorée en réduisant les obstacles dans les étapes de compilation et de test pour la construction de logiciel générant des signaux. La résolution des problèmes de configuration et d'allocation des ressources réduira le nombre de constructions logiciel qui ne génère pas de signal, augmentant ainsi la robustesse des pipelines CI.

Acknowledgements

As I near the end of my journey as a Ph.D. student at McGill University, I would like to express my gratitude to everyone who made my stay in the beautiful city of Montréal a memorable and productive one. I would like to give special thanks to the following people who made this thesis possible.

Ph.D. Advisor. I owe special gratitude to my advisor Professor Shane McIntosh for the excellent guidance and continuous support.

Ph.D. Committee Members. My thanks go out to Professor Martin Robillard, Professor Gunter Mussbacher, and Professor Benjamin Fung for the helpful feedback and insights.

Colleagues. I would like to thank the fellow Rebels at the Software Repository Excavation and Build Engineering Labs: Ray, Shivashree, Christophe, Farida, Noam, Mehran, and Farshad. I extend my sincere thanks to Márton Búr for offering assistance and sharing the experiences.

Collaborators. It was an honour to work with an amazing network of researchers from different parts of the world to solve challenging problems. I am particularly grateful to the collaborators who co-wrote publications related to the content of this thesis (Christian Macho, Martin Pinzger, Yves Junqueira, John Ewart, and Maxime Lamothe) and the co-authors on topics outside of the scope of this thesis (Yash, Yusaira, Marco, Eduardo, Rahul, Andrés, Noam, Michael, Sadnan, Durham, Oliver, and Matthew).

Funding. I am deeply grateful for the generous financial support provided by the McGill Engineering Doctoral Award (MEDA) and the FRQNT Doctoral Research Award. Furthermore, by providing funds to conduct research abroad, Mitacs Accelerate International Award and McGill Graduate Mobility Award helped me expand my research horizons.

Friends and family. Amma, Appachchi, Nangi, and Thamali, I am deeply indebted to you for making me who I am today with your encouragement, support, and love. Thank you to all my friends, old and new, for the stimulating conversations during this journey.

Related Publications

Each of my contributions presented henceforth is a result of the research conducted at the Department of Electrical and Computer Engineering at McGill University under the supervision of Dr. Shane McIntosh. A detailed description of the contributions is presented in Section 1.3.

- Keheliya Gallaba and Shane McIntosh. Use and misuse of continuous integration features: an empirical study of projects that (mis)use Travis CI. *IEEE Transactions on Software Engineering (TSE)*, 2018, pp. 33–50. DOI: 10.1109/tse.2018.2838131. (Chapter 4 and Chapter 5)
- Keheliya Gallaba, Christian Macho, Martin Pinzger, and Shane McIntosh. Noise and heterogeneity in historical build data: an empirical study of Travis CI. In: *Proceedings of the 33rd International Conference on Automated Software Engineering (ASE)*, pp. 87–97. 2018. DOI: 10.1145/3238147.3238171. (Chapter 6)
- Keheliya Gallaba. Improving the robustness and efficiency of continuous integration and deployment. In: *Proceedings of the 35th International Conference on Software Maintenance and Evolution (ICSME), Doctoral Symposium*, pp. 619–623. IEEE, 2019. DOI: 10.1109/icsme.2019.00099.
- Keheliya Gallaba, Yves Junqueira, John Ewart, and Shane Mcintosh. Accelerating continuous integration by caching environments and inferring dependencies. *IEEE Transactions on Software Engineering (TSE)*, 2020. DOI: 10.1109/TSE.2020.3048335. (Chapter 8)

The following publications are not directly related to the material in this thesis, but were produced in parallel to the research performed for this thesis.

- Yash Gupta, Yusaira Khan, Keheliya Gallaba, and Shane McIntosh. The impact of the adoption of continuous integration on developer attraction and retention. In: *Proceedings of the 14th International Conference on Mining Software Repositories (MSR), Mining Challenge*, pp. 491–494. IEEE, 2017. DOI: 10.1109/msr.2017.37.
- Marco Manglaviti, Eduardo Coronado-Montoya, Keheliya Gallaba, and Shane McIntosh. An empirical study of the personnel overhead of continuous integration. In: *Proceedings*

of the 14th International Conference on Mining Software Repositories (MSR), Mining Challenge, pp. 471–474. IEEE, 2017. DOI: 10.1109/msr.2017.31.

- Rahul Amlekar, Andrés Felipe Rincón Gamboa, Keheliya Gallaba, and Shane McIntosh. Do software engineers use autocompletion features differently than other developers? In: *Proceedings of the 15th International Conference on Mining Software Repositories (MSR), Mining Challenge*, pp. 86–89. ACM, 2018. DOI: 10.1145/3196398.3196471.
- Noam Rabbani, Michael S. Harvey, Sadnan Saquif, Keheliya Gallaba, and Shane McIntosh. Revisiting "programmers' build errors" in the visual studio context. In: *Proceedings of the 15th International Conference on Mining Software Repositories (MSR), Mining Challenge*, pp. 98–101. ACM, 2018. DOI: 10.1145/3196398.3196469.
- Durham Abric, Oliver E. Clark, Matthew Caminiti, Keheliya Gallaba, and Shane McIntosh. Can duplicate questions on Stack Overflow benefit the software development community? In: *Proceedings of the 16th International Conference on Mining Software Repositories (MSR), Mining Challenge*, pp. 230–234. 2019. DOI: 10.1109/MSR.2019.00046.

Contents

List of Figures

List of Tables

Part I

Preliminaries

Chapter 1

Introduction

Software today is developed at an accelerated pace. For example, the Facebook Android mobile application has increased their release frequency to one deployment every week [1], while Flickr web application is deployed to production more than ten times a day [2]. The IMVU chat application is released up to 50 times per day [3].

The influence of agile development and the contemporary "continuous" development approaches have enabled the accelerated pace of modern software development. Continuous Integration (CI) [4] is one such software development practice where changes to a codebase are integrated into upstream repositories after being built and verified by an automated workflow. Continuous Deployment (CD) takes this a step further, ensuring that the software can be reliably released at any time by automating the deployment and release workflows as well. Prior work [5–17] has shown that CI/CD is broadly adopted by open source and proprietary software teams (e.g., Mozilla, Google, Microsoft, and ING). The adoption of CI/CD has been linked to increased developer productivity [9], speeding up development [10], and improving software quality [7, 11]. Due to the popularity of CI/CD, cloud-based CI/CD service providers (e.g., Travis CI, CircleCI, Jenkins, and AppVeyor) have also emerged, making CI/CD available to the masses.

1.1 Problem Statement

Software teams often encounter difficulties when adopting CI/CD in their organizations, leading to unstable software delivery pipelines and wasted resources. For example, a suboptimal configuration of Mozilla's CI service was inflating the operating cost of their CI service by

16%.[1] Moreover, a typo in the CI specification of the *geoscixyz/gpgLabs*[2] project halted deployment of new releases.[3] In our work, we tackle three problems that relate to CI/CD services:

Misconfiguration of CI/CD Environments. A typical CI/CD service has different nodes for *creating* build jobs, *processing* them, and *reporting* on the outcome. While configuring job creation and job reporting nodes is relatively simple (e.g., reporting only needs a contact method like an email address and a triggering event type like build failures), configuring job processing nodes is complex, being typically decomposed into install, script, and deploy phases, which are further divided into more specific sub-phases (e.g., before install, after script, after deploy). Misconfiguration of CI/CD environments may yield suboptimal build performance (e.g., violating the semantics of CI/CD specifications hinders the runtime optimizations that CI/CD operators can perform) or conceal faults (e.g., misspelled properties and their associated commands are silently ignored by popular CI/CD runtime environments such as Travis CI).

Misinterpretation of CI/CD Results. In the recent literature [12], build outcomes are considered to be *free of noise*. However, we find that in practice, some builds that are marked as successful contain breakages that need attention yet are ignored. Furthermore, there are builds that are marked as broken. However, since they do not receive the immediate attention of the development team, they may not be as distracting as previously assumed. In prior work, builds are also considered to be *homogeneous*. However, builds vary in terms of the number of executed jobs and the number of supported build-time configurations. Unawareness of the noise and differences in builds adversely affects the decision making in research and practice.

Inefficient Use of CI/CD Resources. A key goal of CI/CD is to provide rapid feedback to software teams and releases to users throughout the development process. Software organizations invest resources in the operation and maintenance of CI/CD services in order to benefit from such a rapid feedback and release cycle. However, inefficient execution and long job durations can lead to wasted resources.

[1] `https://oduinn.com/2013/12/13/the-financial-cost-of-a-checkin-part-2/`
[2] `https://github.com/geoscixyz/gpgLabs`
[3] `https://github.com/geoscixyz/gpgLabs/issues/72`

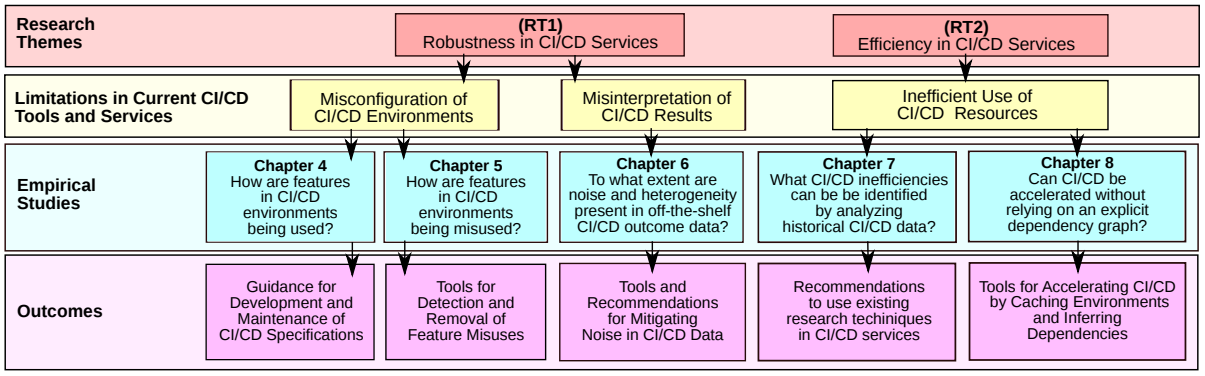


Figure 1.1: An overview of the scope of this thesis.

This thesis aims to address the difficulties that we outline above in order to help development teams to reap the most benefit from CI/CD processes. More specifically, we evaluate the following research hypothesis:

Hypothesis.	Defect-prone and slow CI/CD pipelines can lead to considerable amounts of wasted resources for CI/CD adopters and service providers. Specifications and outcome data from CI/CD services can be leveraged to increase the **robustness** and **efficiency** of CI/CD.

In the next section, we present the overview of our approach for evaluating the hypothesis.

1.2 Thesis Overview

Figure 1.1 provides an overview of the scope of this thesis. We have two main research themes (red boxes) discussed in this thesis. Our research themes are motivated by three main limitations in state-of-the-art CI/CD environments (yellow boxes). To tackle these limitations, we conduct a series of empirical studies (cyan boxes). Finally, we describe potential outcomes that result from each empirical study (magenta boxes). Each empirical study is presented in its own chapter. We introduce our two research themes and the empirical studies below.

(RT1) Robustness in CI/CD Services

Robustness is related to making the infrastructure resilient to unexpected events or mistakes. In terms of the robustness of CI/CD services, misuse of configuration affects CI/CD users,

while misinterpretation of outcome data may negatively impact stakeholders such as CI/CD tool builders and researchers.

Like programming languages, configuration languages also offer features, which can be used or misused. For example, Travis CI users can use features like **branches**, which specify which VCS branches to monitor for commit activity. Commits that appear on the monitored branches will trigger build jobs. CI/CD configuration can also be misused, e.g., when unsupported or deprecated commands are used.

In this thesis, we conduct two empirical studies to investigate how CI/CD features are being used and misused:

Chapter 4: *Usage of Features in CI/CD Environments*

To study how features in CI/CD specification files are being used, we analyze a curated sample of 9,312 open source projects that are hosted on GitHub and have adopted the popular Travis CI service.

Chapter 5: *Misuse of Features in CI/CD Environments*

In this work, we define CI/CD misconfiguration patterns. Then, we create development tools that can automatically detect and remove them.

CI/CD outcome data is used by software practitioners and researchers when building tools and proposing techniques to solve software engineering problems. However, it may be harmful to use this data "off the shelf" without checking for noise and complexities. Our next empirical study characterizes CI/CD outcome data according to harmful assumptions that one may make about its cleanliness and homogeneity.

Chapter 6: *Noise and Heterogeneity in CI/CD Build Data*

In this work, we analyze CI/CD outcome data from large software projects to quantify the noise and characterize their nuances. For this purpose, we use openly available project metadata and CI/CD results of 1,276 GitHub projects that use Travis CI.

(RT2) Efficiency in CI/CD Services

Developers adopt CI/CD with the intention of speeding up development [11]. However, the results of the CI/CD builds could be delayed due to bottlenecks in the execution of CI/CD jobs and other inefficiencies during the operation of CI/CD services. Identifying opportunities for

managing resources efficiently will help CI service providers to keep operational costs low while delivering fast and reliable CI/CD services.

Chapter 7: *CI/CD Service Providers' Perspective*

To study CI/CD from the perspective of service providers, we conduct a case study of CircleCI– one of the most popular CI service providers for projects hosted on GitHub. Our dataset includes 23.3 million builds spanning 7,795 open source projects that use the CircleCI service during the period of 2012–2020.

Build tools that have been proposed to reduce build duration by executing incremental builds have two key limitations. These build tools rely upon a graph of build dependencies that is specified by developers in build configuration files (e.g., Bazel BUILD files). Moreover, the accelerated build tools are designed to replace existing build tools, increasing the barrier to entry.

Chapter 8: *Accelerating Continuous Integration / Continuous Delivery*

To address existing limitations, we propose a build acceleration approach for CI/CD services that disentangles build acceleration from the underlying build tool. We evaluate our approach by mining 14,364 historical CI/CD 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.

1.3 Thesis Contributions

This thesis demonstrates that:

- Research and tooling for CI/CD configuration would have the most immediate impact if it were focused on supporting the configuration of job processing nodes or reducing the complexity of deployment configuration. Moreover, research and tooling for CI/CD configuration should focus on the creation of an initial specification rather than supporting specification maintenance because the configuration files are rarely modified in practice. (Chapter 4)
- Developers misuse and misconfigure CI/CD specifications. The antipatterns that we define can expose a system to security vulnerabilities, cause unintended CI/CD behaviour, or delay SQA activities until after deployment. Gretel, our antipattern detector, can detect misuse and misconfiguration of CI/CD specifications. (Chapter 5)

- Off-the-shelf CI/CD build outcome data is noisy and build breakages vary with respect to the number of impacted jobs and the causes of breakage. Researchers should make sure that noise is filtered out and heterogeneity is accounted for before subsequent analyses are conducted on CI/CD build outcome data. Build reporting tools should consider providing richer interfaces to better represent the nuances of build outcome data. (Chapter 6)
- CI/CD services may benefit from research breakthroughs in the areas of build acceleration and automated program repair. Approaches to make testing and compiling faster will benefit a large proportion of CI/CD users. The heaviest users (and CI/CD providers as a consequence) will benefit most from additional bandwidth during dependency installation. (Chapter 7)
- CI/CD builds can be accelerated by caching the build environment and skipping unaffected build steps. Since our build acceleration approach is agnostic of the programming languages and build tools being used, teams can benefit without requiring considerable build migration effort. Moreover, our approach can accelerate builds with minimal CPU, memory, and storage overhead. (Chapter 8)

1.4 Structure of the Thesis

The remainder of this thesis is organized as follows. Chapter 2 provides background information on the modern CI/CD process and defines key terms. Chapter 3 presents prior research related to improving the robustness and efficiency of CI/CD. Chapters 4 and 5 present the results of our analyses of the use and misuse of CI/CD features, respectively. In Chapter 6, we present our study of the noise and heterogeneity in CI/CD build data. Chapter 7 discusses CI/CD service providers' perspective. Chapter 8 presents the results of our study of accelerating CI/CD. Finally, Chapter 9 draws conclusions and discusses promising avenues for future work.

Chapter 2

Background and Definitions

In this chapter, we define the core concepts of modern CI/CD that are useful for understanding our work. To be consistent with practice, we adhere to the terminology common to popular CI/CD services, using Travis CI as a concrete example where applicable.

2.1 Modern CI/CD Process

The main goal of CI/CD is automating the integration of software as soon as it is developed so that it can be released rapidly and reliably [4]. Figure 2.1 provides an overview of the cycle. We describe each step below.

Build-triggering events

In projects that adopt CI/CD, the cycle begins with a build-triggering event. These events can occur in the development, review, or integration stages of code change development. While a feature is being developed, builds can be triggered manually by the developer to try out the feature under development. Later, when the code is submitted to be reviewed, builds are triggered to avoid wasting the reviewer's time on patches that do not compile. Finally, when the change is integrated into the project VCS, a build is triggered to ensure that the change does not introduce regression errors.

Build job creation service

When a build-triggering event occurs, a build job creation node will add a job to the queue of pending build jobs if certain criteria are met. For example, in Travis CI, developers can specify the VCS branches on which commits should (or should not) generate build jobs.

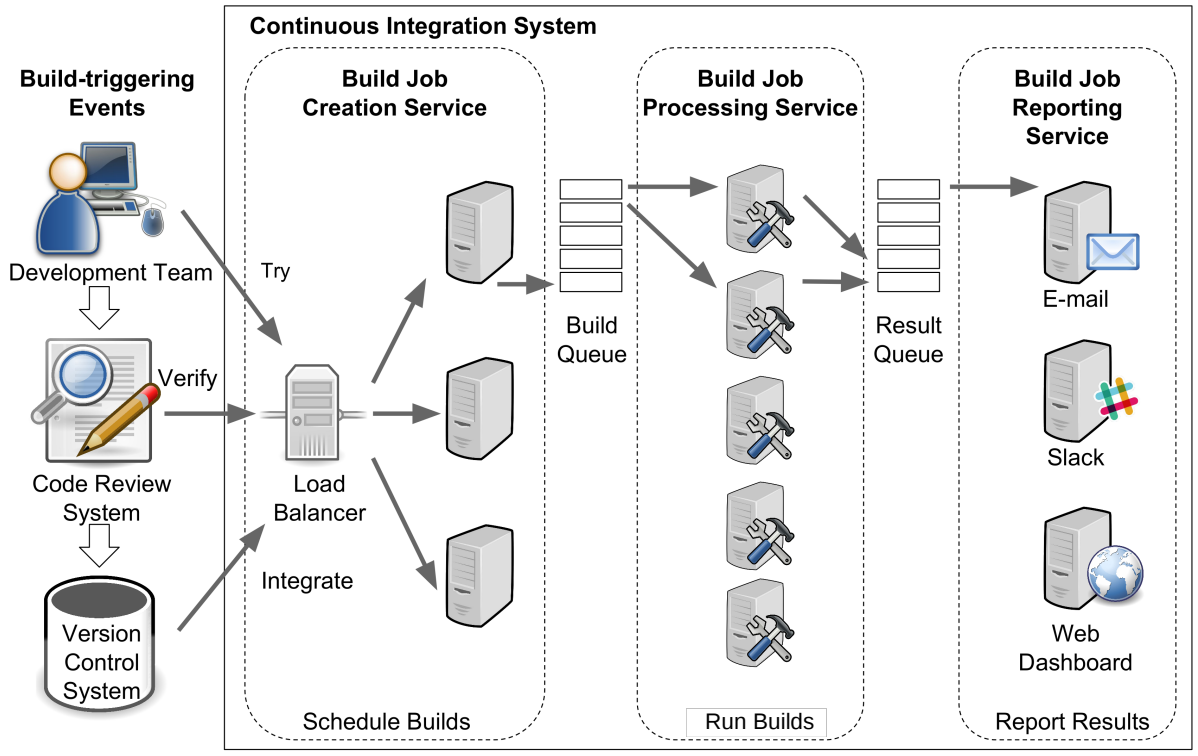


Figure 2.1: Main components of a continuous integration system: Build job creation, processing, and reporting.

Build job processing service

Build jobs in the pending queue will be allocated to build job processing nodes for execution. The job processing node will first download the latest version of the source code and apply the change under consideration. Next, the job processing node will initiate the build process, which will compile the system (if necessary), execute a suite of automated unit and integration tests to check for regression, and in the case of Continuous Delivery (CD) [10], make the updated system available for users to download or interact with. Finally, the job processing node will add the results of the build job to the reporting queue.

Build job reporting service

In this final stage, build job results in the reporting queue will be communicated to the development team. Reporting preferences can be configured such that particular recipients receive notifications when build jobs are marked as successful, unsuccessful, or irrespective of the

job status. Traditionally, these results were shared via mailing lists or IRC channels; however, other communication media are also popular nowadays (e.g., Slack, web dashboards).

Operating and maintaining CI/CD infrastructure is a burden for modern software organizations. As organizations grow, their CI/CD infrastructure needs to scale up in order to handle the increased load that larger teams will generate. Moreover, if additional platforms are added (e.g., to attract more users), this too will generate additional CI/CD load.

2.2 Cloud-Based CI/CD Services

Instead of investing in on-site CI/CD infrastructure, modern organizations use cloud-based CI/CD services, such as Travis CI and CircleCI. These service providers enable organizations to have scalable CI/CD services without operating and maintaining CI/CD infrastructure internally.

Travis CI and CircleCI, the two CI services that we closely analyze in this thesis, offer similar features to their user base. Apart from the cloud-based offering, both services offer an on-premise solution with security and configuration for running CI/CD builds in a private cloud or a data center. Both services can be configured with a YAML file, which is placed in the root directory of the project repository. Both offer running builds on Windows, Linux, Android, iOS, and macOS operating systems. Software projects developed in over 30 popular programming languages can run their CI builds on these services. Furthermore, both Travis CI and CircleCI can integrate with popular version control systems such as GitHub and Bitbucket. Both services support plugins and third-party integration for software development tasks such as static analysis, code coverage, testing, deployment, and monitoring. Travis CI and CircleCI both support parallel testing.

In terms of differences, CircleCI has some advanced developer-friendly features compared to Travis CI. For debugging build failures CircleCI users have the option of connecting to build servers via SSH. Moreover, CircleCI users can choose to parallelize their tests using multiple criteria such as past timing data. CircleCI also supports 'Orbs', which are reusable environment configurations that help to automate repetitive processes, and speed-up the initial setup. Furthermore, while open source projects can run their CI builds free of charge on CircleCI (up to 400,000 credits per month), Travis CI has terminated the free plan for open source projects since November 2020.

Recently, version control service providers have also started offering CI/CD services (e.g., GitHub Actions,[1] GitLab CI/CD,[2] and BitBucket Pipelines[3]) focusing on the convenience of the users to trigger CI/CD workflows via version control activity. Furthermore, major public cloud vendors are also providing CI/CD capabilities (e.g., Microsoft Azure DevOps,[4] AWS CodeStar,[5] and Google Cloud Build[6]) with features to facilitate the entire continuous delivery toolchain.

2.3 Configuring Cloud-Based CI/CD Services

Cloud-Based CI/CD service users can define which tools are needed and the order in which they must be executed to complete a build job. These configuration details are typically stored in a file (e.g., `.circleci/config.yml` for CircleCI, `.travis.yml` for Travis CI) using a YAML-based DSL, which appears in the root directory of a GitHub repository. The configuration file can also specify programming language runtimes, and other environment configuration settings that are needed to execute build jobs. As a concrete example to illustrate, we use the `.travis.yml` file of Travis CI below. However, note that equivalents are available in CircleCI and other cloud-Based CI/CD services.

Figure 2.2 shows that `.travis.yml` files consist of node configuration and build process configuration sections. We describe each section below.

2.3.1 Node Configuration

This section specifies how CI/CD nodes should be prepared before building commences.

- **Build job creation nodes**: In this subsection, nodes that are responsible for creating build jobs can be configured. For example, the `branches` property specifies the VCS branches where changes that land should generate build jobs.
- **Build job processing nodes**: In this subsection, nodes that are responsible for processing build jobs can be configured. For example, since different programming languages have different basic toolchain requirements (e.g., Python projects require the `python` interpreter to be installed, while Node.js projects require the `node` interpreter to be in-

[1] `https://github.com/features/actions`
[2] `https://docs.gitlab.com/ee/ci/`
[3] `https://bitbucket.org/product/features/pipelines`
[4] `https://azure.microsoft.com/en-us/services/devops/`
[5] `https://aws.amazon.com/codestar/`
[6] `https://cloud.google.com/build`

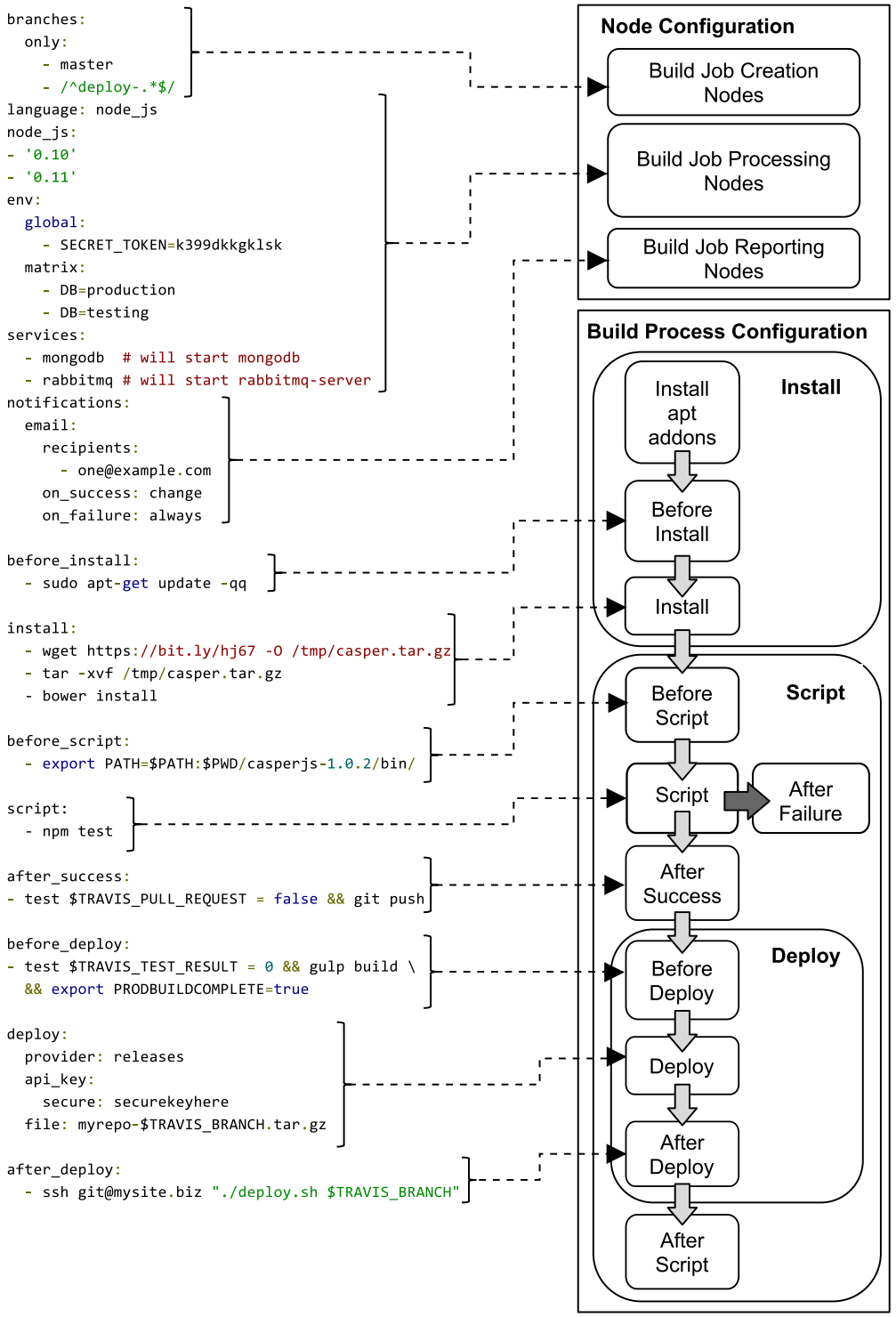


Figure 2.2: A `.travis.yml` configuration file and how it maps to the Travis CI life cycle.

stalled), specifying the `language` property allows the Travis CI runtime to configure processing nodes appropriately. Moreover, if there are libraries and services that need to be installed on the job processing nodes prior to build execution, they can be specified using the `services` property. The environment variables that need to be set prior to build execution can be configured using the `env` property. In practice, CI services rely on existing containers (e.g., Docker) to set up certain services/environments.

- **Build job reporting nodes**: In this subsection, nodes that are responsible for reporting on the status of build jobs can be configured. Notification services, such as e-mail and Slack, are configured to notify the development team about the status of build jobs. For example, using the `notifications` property, Travis CI users can specify the list of recipients of build status reports (`recipients`) and the scenarios under which they should be notified (`on_success`, `on_failure`).

2.3.2 Build Process Configuration

This section is comprised of `install`, `script`, and `deploy` phases, which each consists of sub-phases. These sub-phases check pre- and post-conditions before (`before_X`) and after (`after_X`) executing the main phase.

- The `install` phase prepares job processing nodes for build job execution, and has `install_apt_addons`, `before_install`, and `install` sub-phases. Unless specified, the phase runs a default command for the specified programming language. For example, Travis CI runs `npm install` by default for Node.js projects.
- The `script` phase executes the bulk of the build job, and has `before_script`, `script`, `after_success`, `after_failure`, and `after_script` sub-phases. In this phase, systems are compiled, tested, scanned by static code analyzers, and packaged for deployment. Similar to the `install` phase, `script` runs default commands for the specified programming language, unless otherwise specified. For example, Travis CI runs `npm test` by default for Node.js projects.
- The `deploy` phase makes newly produced deliverables visible to system users, and has `before_deploy`, `deploy`, and `after_deploy` sub-phases. When this phase is present, the CI process is transformed into a continuous delivery process [10], where regression-free changes are released to system users.

2.4 The Anatomy of a CI/CD Build

In a typical CI/CD service, a **build** is comprised of one or more jobs. For example, a build can have multiple jobs, each of which tests the project with a different variant of the development or runtime environment. Once all of the jobs in the build are finished, the build is also finished.

For each job, cloud-based CI/CD providers report one of four outcomes:

- **Passed.** The project was built successfully and passed all tests. All phases terminate with an exit code of zero.
- **Errored.** If any of the commands that are specified in the *before_install, install,* or *before_script* phases of the build lifecycle terminate with a non-zero exit code, the build is labelled as errored and stops immediately. Sometimes CI/CD providers will experience infrastructure failures due to issues that are unrelated to the build. Builds that were running during these incidents will also be labelled as errored.
- **Failed.** If a command in the *script* phase terminates with a non-zero exit code, the build is labelled as failed, but execution continues with the *after_failure* phase.
- **Cancelled.** A Travis CI user with sufficient permissions can abort the build using the web interface or the API. Such builds are labelled as cancelled.

Chapter 3

Related Work

This chapter provides an overview of the current state-of-the-art in six general topics covering the fields relevant to the subsequent chapters of this thesis.

3.1 Continuous Integration

With the practice of CI becoming ubiquitous among practitioners, the topic has been extensively studied by software engineering researchers [3].

Recent work has characterized CI practices and outcomes along different dimensions. Meyer [13] discussed features of the CI tools that were used by practitioners. He emphasizes the importance of good tooling, fully automated builds, fast test suites, feature-toggling, and monitoring for CI. Ståhl and Bosch [14] also provided a systematic overview of CI practices and their differences from a technical perspective. Vasilescu et al. [9] studied quality and productivity outcomes of using CI. They find that teams that are using CI are significantly more effective at merging the pull requests of core members.

In addition to positive outcomes, challenges and limitations of CI have been pointed out by researchers [11, 15]. For example, Hilton et al. [2] analyzed open source projects from GitHub and surveyed developers to understand which CI systems developers use, how developers use CI, and reasons for using CI (or not). They conclude that the main reason why open source projects choose not to use CI is that the developers are not familiar enough with it. Similarly, Elazhary et al. [16] have studied the CI practice at three software organizations based on a mixed methods approach to identify benefits and challenges experienced by the software teams during CI adoption. In a recent qualitative study, Hilton et al. [17] also found that,

when adopting CI, developers face trade-offs between speed and certainty, accessibility and security, and configurability and ease of use.

Other works focus on improving specific stages of the CI process. Beller et al. [18] studied testing practices in CI, particularly focusing on Java and Ruby projects. They conclude testing is an established and integral part in the CI process of open source software. However, Beller et al. [18] also observe a latency of more than 20 minutes between writing code and receiving test feedback from CI when compared to the fast-paced nature of testing in the local environments. They suggest that low test failure rates from CI are a sign that developers submit pre-tested contributions to CI. Similarly, Elbaum et al. [19] propose algorithms based on test case selection and prioritization techniques to make CI processes more cost effective. Other work has studied how to improve the effectiveness of automated testing in CI [20, 30] and how CI can be extended to include additional performance and robustness tests when standard testing frameworks are insufficient for highly concurrent, real-time applications [22].

The goal of our work explained in detail in Chapter 4 of this thesis is to characterize the usage of CI features by analyzing a large corpus of existing CI specifications. Our work is complementary to prior studies, contributing to a larger understanding of how CI tools and techniques are being adopted in real-world projects.

3.2 Software Configuration Smells

To the best of our knowledge, our work on CI misuse discussed in Chapter 5 is the first to define, detect, and remove anti-patterns in CI specifications; however, prior work has explored anti-patterns in the context of other configuration files. Brown et al. [23] published a catalog of anti-patterns and patterns for software configuration management. Shambaugh et al. [24] proposed Rehearsal, a verification tool for Puppet configurations. Sharma et al. [25] have also recently explored smells that are related to the Puppet configuration management language. They presented a set of implementation and design configuration smells that violate recommended best practices. Bent et al. [26] surveyed developers and used the findings to develop a Puppet code quality analysis tool. Rahman and Williams [27] applied text mining techniques to identify defects in Puppet scripts, identifying file system operations, infrastructure provisioning, and user account management properties as characteristics of defective Puppet scripts. Jha et al. [28] proposed a static analysis tool for detecting errors in configuration files of Android apps. In an exploratory empirical study, Cito et al. [29] assessed the quality of Docker configuration files on GitHub, observing that they violate 3.1 linter rules on average.

Recently, similar to our work, more studies have found that CI specifications [30] and processes [31] are susceptible to anti-patterns [23] 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 such as identifying reasons for build breakage in CI [8]. Techniques for automatically fixing build breakages have also been proposed [32–43]. Moreover, recent tools like CI-Odor [30] and CD-Linter [35] suggest fixes for common anti-patterns in CI/CD pipelines. We take a similar approach with our Hansel & Gretel tools (Chapter 5).

Another related concept is architectural or design smells. Marinescu [36] has defined detection strategies for capturing important flaws of object-oriented design that were reported in the literature. Garcia et al. [37] have defined architectural bad smells as architectural design decisions that negatively impact the understandability, testability, extensibility, and reusability of a software system. Moha et al. [38] define smells as poor solutions to recurring implementation and design problems. They also specify four well-known design smells and define their detection algorithms.

The anti-patterns that we propose in Chapter 5 of our thesis share similarities with configuration smells defined in prior work. For example, since externally-hosted scripts are not analyzed by the Travis CI runtime, anti-pattern 1 (redirecting scripts into interpreters) can lead to *non-deterministic errors* and *non-idempotence* problems that were identified by Shambaugh et al. [24]. Alicherry and Keromytis [39] showed that trusting SSH hosts keys (also known as trust-on-first-use) exposes hosts to man-in-the-middle attacks. Our anti-pattern 2 also detects instances where users bypass `ssh` security measures by disabling SSH host key checking. Similar to our work, Rahman et al. [40] have proposed Security Linter, a static analysis tool to identify seven security smells in Infrastructure as Code (IaC) scripts. Our anti-pattern 3 (using irrelevant properties) is similar to the *Invalid Property Value* and *Deprecated Statement Usage* configuration smells proposed by Sharma et al. [25] and the *Silent Failure* problem proposed by Shambaugh et al. [24]. Finally, our anti-pattern 4 (commands unrelated to the phase) is similar to Sharma et al.'s *Misplaced Attribute* and *Multifaceted Abstraction* configuration smells [25]. Indeed, if dependency installation, compilation, and testing commands are all included in the *Script* phase, the tasks in that phase are not cohesive, violating the single responsibility principle.

In summary, expanding on prior work in software configuration smells, we report on CI anti-patterns in Chapter 5 of our thesis. Our work provides value for software teams by helping

them to identify instances that can introduce build correctness, performance, and security problems during the CI process.

3.3 Build Breakage

Build breakage, which refers to the errors and failures that can occur during the process of creating executable software packages from source code, has attracted the attention of software engineering researchers at many occasions during the past decade.

The rate at which builds are broken has been explored in the past. Kerzazi et al. [41] have conducted an empirical study in a large software company analyzing 3,214 builds that were executed over a period of six months to measure the impact of build breakages, observing a build breakage rate of 17.9%, which generates an estimated cost of 904.64 to 2034.92 person hours. Seo et al. [6] studied nine months of build data at Google, finding that 29.7% and 37.4% of Java and C++ builds were broken, respectively. Tufano et al. [42] found only 38% of the change history of 100 subject systems is successfully compilable and that broken snapshots occur in 96% of the studied projects. Hassan et al. [43] showed that at least 57% of the broken builds from the top-200 Java projects on GitHub can be automatically resolved.

To better understand and predict build breakage, past studies have fit prediction models. Hassan and Zhang [44] have demonstrated that decision trees based on change and project attributes can be used to predict the certification result of a build. Wolf et al. [45] used a predictive model that leverages measures of developer communication networks to predict build breakage. Similarly, Kwan et al. [46] used measures of socio-technical congruence, i.e., the agreement of the coordination needs established by the technical domain with the actual coordination activities carried out by project members, to predict the build outcome in a globally distributed software team. In recent work, Luo et al. [12] have used the TravisTorrent dataset to predict the result of a build based on 27 features. They found that the number of commits in a build is the most important factor that can impact the build result. Dimitropoulos et al. [47] use the same dataset to study the factors that have the largest impact on build outcome based on K-means clustering and logistic regression.

For communicating the current status of the build, Downs et al. [48] proposed the use of ambient awareness technologies. They have observed by providing a separate, easily perceived communication channel distinct from the standard team workflow for communicating build status information, the total number of builds increased substantially, and the duration

of broken builds decreased. To help developers to debug build breakage, Vassallo et al. [32] propose a summarization technique to reduce the volume of build logs. For mitigating the impact of build breakage in the context of component-based software development, van der Storm [49] have shown how backtracking can be used to ensure that a working version is always available, even in the face of failure.

Broadly speaking, prior work has treated build breakage as a boolean, pass or fail label. In our work on noise and heterogeneity in CI build data, discussed in Chapter 6 of this thesis, we advocate for a more nuanced interpretation of build breakage that recognizes the noise in build outcome data and heterogeneity of build executions. Recently, complementing to our work, Zolfagharinia et al. [50] report on the CI build inflation in the Perl ecosystem using 30 million builds from the CPAN repository. Moreover, expanding on the theme, Ghaleb et al. [51] measure the impact of noises in build breakage data on modeling build breakages.

In summary, characterizing build outcome data helps software practitioners and researchers when building tools and proposing techniques to solve software engineering problems. While prior work makes important observations, understanding the nuances and complexities of build outcome data has not received sufficient attention by software engineering researchers. To support the interpretation of build outcome data, in Chapter 6 of our thesis, we characterize build outcome data according to harmful assumptions that one may make.

3.4 Analysis of CI Datasets

Researchers have used *TravisTorrent*,[1] a dataset of 2.6 million Travis CI builds curated by Beller et al. [52] to explore different aspects of CI such as noise in build outcomes [53] and long build durations [54]. Hilton et al. [2] also analyzed 1.5 million Travis CI builds to understand how developers use CI. Similarly, Durieux et al. [55] have curated a dataset of 35 million Travis CI jobs. Rausch et al. [56] analyzed Travis CI build failures from 14 open source Java projects and identified 14 error categories where test failures was the most common category. Their analysis shows that Perl packages may fail differently on different runtime environment and operating system combinations and therefore build results should be treated differently. Vassallo et al. [30] have studied build logs of popular Java projects on Travis CI to identify anti-patterns that reduce benefits of CI. Felidré et al. [57] have also used Travis CI build data to identify four bad practices in CI, namely: (1) infrequent commits, (2) poor test coverage, (3)

[1] `https://travistorrent.testroots.org/`

builds staying broken for long periods, and (4) builds taking too long to run. Zhao et al. [58] study the impact of CI on other software development practices by using GitHub projects that use Travis CI and report that more pull requests are successfully closed, even though the requests take longer to be closed with the introduction of CI.

The broad adoption of CI services has presented new opportunities for research on CI/CD. Researchers have interpreted the data generated by CI providers from the perspective of the CI users, discussing challenges and benefits of adopting CI. However, the perspective of the CI provider has remained largely unexplored. Improving the usability of the service by making use of research findings can help to attract new users and to retain existing ones. Therefore, in contrast to the existing literature, in Chapter 7 of this thesis we analyze CI data from the service provider's perspective and report on the CircleCI service broadening the focus to more ecosystems.

3.5 Slow CI/CD Feedback and its Remedies

In a recent literature survey, Widder et al. [59] 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 [58, 60], hindering the premise of rapid CI feedback [56]. Developers also complain about the cost of computational resources and the difficulty of debugging software with a slow CI cycle [17, 61].

Felidré et al. [57] 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 [17, 62] has advocated. Furthermore, 78% of the participants in the study by Hilton et al. [17] 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.

Due to its importance to practitioners, there have been several recent approaches proposed to tackle slow CI builds. Ghaleb et al. [54] 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. [63] use a timing-annotated build dependency graph to forecast build duration. Tufano et al. [64] 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 Chapter 8.

In prior work, several approaches have been proposed to reduce the time taken by CI builds. Abdalkareem et al. [65, 75] suggest to skip CI altogether for commits that do not affect source code. Esfahani et al. [67] describe the CloudBuild distributed build service, which uses content-based caching to skip build steps, saving time and compute resources at Microsoft. Li et al. [68] propose test case selection [69] 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., [19, 70, 80]).

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 specified build configuration files. The manual specification of build dependencies is error-prone [72–84]. Moreover, since organizations may already have non-trivial build specifications that were written for existing build tools, migrating to a new build tool requires a large investment of effort [76, 86]. Therefore, in Chapter 8, we strive to accelerate builds without relying on explicitly specified build dependencies. To simplify the adoption of our approach, we leverage existing CI pipeline specifications where available. Broadly speaking, we strive to deliver a language-agnostic solution so that existing code bases can immediately benefit from our approach with minimal investment of migration effort.

3.6 Continuous Deployment

Although not as common as continuous integration, there have been several studies on how continuous deployment services are designed and operated at scale. Schermann and Leitner [78] propose to use genetic algorithms for scheduling experiments when continuous deployment is practiced in a software organization. Similarly, Günalp et al. [79] present Rondo, a tool suite for continuous deployment of service-oriented applications, which aims for a deterministic and idempotent deployment process.

Going beyond the research experiments, practitioners also report on state-of-the-art DevOps platforms developed at large-scale software organizations to serve their internal use-cases. Esfahani et al. [67] describe how Microsoft's internal distributed build service was de-

signed to speed up the CI/CD workflow of their existing projects. Gupta et al. [80] report on how a large-scale online experimentation platform was designed at Microsoft to provide trustworthy results for internal users in their controlled experiments in a scalable manner. An experience report by Savor et al. [81] present observations from the adoption of a continuous deployment process for cloud-based software at Facebook and OANDA. Similarly, Rossi et al. [1] describe how continuous deployment is practiced during mobile software development at Facebook. Laukkanen et al. [82] surveyed the recent literature for the problems, causes, and solutions when adopting continuous delivery. They point out large commits, merge conflicts, broken builds, and slow integration approval as problems that are related to integration. By interviewing practitioners in 15 ICT companies, Leppänen et al. [83] found that domain-imposed restrictions, resistance to change, customer needs, and developers' skill and confidence are adoption obstacles for continuous deployment.

While prior work focuses on designing and maintaining deployment infrastructure for internal users, exposing the service to external users may also present unique challenges. Focusing on build data from the perspective of CI service providers is important for capacity planning and identifying opportunities to improve existing provider solutions. Identifying opportunities for managing resources efficiently will help CI service providers to keep operational costs low while delivering fast and reliable CI services. Therefore, in Chapter 7 of our thesis we look into the challenges in providing CI/CD services for external users.

Part II

Robustness in CI/CD Services

Chapter 4

Usage of Features in CI/CD Environments

Note. An earlier version of the work in this chapter appears in the *IEEE Transactions on Software Engineering (TSE)* journal [84].

4.1 Introduction

A typical CI service is composed of three types of nodes. First, *build job creation* nodes queue up new build jobs when configured build events occur, e.g., a new change appears in the project Version Control System (VCS). Next, a set of *build job processing* nodes process build jobs from the queue, adding job results to another queue. Finally, *build job reporting* nodes process job results, updating team members of the build status using web dashboards, emails, or other communication channels (e.g., Slack[1]).

In the past, organizations needed to provision, operate, and maintain the build job creation, processing, and reporting nodes themselves. To accomplish this, developers used general purpose scripting languages and automation tools (e.g., Ansible[2]). Since these general purpose tools are not aware of the phases in the CI process, boilerplate features such as progress tracking, error handling, and notification were repeated for each project. Dedicated CI tools such

[1] https://slack.com/
[2] https://www.ansible.com/

as Bamboo,[3] Jenkins,[4] and TeamCity[5] emerged to provide basic CI functionality; however, these CI tools still require that infrastructure is internally operated and maintained.

Nowadays, cloud-based providers such as Travis CI,[6] offer hosted CI services to software projects. Users of these CI services inform the service provider about how build jobs should be processed using a configuration file. This file specifies the tools that are needed during the build job process and the order in which these tools must be invoked to perform build jobs in a repeatable manner.

Like other software artifacts, this CI configuration file is stored in the VCS of the project. Like programming languages, configuration languages also offer features, which can be used or misused. For example, Travis CI users can *use* features like `branches`, which specifies which VCS branches to monitor for commit activity. Commits that appear on the monitored branches will trigger build jobs.

In this chapter, we set out to study how features in CI configuration files are being used. While the most popular CI service might differ from one source code hosting platform to another, prior work has shown that Travis CI is the most popular CI service on GitHub [2], accounting for roughly 50% of the market share.[7] Thus, we begin by selecting a corpus of 9,312 open source projects that are hosted on GitHub and have adopted the popular Travis CI service. Through empirical analysis of the CI configuration files of the studied projects, we address the following research questions about feature usage:

- **RQ1** *What are the commonly used languages in Travis CI projects?*
  Despite being the default Travis CI language, Ruby is only the sixth most popular language in our data set. Node.js is the most popular language in our corpus.
- **RQ2** *How are statements in CI specifications distributed among different sections?*
  We find that 48.16% of the studied Travis CI configuration code applies to *build job processing* nodes. Explicit deployment code is rare (2%). This shows that although the developers are using tools to integrate changes into their repositories, they rarely use these tools to implement *continuous delivery* [10]—the process of automatically releasing code that integrates cleanly.

[3] `https://www.atlassian.com/software/bamboo`
[4] `https://jenkins.io/`
[5] `https://www.jetbrains.com/teamcity/`
[6] `https://travis-ci.com/`
[7] `https://github.com/blog/2463-github-welcomes-all-ci-tools`

- **RQ3** *Which sections in the CI specifications induce the most churn?*
  Most CI configuration files, once committed, rarely change. The sections that are related to the configuration of *job processing nodes* account for the most modifications. In the projects that are modified, all sections are likely to be modified an equal number of times. Similar to RQ2, this again suggests that deployment-related features in CI tools are not being used.

Our study of CI feature usage leads us to conclude that future CI research and tooling would have the most immediate impact if it targets the configuration of job processing nodes.

The remainder of the chapter is organized as follows. Section 4.2 outlines the motivation of our study of CI feature usage. Section 4.3 outlines the design of our study of CI feature usage, while Section 4.4 presents the results. Section 4.5 discusses the threats to the validity of our study. Finally, Section 4.6 draws conclusions.

4.2 CI/CD Feature Usage

As a community, knowing how CI is being used in reality is important for several reasons. First, CI service providers will be able to make data-driven decisions about how to evolve their products, e.g., where to focus feature development to maximize (or minimize) impact. Second, researchers will be able to target elements of CI that are of greater impact to users of CI. Finally, individuals and companies who provide products and services that depend on or are related to CI (such as Hansel & Gretel) will be able to tailor their solutions to fit the needs of target users.

Hilton et al. [2] analyzed a broad spectrum of properties of CI specifications. We aim to complement the prior work by studying how features within CI specifications are used to configure their build nodes and jobs. To do so, we conduct an empirical study of 9,312 GitHub projects that use Travis CI, addressing the following research questions:

- **RQ1** *What are the commonly used languages in Travis CI projects?*
  We first aim to understand whether projects that are developed in certain languages are more common among the Travis CI user base. This will help future tool developers and researchers studying CI processes to identify potential target languages and technologies. This will also help to identify programming language features that can impact CI/CD configuration and adoption.

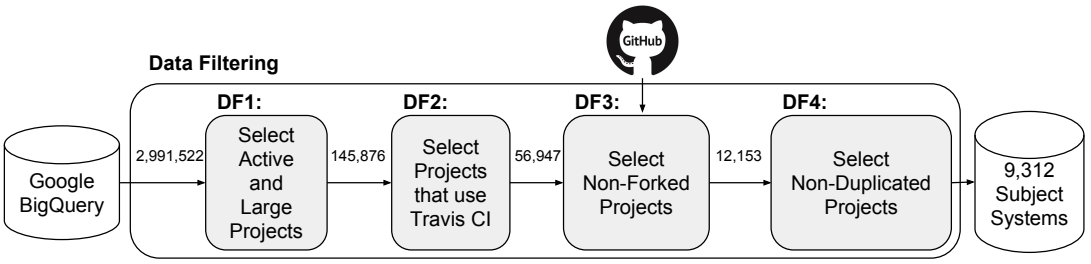


Figure 4.1: An overview of our data filtering approach.

- **RQ2** *How are statements in CI specifications distributed among different sections?*
  To develop an understanding of the spread of CI configuration code across sections, we are interested in the quantity of code that appears within each section.
- **RQ3** *Which sections in the CI specifications induce the most churn?*
  While RQ2 provides a high-level view of which sections in CI specifications require the most code, it does not help in understanding which sections require the most change. To complete the picture, we set out to study how the churn is dispersed among the sections.

4.3 Study Design

In this section, we provide our rationale for studying GitHub projects and explain our data filtering approach.

4.3.1 Corpus of Candidate Systems

In order to arrive at reliable conclusions, it is important to select a large and diverse set of software projects. With this in mind, we begin our analysis with systems that are hosted on the popular GitHub platform.

We start by querying the public GitHub dataset on Google BigQuery[8] for project activity (i.e., the number of commits) and project size heuristics (i.e., the number of files). This query returns 4,022,651,601 commits and 2,133,880,097 files spanning 2,991,522 GitHub repositories.

4.3.2 Data Filtering

While GitHub is a large corpus, it is known to contain projects that have not yet reached maturity [85]. To prevent the bulk of immature projects from impacting our conclusions, we

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

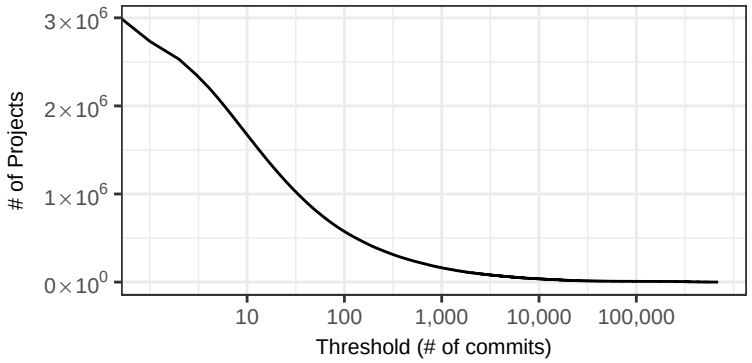


Figure 4.2: Threshold plot for commit activity.

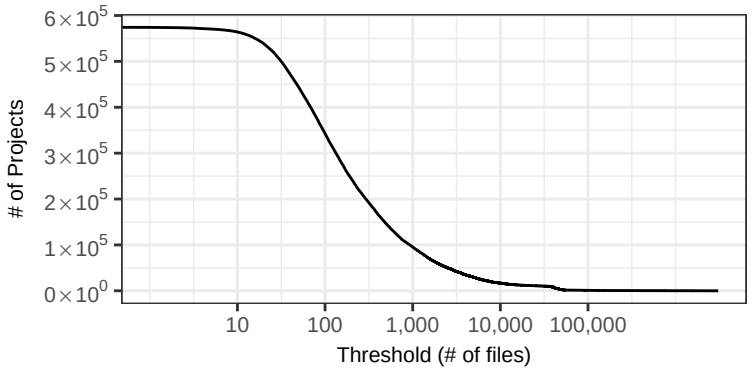


Figure 4.3: Threshold plot for project size.

first apply a set of filters to our GitHub data. Figure 4.1 provides an overview of our data filtering approach. We describe each step in the approach below.

DF1: Select Active and Large Projects

We first remove inactive projects from our corpus. To detect such projects, Figure 4.2 plots threshold values against the number of surviving systems. Selecting a threshold of 100 commits reduces the corpus to 574,325 projects.

Next, we remove small projects from our corpus. To detect such projects, Figure 4.3 again plots threshold values against the number of surviving systems. Selecting a threshold of 500 files further reduces the corpus to 145,876 projects.

DF2: Select Projects that use Travis CI

We focus our study on users of the Travis CI service for two reasons. First, while other CI services are available, Travis CI is the most popular, accounting for roughly 50% of the CI market on GitHub.[7] CircleCI ranks second with roughly 25%, while Jenkins (a CI tool rather than a service) ranks third with roughly 10%. Second, since other CI services have a similar configuration syntax (YAML-based DSL), it is likely that our observations will be applicable to other CI services. We elaborate on this in Section 5.5.4.

To identify GitHub projects that use Travis CI, we check for a `.travis.yml` configuration file in the root directory. This filter reduces the corpus to 56,947 projects.

DF3: Select Non-Forked Projects

Forking[9] allows GitHub users to duplicate a repository in order to make changes without affecting the original project. Developers working on forked repositories can submit *Pull Requests* to contribute changes to the original project.

Forks should not be analyzed individually, since they are primarily duplicates of the forked repository. If forks are not removed from the corpus, the same development activity will be counted multiple times. We detect forks using the GitHub API. Repositories that are flagged as forks according to this API are removed from our corpus. This filter reduces the corpus to 12,153 projects.

DF4: Select Non-Duplicated Projects

The DF3 filter only removes explicitly forked repositories that were created using the GitHub fork feature. Repositories may also be re-uploaded under a different owner and/or name without using the fork feature.

To detect these duplicated repositories, we extract the list of commit hashes (SHAs) in each of the candidate repositories that survive the prior filters. If any two repositories share more than 70% of the same commit SHAs, we label both repositories as duplicates. Since we cannot automatically detect which of the duplicated repositories is the original repository and which ones are the copies, we remove all duplicated repositories from our corpus. 9,312 candi-

[9] `https://help.github.com/articles/fork-a-repo/`

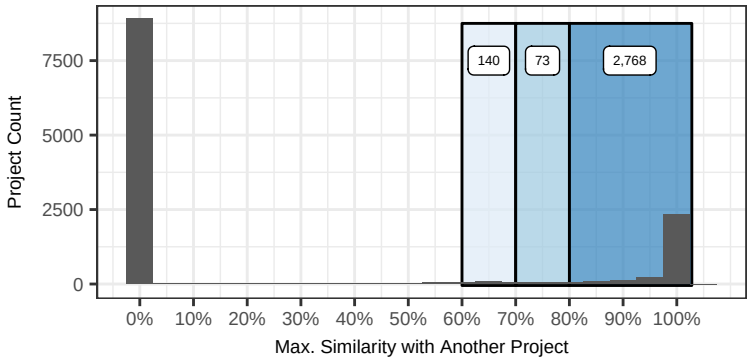


Figure 4.4: A histogram of the maximum commit similarity among the candidate repositories.

date repositories survive this final filter and are selected as subject systems for the following analyses.

To check if the selected similarity threshold for filtering out duplicated projects is suitable, for each project that survives the DF1–DF3 filters, we compute all pairwise commit similarity percentages. Then, for each project, we select the maximum similarity percentage. Figure 4.4 shows the histogram of these maximum similarity percentages. We observe a largely bimodal distribution where many projects are either distinct (similarity = 0%) or almost identical to another project in terms of commit SHAs (similarity ≈ 100%). Indeed, a more stringent 60% threshold only removes 140 more projects (1.50%) and a more lenient threshold of 80% only adds 73 projects (0.78%), indicating that sample does not depend heavily upon the threshold value.

4.3.3 Domain of the Subject Systems

To understand the domain of subject systems, we need to classify each subject system by inspecting their source and documentation. Since this is impractical in our context, we analyze a randomly selected subset of 152 subject systems. Table 4.1 shows that our corpus contains a broad variety of subject systems, including games, and web and mobile apps. Projects that have the functionality of multiple domains (e.g., Development tools vs DevOps) have been categorized under the domain that is covered by the largest number of features in the project. Since we previously selected active and large projects for our analysis, in this sub-sample, the lower bounds for the number of commits and number of files thresholds are 111 and 503, respectively.

Table 4.1: Domains in a sub-sample of our subject systems.

Type	# Projects	Percentage
Web Application	23	15.13
Graphics/Visualization	21	13.82
Application Framework/Library	15	9.87
Development Tools	15	9.87
Communication/Collaboration Tool	13	8.55
DevOps	10	6.58
Scientific Computing	10	6.58
Games/Game Engine	8	5.26
Mobile Application	7	4.61
Other	30	19.74
Total	**152**	**100.00**

4.4 Study Results

In this section, we present the results of our CI usage study with respect to our three research questions. For each research question, we first present our approach for addressing it, followed by the results that we observe.

(RQ1) What are the commonly used languages in Travis CI projects?

Approach. We identify the commonly used languages in Travis CI projects by detecting the setting of the *language* property in the Travis CI configuration file.

Results. Table 4.2 shows the ten most popular languages in our corpus of studied projects. Hilton et al. [2] explored the rate at which users of particular languages adopt CI, observing higher rates of adoption in projects that are primarily implemented using dynamic languages. Six of the top ten languages with the highest rates of CI adoption [2] appear in our list, i.e., JavaScript (Node.js in our setting), Ruby, Go, Python, PHP, and C++. The four languages from the Hilton et al. setting that do not appear in our sample (i.e., Scala, CoffeeScript, Clojure, and Emacs Lisp) are infrequently used, altogether appearing in 5.8% of the projects in the top ten languages in their setting.

When compared with the language statistics released by GitHub,[10] we find nine of our top ten languages are among the ten most popular languages on GitHub (by opened pull

[10] `https://octoverse.github.com/`

Table 4.2: CI usage by programming language.

Language	# Projects	%
Node.js	1,460	15.68
Java	1,337	14.36
PHP	1,163	12.49
Python	1,122	12.05
C++	995	10.69
Ruby	811	8.71
C	702	7.54
Go	290	3.11
Objective-C	250	2.68
Android	195	2.09
Other	987	10.60

requests). Android does not appear in the list by GitHub because applications for the Android platform are developed in Java programming language and therefore grouped with Java projects. C# appears in GitHub's top ten, but not ours. Although not shown, C# appears in 149 projects, and would rank eleventh.

***Observation 1**: Despite being the default Travis CI language, Ruby is not the most popular language in our corpus of studied systems.* Table 4.2 shows that 811 projects are labelled explicitly as Ruby projects, making Ruby the sixth ranked language in our corpus. There are an additional 421 projects that do not specify a *language* property. In this case, the Travis CI execution environment assumes that the project is using Ruby. Even if all 421 of these unlabelled projects are indeed Ruby projects, this would only increase the Ruby project count to 1,232, which would rank third.

***Observation 2**: Node.js is the most popular language in our corpus of studied systems.* Table 4.2 shows that there are 1,460 projects (16%) that are labelled explicitly as Node.js projects in our corpus. Our study is not the only context in which the popularity of Node.js has been observed. For example, according to a recent StackOverflow survey[11] of 64,000 developers, Node.js was the most commonly used framework. Moreover, the recent *left-pad* debacle, where the removal of an npm package for left-padding strings had a ripple effect that crippled several popular e-commerce websites,[12] highlights the pivotal role that Node.js plays in the development stacks of several prominent web applications.

[11] `http://stackoverflow.com/insights/survey/2017`

[12] `https://www.theregister.co.uk/2016/03/23/npm_left_pad_chaos/`

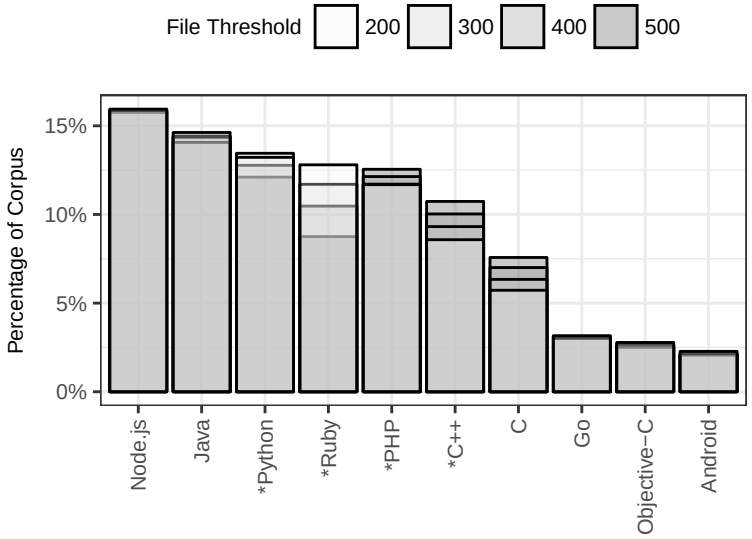


Figure 4.5: The percentage of the corpus that uses the ten most popular languages. Asterisks (*) denote languages that change ranks when the file count threshold changes (DF1).

Since some languages may require more files than others, we repeat our analysis with four file count threshold values (DF1). Figure 4.5 shows that while the third through sixth ranked languages vary, six ranks are resilient to threshold changes and Node.js remains the most popular language.

Summary: *Although Ruby is the default language in Travis CI, Node.js is more popular in our sample.*
Implications: *Since language popularity fluctuates, CI service providers should carefully consider whether a popular language of the day should be implicit when no language is declared explicitly.*

(RQ2) How are statements in CI specifications distributed among different sections?

Approach. To answer this research question, we first label each property in the `.travis.yml` file as related to CI node configuration or build process configuration. The tags that specify the phases in the CI process are labelled as build process configuration. The tags that are related to CI node configuration are further divided into four sub-categories depending on the type of CI nodes that are being configured, i.e., build job creation, build job processing,

Table 4.3: The identified build node configuration tags.

Sub-category	**Key**
Creation	`branches`
Processing	`addons, android, bundler_args, compiler, cran, d, dart, dist, dotnet, elixir, env, gemfile, ghc, git, go, haxe, jdk, julia, language, lein, matrix, mono, node, node_js, nodejs, os, osx_image, otp_release, perl, php, podfile, python, r, r_binary_packages, r_build_args, r_check_args, r_github_packages, r_packages, repos, ruby, rust, rvm, sbt_args, scala, services, smalltalk, solution, sudo, virtualenv, warnings_are_errors, with_content_shell, xcode_scheme, xcode_sdk, xcode_workspace, xcode_project`
Notification	`notifications`
Other	`before_cache, cache, group, source_key`

build status notification, or other. Table 4.3 shows our mapping of `.travis.yml` tags to these subcategories. There is a large number of processing node configuration tags compared to other subcategories to support configuration options specific to various language tool chains.

We then parse the `.travis.yml` files of our subject systems. We use the parsed output to count lines in each of the sections of each file. Finally, we apply the Scott-Knott Effect Size Difference (ESD) test [86]—an enhancement to the Scott-Knott test [87], which also considers the effect size when clustering CI sections into statistically distinct ranks.

Results. Table 4.4 shows the popularity of the sections, as well as their overall length and proportion within the corpus. Tags belonging to some categories do not appear in the config-

Table 4.4: The popularity of `.travis.yml` sections, as well as their length and proportion of lines in our corpus.

	Section	# Projects	# lines	% lines
CI Node Config.	creation	1,441	2,236	1.45
	processing	8,852	74,285	48.16
	reporting	2,914	7,361	4.77
	other	1,836	3,500	2.27
Build Process Config.	before_install	3,551	14,452	9.37
	install	3,519	11,895	7.71
	before_script	3,863	14,597	9.46
	script	7,122	18,972	12.30
	after_script	626	1,111	0.72
	before_deploy	115	362	0.23
	deploy	343	2,918	1.89
	after_deploy	23	41	0.03
	after_failure	223	391	0.25
	after_success	1,243	2,113	1.37

uration of all projects. In those situations, the default behaviour for that tag is performed. For example, `branches` tag related to job creation configuration is only specified in 1,441 projects. In projects that the `branches` tag is not specified, build jobs are created for all branches except the `gh-pages` branch by default.

Observation 3*: For CI node configuration, sections that are related to* job processing *nodes appear in the most projects.* Table 4.4 shows that 8,852 (95.06%) of the studied `.travis.yml` files include job processing. Moreover, 48.16% of the CI code is in the job processing node configuration section.

Observation 4*: For build process configuration, sections that are related to the* `script` *phase appeared in the most projects.* Table 4.4 shows that 7,122 (76.48%) of the studied projects have `script` commands in their `.travis.yml` files. Moreover, 12.30% of the CI code appears in the `script` phase.

Observation 5*:* Job processing *configuration and* `script` *phase configuration appear in statistically distinct ranks when compared to other sections.* Figure 4.6a shows the distribution of commands in each section. The sections are ordered according to the ranks from the Scott-Knott ESD test. For example, the *jruby/activerecord-jdbc-adapter* project,[13] a database adapter

[13] `https://github.com/jruby/activerecord-jdbc-adapter`

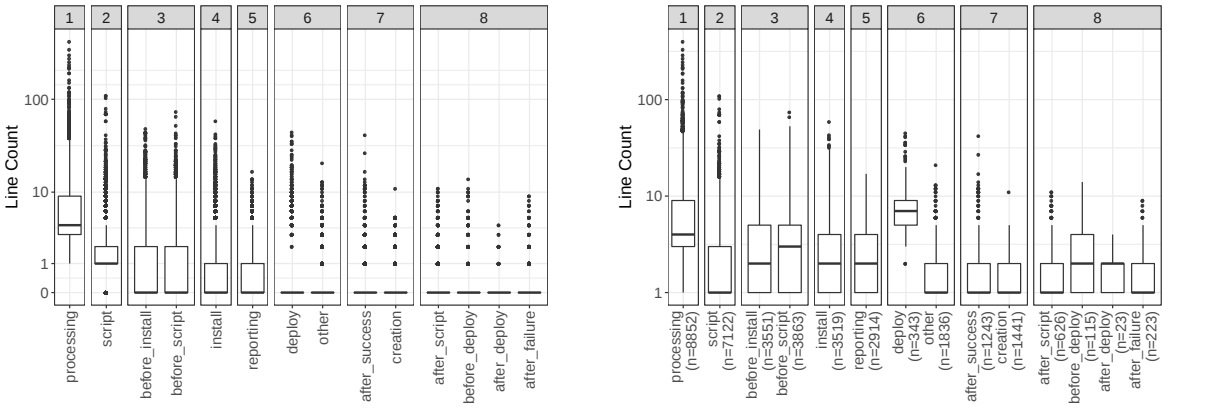
for Ruby on Rails, uses 400 lines for *job processing* configuration. Most of the lines in this case are used for specifying different JDK and JRuby version combinations to be installed on the job processing nodes. Moreover, in the *joshuarowley42/BigBox-Pro-Marlinr* project,[14] (3D printer firmware) 109 lines appear in the *script* phase.

***Observation 6**: Although the `deploy` phase only appears in 343 (4%) of all projects, the median number of commands is high when compared to other sections.* Since it is not mandatory to specify commands for all of the sections, it is rare that all valid sections appear in any given configuration file. Figure 4.6b shows the distribution of commands after removing zero-length sections. The difference in the `deploy` phases in Figure 4.6a (with zeros) and Figure 4.6b (without zeros) is striking. It appears that when the `deploy` phase is included, it tends to require plenty of `.travis.yml` configuration code. For example, the *oden-lang/oden* project[15] requires 42 lines of code to describe their deployment process. These lines of code describe how to deploy the release artifacts for a specific release and the current commit on the `master` branch to Amazon S3. Indeed, it may be the case that organizations avoid using `deploy` phase features because it requires lengthy and complex configuration.

The `.travis.yml` file supports configuration of deployment to many popular cloud services including AWS, Azure, Google App Engine, and Heroku. So it is unlikely that the reason for developers not using Travis CI for deployment is lack of platform support. Since Ansible is a popular tool used by developers for the automation of deployments, we study the use of Ansible as an alternative to the deployment features of Travis CI in our corpus by searching for syntactically valid Ansible playbooks. Unfortunately, we find only 109 (1%) projects where Ansible is being used. Further studies are needed to identify why deployment features of Travis CI are rarely used.

Summary: *Although code for configuring job processing nodes is most common (48.16%), and deployment code is rare (1.89%), when present, deployment code accounts for a large proportion of the CI specification.*
Implications: *Research and tooling for CI configuration would have the most immediate impact if it were focused on supporting the configuration of job processing nodes or reducing the complexity of deployment configuration.*

[14] `https://github.com/joshuarowley42/BigBox-Pro-Marlin`
[15] `https://github.com/oden-lang/oden`



(a) The distribution for all projects.

(b) The distribution after removing zero-length sections.

Figure 4.6: Line counts in each section of the `.travis.yml` file.

(RQ3) Which sections in the CI specifications induce the most churn?

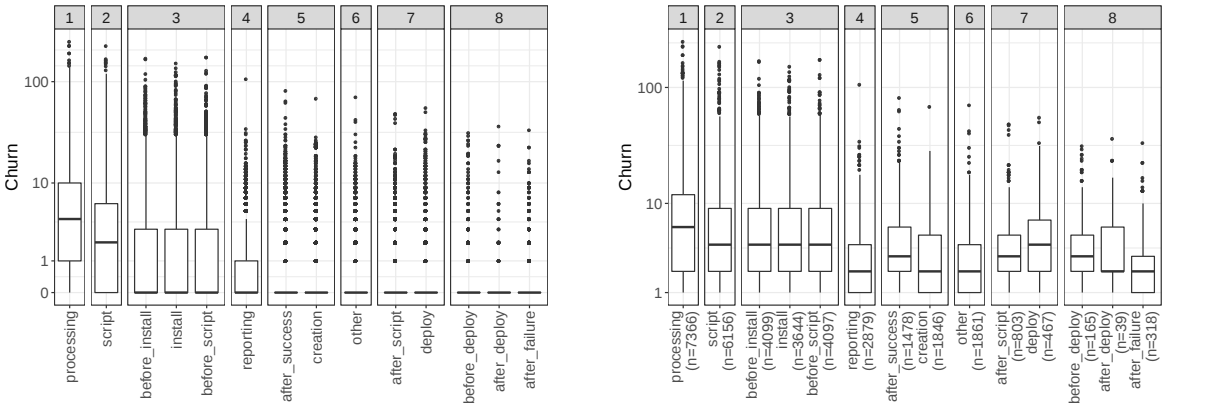
Approach. First, we count the number of commits that have modified the `.travis.yml` file of each project. Then, using the line-to-section mapping (see Section 2), we attribute changed (i.e., added and/or removed) lines to sections in the file that have been modified by each of these changes. Finally, we apply the Scott-Knott ESD test to split the sections into statistically distinct ranks.

Results. Figure 4.7a shows the churn (i.e., the degree to which a given section in the `.travis.yml` file has changed over time). The `.travis.yml` files in the subject systems are modified 18.06 times on average in their lifetime, with a maximum of 366 changes in the *lolli42/TYPO3.CMS-Catharsis* project;[16] however, the churn in each section is very low. In more than 75% of the studied projects, configuration sections are modified fewer than 10 times. These results complement the work of Hilton et al. [2], who observed similar overall trends in the rates of change in `.travis.yml` files (median of 12, maximum of 266).

***Observation 7**: The sections that are related to* job processing *node configuration account for the most modifications over time.* For example, *TechEmpower/FrameworkBenchmarks*,[17] a project that provides performance benchmarks for web application frameworks, has 290 modifications to its `.travis.yml` file, of which, 242 modify its *job processing* node configuration. In this case, it is because the benchmarks are contributed by the developer community and the benchmarks for each framework require job processing nodes to be configured differently.

[16] `https://github.com/lolli42/TYPO3.CMS-Catharsis`

[17] `https://github.com/TechEmpower/FrameworkBenchmarks`



(a) The distribution for all projects.

(b) The distribution after removing zero-length sections.

Figure 4.7: The churn of each section in the `.travis.yml` file.

This complements our earlier observation that most of the effort in configuring CI is spent on the processing node configuration.

Hilton et al. [2] studied the frequency of reasons for CI changes and observed different rankings than those that we observe. This discrepancy is likely due to differences in the granularity of our analyses. For example, in our analysis, we study the distribution of project-specific rates of change, while their analysis uses a single measurement of the overall rates of change for each identified reason. Nonetheless, there are similarities in our rankings. For example, their top ranked reason for CI change is related to the build matrix, which is a subset of our top ranked job processing section.

***Observation 8**: In the projects that are modified, all sections are likely to be modified an equal number of times.* Since Figure 4.7a shows that sections after the fourth rank are not modified in most of the projects (i.e. the median churn of these sections is 0.), we omit such projects in the next box plot shown in Figure 4.7b. Here, we can observe that the median churn for all of the sections is in the range of 1–10.

Summary: *In 75% of the studied configurations, sections of `.travis.yml` files are modified fewer than ten times.*
Implications: *Research and tooling for CI configuration should focus on the creation of an initial specification rather than supporting specification maintenance.*

4.5 Threats to Validity

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

4.5.1 Internal Validity

Projects may use Travis CI without a `.travis.yml` file. In this case, the Travis CI runtime assumes that the project is using Ruby and would apply the conventional Ruby CI process. Since we are unable to identify such projects automatically, we only consider projects with a `.travis.yml` file in the root directory of the project. Some subject systems may use multiple CI/CD services or could be migrating across different CI/CD services at the time of analysis. Therefore, studying only one CI service usage of such projects may provide an incomplete picture of their CI process. Further studies are needed to understand the CI usage of projects with multiple CI/CD service configurations.

4.5.2 External Validity

Threats to external validity are concerned with the generalizability of our findings. We focus only on open source subject systems, which are hosted on GitHub and use Travis CI as the CI service provider. However, since GitHub is one of the most popular hosting platforms for open source software projects and Travis CI is the most widely adopted CI service among open source projects [2], our findings are applicable to a large proportion of open source projects. Nonetheless, replication studies using other CI services may yield further insight.

4.5.3 Construct Validity

CI usage statistics are computed using various scripts that we have written. These scripts may themselves contain defects, which would affect our results. To address this threat, we test our tools and scripts on subsamples of our datasets, and manually verify the results.

The filters that we apply to remove small, inactive, and duplicated repositories from our corpus are based on thresholds, i.e., project size in files, project activity in commits, and rate of duplication in percentage of duplicated commits. The specific threshold values that we selected may impact our observations. With this in mind, we did not select threshold values arbitrarily. First, we analyze threshold plots to understand the impact that various threshold

values will have on the number of retained systems. Second, we perform sensitivity analyses (Figures 4.4 and 4.5), where the impact of selecting different thresholds is shown to be minimal.

4.6 Chapter Summary

CI has become a widely used practice among many software teams today. A CI service typically consists of nodes for creating, processing, and reporting of build jobs. To mitigate the overhead of maintaining and operating this infrastructure themselves, many organizations are moving to cloud-based CI services. These services allow for customizing the CI process using configuration files.

Through our study of 9,312 open source systems that use Travis CI, we make the following observations about the use of CI specifications:

- Despite being the default Travis CI language, Ruby is not the most popular language in our corpus of studied systems. Node.js is the most popular language in our corpus of studied systems (Observations 1 & 2).
- In terms of CI node configuration, sections that are related to *job processing* nodes appear in the most projects, while for build process configuration, sections that are related to the `script` phase appear in the most projects (Observations 3 & 4).
- *Job processing* configuration and `script` phase configuration have statistically distinct and higher ranks in projects compared to other sections (Observation 5).
- Although commands in the `deploy` phase appear only in 343 projects (3.68%), the median number of commands is comparable to other sections. (Observation 6)
- The CI code that configures job processing nodes accounts for the most modifications. In the projects that are modified, all sections are likely to be modified an equal number of times (Observations 7 & 8).

Our CI usage results suggest that the most natural direction for future research and tooling in CI would target the configuration of job processing nodes.

Chapter 5

Misuse of Features in CI/CD Environments

Note. An earlier version of the work in this chapter appears in the *IEEE Transactions on Software Engineering (TSE)* journal [84].

5.1 Introduction

Like other software artifacts, CI configuration file is stored in the VCS of the project. Since the build process that is being invoked tends to evolve [88, 98], this CI configuration file must also evolve to keep pace. Indeed, CI configuration code may degrade in quality and may accrue technical debt if it is not maintained properly. Like programming languages, CI configuration can also be *misused*, e.g., when unsupported or deprecated commands are used.

In this chapter, we set out to study how CI features are being misused. To study misuse, we define four anti-patterns based on formal Travis CI documentation,[1] and informal documentation from the Travis CI user community: (1) redirecting scripts into interpreters (e.g., `curl https://install.sandstorm.io|bash`); (2) bypassing security checks (e.g., setting the `ssh_known_hosts` property to unsafe values); (3) using irrelevant properties; and (4) using commands in an incorrect phase (e.g., using *install* phase commands in the *script* phase). Using Hansel—our tool for detecting anti-patterns in `.travis.yml` files—we address the following research question:

[1] `https://docs.travis-ci.com/`

- **RQ1** *How prevalent are anti-patterns in CI specifications?*
  Hansel detects at least one anti-pattern in the CI specifications of 894 projects in the corpus (9.60%), and achieves a recall of 82.76% in a sample of 100 projects.

Using Gretel—our anti-pattern removal tool for CI configuration code—we address the following research questions:

- **RQ2** *Can anti-patterns in CI specifications be removed automatically?*
  Yes, Gretel can remove the detected cases of the most frequent anti-pattern automatically with a precision of 69.60%. This increases to 97.20% if a post hoc manual inspection phase is included.
- **RQ3** *Are automatic removals of CI anti-patterns accepted by developers?*
  Yes, we submitted 174 pull requests that contain Gretel-generated fixes, of which, developers have: (1) responded to 49 (response rate of 28.16%); and (2) accepted 36 (20.69% of submitted pull requests and 73.47% of pull requests with responses).

Our study of misuse of CI shows that anti-patterns that threaten the correctness, performance, and security of build jobs are impacting a considerable proportion of Travis CI users (9.60%). Hansel and Gretel can detect and remove these anti-patterns accurately, allowing teams to mitigate or avoid the consequences of misusing CI features.

The remainder of the chapter is organized as follows. Sections 5.2 and 5.3 outline the motivation for and design of our study of CI misuse, respectively, while Section 5.4 presents the results. Section 5.5 discusses the broader implications of our results. Section 5.6 discloses the threats to the validity of our study. Finally, Section 5.7 draws conclusions.

5.2 Anti-patterns in CI/CD Specifications

If improperly configured, Travis CI build jobs may have unintended behaviour, resulting in broken or incorrect builds. Violating the semantics of CI specifications could also introduce maintenance and comprehensibility problems. Furthermore, the Travis CI runtime environment may be unable to optimize provisioning of CI job processing nodes for specifications where semantics are violated.

To help Travis CI users avoid common pitfalls, the Travis CI team provides TravisLint,[2] an open source tool that scans `.travis.yml` files for mistakes (e.g., YAML formatting issues,

[2] `https://github.com/travis-ci/travis.rb#lint`

missing mandatory fields). If the issues are fixed, TravisLint can prevent configuration errors from breaking project builds.

5.2.1 Research Questions

The `.travis.yml` files that are syntactically valid can still violate the semantics of Travis CI and introduce build correctness, performance, and security problems.

To detect such semantic violations, we propose Hansel—a `.travis.yml` anti-pattern detector. Then, we also propose Gretel—a tool for removing anti-patterns from `.travis.yml` files. We apply Hansel and Gretel to the 9,312 `.travis.yml` files in our corpus in order to address the following research questions:

- **RQ1** *How prevalent are anti-patterns in CI specifications?*
  In this research question, we aim to study what type of CI anti-patterns are commonly occurring in software projects "in the wild".
- **RQ2** *Can anti-patterns in CI specifications be removed automatically?*
  This research question explores whether the detected anti-patterns can be fixed automatically and to what degree are the transformed files still valid.
- **RQ3** *Are automatic removals of CI anti-patterns accepted by developers?*
  This research question explores whether our anti-pattern detection technique is useful for real developers in practice. If developers accept our fixes and integrate them into their projects, it would suggest that our findings are useful to some degree.

5.3 Study Design

We implement Hansel to detect anti-patterns and Gretel to remove them. In a nutshell, Hansel parses a `.travis.yml` file using YAML and bashlex parsers in order to detect anti-patterns. Then, Gretel applies the ruamel.yaml serialization/deserialization framework[3] to remove the detected anti-patterns automatically.

We define CI specification anti-patterns as violations of best practices in CI configuration files that could hinder the correctness, performance, or security of the CI process of a software system. Similar to the approach followed by prior work [25, 28], we first read the rules implemented by TravisLint,[2] formal Travis CI documentation,[1] informal documentation from the

[3] `https://pypi.python.org/pypi/ruamel.yaml`

Travis CI user community (e.g., blogs, posts on Q&A sites such as StackOverflow) and inspect a sample of artifacts (i.e., `.travis.yml` files) to prepare a list of recommended best practices. Then, we group the related best practices and deduce the corresponding anti-patterns (i.e., cases where best practice are being violated).

Below, for each anti-pattern, we present our rationale for labelling it as an anti-pattern, and the approach that (1) Hansel uses to detect it and (2) Gretel uses to remove it.

Anti-pattern 1: Redirecting Scripts into Interpreters

Motivation. A common approach to software package installation is to download a script from a hardcoded URL and pipe it into a shell interpreter. For example, the installation instructions for the *Sandstorm* package,[4] a self-hostable web productivity suite, includes a shell command: `curl https://install.sandstorm.io|bash`. While this installation procedure is convenient, it is known to be susceptible to security vulnerabilities.[5] Moreover, if a network failure occurs during the execution of the `curl` command, the installation script may only be partially executed.

Detection. In order to detect this anti-pattern, we follow a three-step approach. First, we parse the `.travis.yml` file to identify commands that contain a pipe. Next, those commands are split into the pre- and post-pipe sub-commands using the `bashlex` library. We check the pre-pipe command for known downloaders (i.e, `wget`, `curl`). We then check the post-pipe command for known shell interpreters (i.e., `sh`, `bash`, `node`). If both of these conditions are met, we identify the command as an instance of this anti-pattern.

Removal. CI specifications should verify the integrity of externally hosted scripts before executing them. This could be achieved by automatically verifying the script after downloading it but before execution. Alternatively, one could download the installation scripts, verify their integrity, and commit known-to-be secure versions to the VCS. Since either solution requires changes that are beyond the scope of the `.travis.yml` file, we have not implemented an automatic removal for this anti-pattern in Gretel yet.

[4] `https://sandstorm.io/install`

[5] `https://www.idontplaydarts.com/2016/04/detecting-curl-pipe-bash-server-side/`

Anti-pattern 2: Bypassing Security Checks

Motivation. During the CI process, if the Travis CI job processing node communicates with other servers via SSH for transferring artifacts, it is important to have this connection be configured securely. A misconfigured connection can make job processing node(s) vulnerable to network attacks. For example, using the `ssh_known_hosts` property in the `addons` section of the `.travis.yml` file exposes job processing nodes to man-in-the-middle attacks.[6,7]

Detection. We parse `.travis.yml` files and check whether they satisfies at least one of the following conditions:

- There exists an `addons` section, which contains an `ssh_known_hosts` property.
- There exists a command containing the line `StrictHostKeyChecking=no`.
- There exists a command containing the line `UserKnownHostsFile=/dev/null`.

Removal. To remove this anti-pattern, three steps should be followed. First, all of the vulnerability-inducing lines (`ssh_known_hosts`, `StrictHostKeyChecking=no`, `UserKnownHostsFile=/dev/null`) should be removed from the `.travis.yml` file. Second, a `known_hosts` resource should be created in the repository and the argument `-o UserKnownHostsFile=known_hosts` should be provided whenever `ssh` is invoked.

Anti-pattern 3: Using Irrelevant Properties

Motivation. Travis CI users may specify properties in the `.travis.yml` file that are not used by Travis CI at runtime. These properties may be user mistakes (e.g., typos) or features that Travis CI has later deprecated and/or retired. For example, the `.travis.yml` file may contain an *after_install* property; however, that property is not supported by Travis CI. This likely occurs because both *deploy* and *script* phases have post-execution clean-up phases (i.e., *after deploy, after script*), so Travis CI users may assume that the convention is followed by the *install* phase without carefully checking the user documentation.

The dangerous consequence of specifying irrelevant properties is that the Travis CI environment ignores unsupported properties. While the omission of the unsupported property is logged as a warning, it is unlikely that developers will check these warning if build jobs are successful.

[6] `https://annevankesteren.nl/2017/01/secure-secure-shell`
[7] `https://docs.travis-ci.com/user/ssh-known-hosts/#Security-Implications`

Detection. In Section 4.4, we have mapped valid properties to sections in the `.travis.yml` file. We detect instances of anti-pattern 3 by parsing the `.travis.yml` file and checking whether each property appears in its mapped section. Unrecognized properties are reported as anti-patterns.

Removal. There are several ways to fix this anti-pattern. First, if the unrecognized property is `after_install`, GRETEL removes the `after_install` phase in the configuration and moves its commands to the end of the `install` phase. If the `install` phase does not exist, it is created and, to preserve the pre-existing behaviour, the default commands are added (e.g., `npm install` for NODE.JS projects) before appending the `after_install` content.

Second, if the unrecognized property is similar to a recognized one (i.e., with a Levenshtein distance close to zero), the unrecognized property is corrected. For example, *before_srcipt* will be corrected to *before_script*.

In other cases, GRETEL warns the user that the unrecognized properties will be ignored by the TRAVIS CI runtime.

Anti-pattern 4: Commands Unrelated to the Phase

Motivation. Violating the semantics of a phase by including unrelated commands can introduce maintenance problems. If the commonly-accepted phases are not used for the intended purpose, new members of the project will find it difficult to understand the behaviour of the CI process. Moreover, the various runtime optimizations that TRAVIS CI performs in order to speed up builds (e.g., caching) may be suboptimal if phases are used in unintended ways.

Detection. We begin by identifying commands that we suspect should appear in a given phase. For example, in NODE.JS projects, we expect to find package installation commands such as `npm install` in the `install` phase (or one of its sub-phases) and testing framework commands such as `mocha` in the `script` phase (or one of its sub-phases).

Build tools vary based on programming language. For example, NODE.JS projects typically use `npm` for managing dependencies, whereas PYTHON projects typically use `pip`.

While we define this anti-pattern in language-agnostic terms, due to the plethora of language-specific tools, we must detect the anti-pattern in a language-aware manner. Since, according to the results of RQ1, NODE.JS is the most popular language among our subject systems, we prototype the detection of this anti-pattern for NODE.JS projects.

Table 5.1: Well-bounded commands at each phase.

Phase	Functionality	Command
Install	Install dependencies	`npm install, apt-get install, bower install, jspm, tsd`
Script	Testing	`npm test, mocha, jasmine-node, karma, selenium`
	Run Interpreter/Framework	`node, meteor, jekyll, cordova, ionic`
	Static Analysis	`codeclimate, istanbul, codecov, coveralls, jscover, audit-package`
Deploy	Deploying by script	`sh .*deploy.*.sh`

We consider instances of well-bounded commands that we find in other phases to be instances of this anti-pattern. Table 5.1 shows the well-bounded commands that we detect by analyzing the Node.js sample semi-automatically. To detect instances of this anti-pattern, we parse the `.travis.yml` file, associating commands with phases. If a well-bounded command from Table 5.1 is found outside of the phase to which it is bounded, we flag it as an anti-pattern.

Removal. To remove this anti-pattern automatically, we select the projects that have `install`-related commands in other phases (because we find that it is the most commonly occurring variant of this anti-pattern). Gretel removes these commands from non-`install` phases and appends them to the end of the `install` phase. If the project does not have an `install` phase, it is created and the default commands are added (to preserve the pre-existing behaviour) before appending the other commands.

5.4 Study Results

In this section, we present the results of our CI misuse study with respect to our three research questions.

(RQ1) How prevalent are anti-patterns in CI specifications?

***Observation 1**: 894 of the 9,312 subject systems (9.6%) have at least one anti-pattern in their CI specifications.* 862 of those (96%) have one type of anti-pattern. 31 of the remaining 32 projects have two types of anti-patterns. The *AngularjsRUS/angular-doc* project,[8] which provides the Russian version of the AngularJS documentation, has three types of anti-patterns (the only missing anti-pattern is #3).

***Observation 2**: In a sample of 100 random projects, Hansel achieves a recall of 82.76%.* To estimate the recall of Hansel, we manually identify anti-patterns in a randomly selected

[8] `https://github.com/AngularjsRUS/angular-doc`

sample of 100 `.travis.yml` files from our corpus. We apply HANSEL to the same sample, and compute the $recall = \frac{\#\ anti\text{-}patterns\ found\ by\ HANSEL}{\#\ anti\text{-}patterns\ found\ manually}$.

Our manual analysis uncovers 29 instances of the anti-patterns in the sample. HANSEL can detect 24 of these instances, achieving a recall of 82.76%. Three of the five false negatives are instances of anti-pattern 1 (redirecting scripts into interpreters), where the downloaded file is immediately piped into an extractor rather than an interpreter (e.g., `wget -O - <URL>|tar -xvJ`). These are borderline cases because the downloaded content is not being executed. However, content is still extracted without verifying its integrity. If we relax the interpreter requirement of HANSEL's detector for anti-pattern 1, the recall improves to 93.10% (27 of 29).

In the remaining two false negatives, HANSEL fails to find anti-pattern 4 (commands unrelated to the phase) where `composer.phar`, the dependency management tool for PHP, is used in the `before_script` phase. Our initial mapping of commands to phases did not bind the `composer` tool to the `install` phase (see Table 5.1). This can easily be remedied by adding the missing binding.

***Observation 3**: The majority of instances of anti-pattern 1 are installing the popular METEOR web framework.* We detect 206 instances where scripts are being downloaded and piped into shell interpreters directly, of which, 106 (51%) are in projects using NODE.JS. In these 106 projects, we find that 94 of them (88%) are using the above anti-pattern to install the METEOR web framework.[9] In fact, the METEOR documentation instructs users to install the framework using this method (`curl https://install.meteor.com|/bin/sh`).[10]

We reached out to the METEOR team to discuss the potential security implications of this installation approach. The METEOR team explained that the developer community is divided about using script redirection to install software packages. On the one hand, some have shown how script redirection can be exploited by attackers[5] or how networking interruptions during the download command may lead to partial execution of the installation script.[11] On the other hand, members of the SANDSTORM project defend script redirection for cases where script downloads are served strictly over HTTPS.[12] The SANDSTORM team argues that script redirection allows developers to iterate faster by avoiding the hassle of maintaining a variety of package formats for different platforms (e.g., `.rpm` and `.deb` for RedHat-type and Debian-

[9] `https://www.meteor.com`
[10] `https://www.meteor.com/install`
[11] `https://www.seancassidy.me/dont-pipe-to-your-shell.html`
[12] `https://sandstorm.io/news/2015-09-24-is-curl-bash-insecure-pgp-verified-install`

type Linux distributions, respectively). Moreover, discussion threads on HackerNews[13] argue that other standard package distribution methods (e.g., binary installers, package managers) are also susceptible to man-in-the-middle attacks unless the delivered packages are signed cryptographically. The Meteor team argue that they have not been able to identify a more secure alternative for the script redirection installation method.

If a project advocates for the script redirection installation method, we propose the following guidelines:

- The installation script should be served over HTTPS.
- The installation script should be made resilient to network interruptions by wrapping the core script behaviour in a function, which is invoked at the end of the script. Doing so will prevent partial execution of the script, since the interpreter will only execute the script instructions when the function is invoked at the end.
- Users should regularly audit the installation script.

However, when the project has identified the supported platforms or has accumulated several external dependencies, migration to a package manager may pay off.

***Observation 4**: Although rare, there are instances anti-pattern 2 in Travis CI specifications.* Hansel detects 63 instances of this anti-pattern in our corpus. In 37 (58.73%) of these cases, the `StrictHostKeyChecking=no` command is being used. This command disables an interactive prompt for permission to add the host server fingerprint to the `known_hosts` file. Developers may disable the prompt because it will impede cloning a repository via SSH in a headless environment, such as Travis CI, which can lead to build breakage. However, setting `StrictHostKeyChecking=no` exposes the host to man-in-the-middle attacks by skipping security checks in `ssh`.

In 18 instances (28.57%), the `ssh_known_hosts` property is set in the `addons` section to define host names or IP addresses of the servers to which Travis CI job processing nodes need to connect during the CI process. This is insecure because if the network is compromised (e.g., by DNS spoofing), Travis CI job processing nodes may connect and share private data with an attacker's machine.

In another eight instances of anti-pattern 2 (12.70%), `UserKnownHostsFile=/dev/null` is being used. In this case, host server fingerprints are written to and read from an empty file, effectively disabling host key checking, and exposing the host to man-in-the-middle attacks.

[13] `https://news.ycombinator.com/item?id=12766049`

The secure way to prevent the interactive prompt from interrupting scripted operations is to store the private keys of the hosts that Travis CI job processing nodes connect to in a `known_hosts` file. The file may be enabled within the `.travis.yml` file using the `-o UserKnownHostsFile=<file_name>` property.

***Observation 5**: Irrelevant properties that are ignored by Travis CI runtime (anti-pattern 3) appear frequently.* Hansel detects 242 instances of anti-pattern 3, which can present imminent concerns or future risks (see Table 5.2).

Making spelling mistakes when defining properties and placing properties in the incorrect location within the `.travis.yml` are example causes of irrelevant properties that raise imminent concerns. We find 74 instances of misspelled properties in our corpus. These misspelled properties are an imminent concern because misspelled properties and all of the commands that are associated with those properties are ignored by the Travis CI runtime. In the best case, ignored properties will lead to build breakage, which is frustrating and may slow development progress down. In the worst case, the CI job will successfully build while producing incorrect deliverables, which may allow failures or unintended behaviour to leak into official releases.

We also find 148 instances of misplaced properties in our corpus. For example, the `webhooks` property should be defined as a sub-property of the `notifications` property; however, it appears as a root-level property in four subject systems. This is an imminent concern because misconfigured properties are also ignored by the Travis CI runtime.

We label the use of experimental or deprecated features in the Travis CI specification as a future risk. There are 15 instances of using experimental properties in the corpus. For example, the undocumented `group` property allows users to specify which set of build images are to be used by the Travis CI runtime. Since this feature is actively being developed, the Travis CI team does not recommend using it yet. Projects that use the `group` property may encounter future problems if the property name or behaviour changes.

Users may also use deprecated properties such as `source_key`. We find five instances of use of deprecated features in the corpus. They present a future risk because Travis CI may stop supporting these properties at any time.

***Observation 6**: The most common variant of anti-pattern 4 is using `install` phase commands in the `script` phase.* Table 5.3 shows that commands that we expect to appear in the `install` phase appear 467 times in other phases. We find that this often occurs because de-

Table 5.2: Examples of irrelevant properties that we observed in `.travis.yml` files.

	Reason	Examples
Imminent Concerns	Misspelled properties	`notications`, `notificactions`, `notification`, `deployg`, `before install`, `before_srcipt`, `before-install`, `before-script`, `branch`, `phps`
	Misplaced properties	`only`, `webhooks`, `on_failure`, `on_success`, `irc`, `email`, `exclude`, `fast_finish`
Future Risks	Experimental Features	`group`
	Deprecated features	`source_key`

Table 5.3: Commands that appear in unrelated phases.

Expected in \ Observed in	**Install**	**Script**	**Deploy**
Install	-	467	0
Script	0	-	0
Deploy	0	52	-

velopers prepend lines to install required packages to the body of the `script` phase. By not using *install* phase for installing dependencies, these projects are unable to leverage Travis CI runtime optimizations (e.g., caching), which speed up builds.

The commands that we expect in the `deploy` phase appear 52 times in the `script` phase. We find that developers tend to run deployment-related commands in the `script` phase immediately after compiling and testing.

The Travis CI team states that compiling and testing tasks should appear in the `script` phase. The `deploy` phase is typically reserved for uploading deliverables to cloud service providers (e.g., Heroku, AWS, Google App Engine) or package repositories (e.g., NPM, PyPI, RubyGems). This separation of concerns allow the Travis CI runtime to optimize resources

within its CI infrastructure. For example, during the `script` phase, the infrastructure can be tuned to perform more CPU- and I/O-heavy operations, while during the `deploy` phase, the infrastructure can allocate additional network bandwidth and less CPU horsepower. If the separation of concerns is not respected, the Travis CI team cannot make such optimizations.

***Observation 7**: Developers often violate semantics by applying static analysis too late in the CI process.* For detecting semantics violations in sub-phases of the CI process, we search for calls to popular code coverage and static analysis tools (listed in the 'static analysis' row of Table 5.1) in the `after_script` phase. We detect 40 of such instances.

One plausible explanation for the occurrence of this anti-pattern is that developers may assume that the `after_script` phase is executed immediately after the `script` phase, similar to how the `after_deploy` phase is executed immediately after the `deploy` phase. Yet, as shown in Figure 2.2, the `after script` phase is executed after deployment-related phases are executed. Indeed, we find 40 cases where static analysis tools are being executed at the end of the CI process, after deployment, when is likely too late to act upon issues that are detected.

Summary: *Developers misuse and misconfigure CI specifications. The anti-patterns that we define can expose a system to security vulnerabilities, cause unintended CI behaviour, or delay SQA activities until after deployment.*
Implications: *Hansel, our anti-pattern detector, can detect misuse and misconfiguration of CI specifications. If Hansel's warnings are addressed, the consequences of CI misuse and misconfiguration can be avoided.*

(RQ2) Can anti-patterns in CI specifications be removed automatically?

Approach. We aim to check whether Hansel-detected anti-patterns can be removed automatically. To do so, we randomly select a subset of candidates for removal and manually classify them until we achieve *saturation* [90], i.e., when new data do not add to the meaning of the categories. In our case, saturation was achieved after analyzing 250 candidates for removal, where no new categories were detected during the analysis of the last 79 candidates.

Before transforming the candidates, we check whether they are valid specifications by using the TravisLint tool.[2] We then apply Gretel to the valid candidates in order to remove

```
language: node_js
node_js:
- '0.10'
before_script:
- cd frontend
- npm install -g bower grunt-cli
- npm install
- bower install
script:
- grunt test
```

```
language: node_js
node_js:
- '0.10'
install:
- cd frontend
- npm install -g bower grunt-cli
- npm install
- bower install
script:
- grunt test
```

Figure 5.1: An example where a state-altering command affects the removal of an anti-pattern. The `cd` command should also be migrated to the `install` phase along with the `npm install` and `bower install` commands.

the anti-pattern. We apply TravisLint again to the transformed files to make sure that they are still valid. Finally, we manually inspect the instances of removed anti-patterns to check whether the transformation has changed the behaviour of the original specification.

Results. We find that 174 of the 250 randomly selected anti-pattern instances (69.60%) can be removed automatically. Moreover, 69 (27.60%) of the remaining cases can be fixed, but require manual verification to ensure that the original behaviour is preserved. We perform this manual verification and provide three observations about these 69 cases.

***Observation 8**: There are 38 instances of anti-patterns where the command under analysis is preceded by a state-altering command.* The state-altering commands include:

- File system operations (i.e., `cp`, `cd`, `mv`, `mkdir`).
- Package managers (i.e., `npm update`, `npm cache clean`, `gem update`, `apt-get update`, `bower cache clean`, `git submodule update`).
- Environment variable and database-related operations.

State-altering commands may also need to migrate along with the anti-pattern commands to the more appropriate section. Figure 5.1 shows an example where a state-altering command impacts the removal of an anti-pattern, taken from *lamkeewei/battleships*,[14] a tool for building

[14] `https://github.com/lamkeewei/battleships`

Python apps for the Google App Engine. In this case, lines 6–8 are implicated in the anti-pattern, but line 5 must be executed before lines 6–8, and thus, must be included in the fix.

Observation 9*: In 12 instances, there are compound commands that are connected by a double ampersand.* In this case, the `bash` shell only invokes the command(s) that follow after the ampersands if the command(s) that precede the ampersands did not fail (i.e., returned an error code of zero). Installation commands that appear before the ampersands can be safely moved to the `install` phase while preserving this behaviour, since if the `install` phase fails, the build job terminates with an error status in Travis CI.

Observation 10*: In 29 instances, limitations in the* ruamel.yaml *framework*[3] *lead to problems in the removal of anti-patterns.* The problems that we encountered are listed below:

- Version numbers may be parsed as floating point numbers, causing trailing zeros to be removed in the output. For example, `0.10` is transformed into `0.1`.
- Property-level comments are missing after removal.
- Duplicate properties are missing after removal.
- Line breaks in multi-line commands are replaced with `'\n'` after removal.

We manually fix these minor issues before proceeding.

Seven (2.80%) of the remaining projects use the Yarn package manager[15] along with npm to manage dependencies. The removals that we propose are incompatible with such projects. We plan to add support for Yarn and other package managers in the future.

Summary: *The detected instances of the most frequent CI anti-pattern can be removed automatically in 69.60% of cases. This improves to 97.20% if a post hoc manual inspection phase is included (semi-automatic removal).*
Implications: *Hansel-detected anti-patterns can be removed (semi-)automatically with Gretel to avoid the consequences of CI misuse and misconfiguration.*

[15] `https://yarnpkg.com/en/`

(RQ3) Are automatic removals of CI anti-patterns accepted by developers?

Approach. To better understand the utility of Gretel, we apply it to the 174 instances that could be removed automatically to fix the anti-patterns and offer these improvements to the studied projects as pull requests.

Results. Of the submitted pull requests, 49 received responses from the projects' developers (response rate: 28.16%).

***Observation 11**: 36 of the 49 pull requests that received responses (73.47%) have been accepted and integrated by the subject systems.* Of the 49 anti-pattern fixes to which developers responded, 36 have already been accepted by the projects at the time of this submission.

13 pull requests were rejected by project maintainers. Two of the 13 were rejected because our pull request appeared to introduce build breaks, which were introduced by other commits that are not related to the removal of anti-patterns. In another two pull requests, the developers did not understand why our change had added new commands. These commands were added to preserve the implicit behaviour of phases that did not exist prior to applying our removal. Two other rejected pull requests came from projects that are no longer being maintained.

Only in one pull request were our changes rejected because the developer did not agree with our premise that this change is beneficial. The developer pointed to Travis CI documentation, which has an example to demonstrate an unrelated feature that uses `install`-related commands in the `before_script` phase.[16] We contacted the Travis CI team regarding this and they agreed that the documentation needs to be fixed by moving the `install` commands out of the `before_script` phase in the example as it is violating the semantics.

The six other rejected pull requests were closed without any explanation from the project maintainers.

Summary: *Automated fixes for CI anti-patterns are often accepted by developers and integrated into their projects (73.47% of pull requests that received a response or 20.68% of all submitted pull requests).*
Implications: *Hansel and Gretel produce patches that are of value to active development teams.*

[16] `https://docs.travis-ci.com/user/languages/javascript-with-nodejs/#Using-Gulp`

5.5 Further Insights into CI/CD Misuse

In this section, we discuss the observed results further in terms of misuse of CI.

5.5.1 Dependence on Default Behaviour

The Travis CI design conforms to the principle of "convention-over-configuration". When no command is specified for a phase (i.e., the phase is not configured), a set of default commands (i.e., the convention) is automatically executed. For example, in Node.js projects, the current default behaviour for the `install` and `script` phases is to invoke `npm install` and `npm test`, respectively.

The "convention-over-configuration" principle might introduce problems if the conventions change. These changes may break the builds of projects that depended upon the old convention. Other tools that conform to the "convention-over-configuration" principle (e.g., Maven, Rails) address the problem of changing conventions by maintaining versions of the schema of the configuration file. This makes the convention that is associated with each version explicit.

Although we do not classify it as an anti-pattern, we detect 5,913 projects in our corpus (63.5%) that depend upon the Travis CI convention. A future change to the convention could affect break the builds of these 5,913 projects.

The convention of Travis CI must evolve to keep up with changes in the build tool ecosystem; however, without a versioning mechanism, configurations that depend upon the prior convention may be susceptible to breakage. For example, for Objective-C builds, the current Travis CI convention is to invoke `xctool`.[17] Facebook, the organization that maintains `xctool`, have deprecated it as of 2016.[18] If the Travis CI team switches the convention to an actively supported tool, the builds of projects that depend upon the existing `xctool` convention will be broken.

Furthermore, evolution of the Travis CI lifecycle itself may introduce build breakage. For example, a recent change to the behaviour of the `after_script` phase[19] ensures that the phase is executed at the end of the CI process. Moreover, its commands are executed regardless of the outcome of the previous phases. Due to this change, projects that depended upon

[17] `https://github.com/facebook/xctool`
[18] `https://github.com/facebook/xctool/blob/master/README.md#features`
[19] `https://blog.travis-ci.com/after_script_behaviour_changes/`

failures in the `after_script` phase preventing the build from proceeding to the `deploy` phase had to move such commands to the `script` phase.

Conversely, when CircleCI, a competing CI service, made substantial changes to the YAML DSL of their configuration files, they introduced a new schema version,[20] while still supporting the old version. The name of the configuration file was also changed from `circle.yml` to `config.yml`, making it difficult for users to mistakenly add deprecated properties in the new configuration file.

5.5.2 Storage of Sensitive Data

We find that projects in our sample have stored sensitive data, such as passwords, private keys, and other security-related properties, in the `.travis.yml` file. For example, the *huginn/huginn* project[21] has `APP_SECRET_TOKEN` defined as a public environment variable in the `.travis.yml` file. Having these properties insecurely recorded in plain text within the `.travis.yml` file can expose the project and potentially, the Travis CI infrastructure to exploits. Sensitive data, such as API credentials, should be encrypted and stored under the `secure` property in the `.travis.yml` file.

We perform an exploratory analysis to estimate the number of instances of sensitive data being stored in plain text in our corpus. We search for the security-related suffixes `key`, `token`, and `secret` in the names of environment variables that appear outside of the `secure` property. To prevent double counting, we remove occurrences where the environment variable setting is the value of another environment variable. If the other environment variable is stored insecurely, we will have already reported it.

Table 5.4 shows that we detect 189 projects with instances of sensitive data in our corpus (2.03%). This is a lower bound, since our suffix matching approach does not detect all environment variables that contain sensitive data.

5.5.3 Dependence on External Scripts

Another potential anti-pattern is placing a large amount of CI logic in external scripts. For example, the *aescobarr/natusfera* project[22] uses the `before_install`, `before_script`, `script`,

[20] `https://circleci.com/docs/2.0/migrating-from-1-2/`
[21] `https://github.com/huginn/huginn`
[22] `https://github.com/aescobarr/natusfera`

Table 5.4: Sensitive data in `.travis.yml` files.

Type	Property Name	# of Projects
Keys	SAUCE_ACCESS_KEY, GITHUB_OAUTH_KEY, BROWSER_STACK_ACCESS_KEY, TOKEN_CIPHER_KEY, TESTSUITE_BROWSERSTACK_KEY, RECAPTCHA_PRIVATE_KEY, RAILS_SECRET_KEY, IMGUR_API_KEY	124
Tokens	ATOM_ACCESS_TOKEN, CODECLIMATE_REPO_TOKEN, COVERALLS_REPO_TOKEN, APP_SECRET_TOKEN, ADMIN_APP_TOKEN, CODACY_PROJECT_TOKEN	56
Secrets	GITHUB_CLIENT_SECRET, JWT_SECRET, APP_SECRET, OPBEAT_SECRET, WEBHOOK_SECRET	9

and `after_script` phases; however, each phase just calls an external script. Since logic in external scripts is hidden from the Travis CI runtime (recall Observation 14), optimizations will be suboptimal.

We again perform an exploratory analysis to study the use of external scripts. We search for commands in our corpus that invoke the `sh` or `bash` interpreters or have the `.sh` extension. Applying this, we detect at least one shell script has been used by 1,924 projects in our corpus (20.6%).

5.5.4 Applicability to Other CI/CD Services

Other popular CI services, such as CircleCI, Wercker, and AppVeyor also use YAML DSLs for specifying CI configuration. Thus, the anti-patterns that we define in this paper may also apply to these services. For example, CircleCI uses a `config.yml` file[23] to configure the CI process. Since commands to be executed during build jobs are specified in this file, anti-pattern 1 (i.e., redirecting scripts into interpreters) may occur in CircleCI specifications.

[23] `https://circleci.com/docs/2.0/`

CircleCI users are also susceptible to the anti-pattern 2 (i.e., bypassing security checks) because users can manually set `StrictHostKeyChecking=no` in the `config.yml` file, exposing the host to man-in-the-middle attacks, when executing commands that require an SSH connection.[24]

CircleCI is robust to anti-pattern 3 (i.e., using irrelevant properties) because build jobs terminate immediately if an unsupported property is processed in the `config.yml` file. This behaviour differs from Travis CI, where unsupported properties do not prevent build jobs from proceeding.

CircleCI users are susceptible to anti-pattern 4 (commands unrelated to the phase). Similar to `.travis.yml` files, `config.yml` files have seven sections that represent phases of the CI process (i.e., `machine`, `checkout`, `dependencies`, `database`, `compile`, `test`, and `deployment`). Each phase has three sub-phases (i.e., `pre`, `override`, and `post`). Similar to Table 5.1, we can map commands to CircleCI phases where they should appear.

5.6 Threats to Validity

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

5.6.1 Internal Validity

The list of anti-patterns that we present in the chapter is not exhaustive. However, to the best of our knowledge, this chapter is the first to define, detect, and remove anti-patterns in CI specifications. Our set of anti-patterns is a starting point for future studies to build upon. Future studies that define anti-patterns using other data sources, e.g., developer surveys [26], may prove fruitful.

Hansel uses a lightweight approach to detect instances of anti-patterns. A more rigorous analysis may uncover additional instances of anti-patterns. Thus, our anti-pattern frequency results should be interpreted as a lower bound.

Projects may use Travis CI without a `.travis.yml` file. In this case, the Travis CI runtime assumes that the project is using Ruby and would apply the conventional Ruby CI process.

[24] `https://discuss.circleci.com/t/add-known-hosts-on-startup-via-config-yml-configuration/12022`

Since we are unable to identify such projects automatically, we only consider projects with a `.travis.yml` file in the root directory of the project.

5.6.2 External Validity

In terms of the generalizability of our results to other systems, we focus only on open source subject systems, which are hosted on GitHub and use Travis CI as the CI service provider. GitHub is one of the most popular hosting platforms for open source software projects and Travis CI is the most widely adopted CI service among open source projects [2]. Therefore, our findings are applicable to a large proportion of open source projects. Moreover, given the similarities among the popular CI services (see Section 5.5.4), our observations are likely applicable to some degree.

5.6.3 Construct Validity

Our proposed CI anti-patterns are subject to our interpretation. To mitigate this threat, we review Travis CI documentation and consult with the Travis CI support team when inconsistencies are encountered. Furthermore, the rate at which our pull requests are being accepted (73.47%) is suggestive of the value of addressing these anti-patterns.

5.7 Chapter Summary

Similar to programming languages, the features in CI configuration files can be used and misused by the developers. Through our study of 9,312 open source systems that use Travis CI, we make the following observations about the misuse of CI specifications:

- Hansel detects anti-patterns in the Travis CI specifications of 894 out of the 9,312 studied projects (9.60%). Moreover, in a sample of 100 projects, Hansel achieves a recall of 82.76% (Observations 1–7).
- The instances of anti-pattern 4 can be removed automatically in 69.60% of the subject systems. This percentage can be increased to 97.20% if a post hoc manual inspection phase is included (Observations 8–10).
- Of the 49 pull requests for instances that are removed automatically and to which developers responded, 36 (73.47%) have been accepted (Observation 11).

Our CI misuse study shows that anti-patterns that threaten the correctness, performance, and security of build jobs are impacting a considerable proportion of Travis CI users (9.60%). Hansel and Gretel can detect and remove these anti-patterns accurately, allowing teams to mitigate or avoid the consequences of misusing CI features.

Chapter 6

Noise and Heterogeneity in CI/CD Build Data

Note. An earlier version of the work in this chapter appears in the *Proceedings of the 33rd International Conference on Automated Software Engineering (ASE 2018)* [53].

6.1 Introduction

After making source code changes, developers execute automated builds to check the impact on the software product. These builds are triggered while features are being developed, when changes have been submitted for peer review, and/or prior to integration into the software project's version control system.

Tools such as Travis CI facilitate the practice of Continuous Integration (CI), where code changes are downloaded regularly onto dedicated servers to be compiled and tested [3]. The popularity of development platforms such as GitHub and CI services such as Travis CI have made the data about automated builds from a plethora of open source projects readily available for analysis.

Characterizing build outcome data will help software practitioners and researchers when building tools and proposing techniques to solve software engineering problems. For example, Rausch et al. [56] identified the most common breakage types in 14 Java applications and Vassallo et al. [8] compared breakages from 349 open source Java projects to those of a financial organization. While these studies make important observations, understanding the nuances and complexities of build outcome data has not received sufficient attention by software engineering researchers. Early work by Zolfagharinia et al. [91] shows that build failures in the

Perl project tend to be time- and platform-sensitive, suggesting that interpretation of build outcome data is not straightforward.

To support the interpretation of build outcome data, in this chapter, we set out to characterize build outcome data according to two harmful assumptions that one may make. To do so, we conduct an empirical study of 3,702,071 build results spanning 1,276 open source projects that use the Travis CI service.

Noise. First, one may assume that build outcomes are *free of noise*. However, we find that in practice, some builds that are marked as successful contain breakages that need attention yet are ignored. For example, developers may label platforms in their Travis CI configurations as `allow_failure` to enable experimentation with support for a new platform. The expectation is that once platform support has stabilized, developers will remove `allow_failure`; however, this is not always the case. For example, the *zdavatz/spreadsheet*[1] project has had the `allow_failure` feature enabled for the entire lifetime of the project (five years). Examples like this suggest that noise is likely present in build outcome data.

There are also builds that are marked as broken that do not receive the immediate attention of the development team. It is unlikely that such broken builds are as distracting for development teams as one may assume. For example, we find that on average, two in every three breakages are *stale*, i.e., occur multiple times in a project's build history. To quantify the amount of noise in build outcome data, we propose an adapted signal-to-noise ratio.

Heterogeneity. Second, one may assume that builds are *homogeneous*. However, builds vary in terms of the number of executed jobs and the number of supported build-time configurations. For example, if the Travis CI configuration includes four Ruby versions and three Java versions to be tested, twelve jobs will be created per build because 4×3 combinations are possible. Zolfagharinia et al. [91] observed that automated builds for Perl package releases take place on a median of 22 environments and seven operating systems. Builds also vary in terms of the type of contributor. Indeed, build outcome and team response may differ depending on the role of the contributor (core, peripheral).

In this chapter, we study build heterogeneity according to matrix breakage purity, breakage reasons, and contributor type. We find that (1) environment-specific breakages are as common as environment-agnostic breakages; (2) the reasons for breakage vary and can be classified into

[1] `https://github.com/zdavatz/spreadsheet`

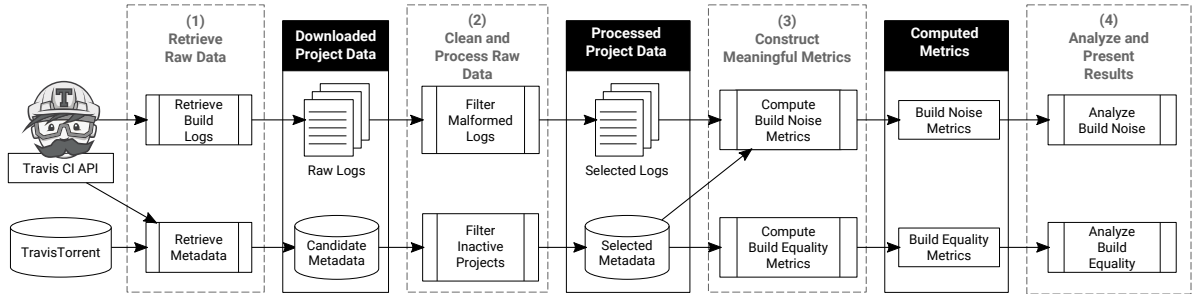


Figure 6.1: An overview of the approach we followed for data analysis.

five categories and 24 subcategories; and (3) broken builds that are caused by core contributors tend to be fixed sooner than those of peripheral contributors.

Take-away messages. Build outcome data is noisy and heterogeneous in practice. If build outcomes are treated as the ground truth, this noise will likely impact subsequent analyses. Therefore, researchers should filter out noise in build outcome data before conducting further analyses. Moreover, tool developers and researchers who develop and propose solutions based on build outcome data need to take the heterogeneity of builds into account.

In summary, this paper makes the following contributions:

- An empirical study of noise and heterogeneity of build breakage in a large sample of Travis CI builds.
- A replication package containing Travis CI specification files, metadata, build logs at the job level, and our data extraction and analysis scripts.[2]
- A taxonomy of breakage types that builds upon prior work.

The remainder of the chapter is organized as follows: Section 6.2 describes the research methodology. Sections 6.3 and 6.4 present our findings related to noise in build outcome and build heterogeneity, respectively. Section 6.5 discusses the broader implications of our study for the research and tool building communities. Section 6.6 outlines the threats to validity. Finally, Section 6.7 concludes the chapter.

6.2 Study Design

In this section, we describe our rationale for selecting the corpus of studied systems and our approach to analyze this large corpus of build data, which follows Mockus' four-step procedure [92] for mining software data. Figure 6.1 provides an overview of our approach.

6.2.1 Corpus of Candidate Systems

We conduct this study by using openly available project metadata and build results of GitHub projects that use the Travis CI service to automate their builds. GitHub is the world's largest hosting service of open source software, with around 20 million users and 57 million repositories, in 2017.[3] A recent analysis shows that Travis CI is the most popular CI service among projects on GitHub.[4]

6.2.2 Retrieve Raw Data

We begin by retrieving the TravisTorrent dataset [52], which contains build outcome data from GitHub projects that use the Travis CI service. As of our retrieval, the TravisTorrent dataset contains data about 3,702,595 build jobs that belong to 680,209 builds spanning 1,283 GitHub projects. Those builds include one to 252 build jobs (median of 3). In addition to build-related data, the TravisTorrent dataset contains details about the GitHub activity that triggered each build. For example, every build includes a commit hash (a reference to the build triggering activity in its *Git* repository), the amount of churn in the revision, the number of modified files, and the programming language of the project. TravisTorrent also includes the number of executed and passed tests.

TravisTorrent alone does not satisfy all of the requirements of our analysis. Since TravisTorrent infers the build job outcome by parsing the raw log, it is unable to detect the outcome of 794,334 jobs (21.45%). Furthermore, TravisTorrent provides a single *broken* category, whereas Travis CI records build breakage in three different categories (see Subsection 6.2.4).

To satisfy our additional data requirements, we complement the TravisTorrent dataset by extracting additional data from the REST API that is provided by Travis CI. From the API, we collect the CI specification (i.e., `.travis.yml` file) used by Travis CI to create each build

[2] `https://github.com/software-rebels/bbchch`
[3] `https://github.com/blog/2345-celebrating-nine-years-of-github-with-an-anniversary-sale`
[4] `https://github.com/blog/2463-github-welcomes-all-ci-tools`

job and the outcome of each build job. To enable further analysis of build breakages, we also download the plain-text logs of each build job in the TravisTorrent dataset.

6.2.3 Clean and Process Raw Data

Since we focus on build breakages, we filter away projects that do not have any broken builds. This excludes from our analysis toy projects that have configured CI initially but do not use CI services. 1,276 projects (out of 1,283) survive this filter.

We observe that 996 build logs do not parse cleanly. When retrieving these logs, the Travis CI API returned a truncated or invalid response. We also filter these logs out of our analysis; however, we do note that these 996 logs account for a negligible proportion of the sample of analyzed build logs (996/3,702,595 = 0.03%).

6.2.4 Construct Meaningful Metrics

In this subsection, we first define the Travis CI concepts that are useful for understanding our work. Then, we define the metrics that we use to operationalize the study dimensions.

Core Concepts in Travis CI. In this chapter, we adhere to the terminology as defined in Section 2.4 and the official Travis CI documentation.[5]

Projects that use the Travis CI service inform Travis CI about how build jobs are to be executed using a `.travis.yml` configuration file. The properties that are set in this configuration file specify which revisions will initiate builds, how the build environments are to be configured for executing builds, and how different teams or team members should be notified about the outcome of the build. Furthermore, the configuration file specifies which tools are required during the build process and the order in which these tools need to be executed.

Metrics. Based on the above concepts, we define seven metrics to analyze build breakage. These metrics are not intended to be complete, but instead provide a starting point for inspecting build breakage for suspicious entries that future work can build upon. Our initial set of metrics belong to two dimensions.

- **Build noise metrics.** In this dimension, we compute the rate at which build breakage is *actively ignored* and *passively ignored*. In addition, we measure the *staleness* of each broken build, i.e., the rate at which breakages are recurring. Finally, we compute the

[5] `https://docs.travis-ci.com/user/for-beginners/`

Signal-To-Noise Ratio (SNR) to measure the proportion of noise in build outcome data caused by passively and actively ignored build breakage.

- **Build heterogeneity metrics.** In this dimension, for each broken build, we compute the *matrix breakage purity* and classify broken builds by the *root cause*. For practical reasons, we extract root causes for build breakage from the 67,267 jobs that use the Maven build tool. This allows us to build upon the Maven Log Analyzer [34], which can classify five types and 24 subtypes of Maven build breakage. Finally, we classify each of the version control revisions that are associated with each build, according to *contributor type* (i.e., core or peripheral contributors).

6.2.5 Analyze and Present Results

Using the metrics that we define in Section 3.4, we (1) plot their values using bar charts, line graphs, scatterplots, and bean plots [93]; and (2) conduct statistical analyses using Spearman's ρ, Wilcoxon signed rank tests, and Cliff's δ.

6.3 Noise in Build Breakage Data

The final build outcome does not always tell the complete story. Indeed, a broken build outcome may not indicate a problem with the system, but rather a problem with the build system or test suite. Conversely, a passing build outcome may only be labeled as such because breakages in particular jobs are being ignored.

In this section, we present the results of our noisiness study in terms of outcomes of builds that are actively ignored (6.3.1), passively ignored (6.3.2), or stale (6.3.3). We also provide an overview of the signal-to-noise ratio in the studied corpus (6.3.4).

6.3.1 Actively Ignored by Developers

Motivation. Support for new runtime environments is often slowly rolled out through adaptive maintenance [94]. While support for new platforms are in the experimental stage, developers may ignore build breakage on these platforms.

To prevent failing jobs in experimental areas of the codebase from causing build breakage, Travis CI users can set the `allow_failure` property when testing against versions or con-

figurations that developers are not ready to officially support.[6] In other words, a job may fail; however, because the developers chose to ignore the outcome of its configuration, the outcome of the build is passing. So if analyses assume the build is successful because the reported outcome is passing, *actively ignored breakages* may introduce noise. Therefore, we analyze how often breakages are actively ignored in our corpus.

Approach. We begin by selecting all of the 496,240 passing builds in our dataset. From those builds, we select the ones with failing jobs. Then, we retrieve the corresponding version of the `.travis.yml` for each of those selected builds and check if the `allow_failure` property is enabled for the failing jobs.

Results. In addition to computing how often passing builds contain failing jobs, Figure 6.2 shows how the percentage of actively ignored failing jobs is distributed in passing builds that had at least one ignored failed job.

***Observation 1**: 12% of passing builds have an actively ignored failure.* Of the 496,240 passing builds in our corpus, 59,904 builds had at least one actively ignored failure. Moreover, Figure 6.2 shows that in the passing builds that had at least one actively ignored failing job, the median percentage of ignored failing jobs is 25%.

In an extreme case, 87% of the jobs were actively ignored. We observe this in the *rubycas/rubycas-client*[7] project where the `allow_failure` property is set in 33 out of the 38 jobs.[8] Upon closer inspection, we observe that this is an example of the intended use of the `allow_failure` property. This build specifies eleven *Ruby* versions as runtimes and four *Gemfiles* for dependency management. Six of the combinations are explicitly excluded. Thus, 38 jobs are created for each build ($11 \times 4 - 6$). All of the 33 jobs that have the `allow_failure` property set fail. In subsequent builds, after several source code changes by the development team, all of these failing jobs begin to pass. Finally, the development team removes the `allow_failure` property from these jobs with an accompanying commit message that states that "builds should fail on released versions of ruby and rails". The development team only ignored failures while they improved their support for multiple ruby and rails versions. Vassallo et al. [35] also have observed that `allow_failure` property is sometimes used for valid reasons (e.g., temporarily in jobs that are not fully implemented yet).

[6] `https://docs.travis-ci.com/user/customizing-the-build/#Rows-that-are-Allowed-to-Fail`
[7] `https://github.com/rubycas/rubycas-client`
[8] `https://travis-ci.org/rubycas/rubycas-client/builds/5604025`

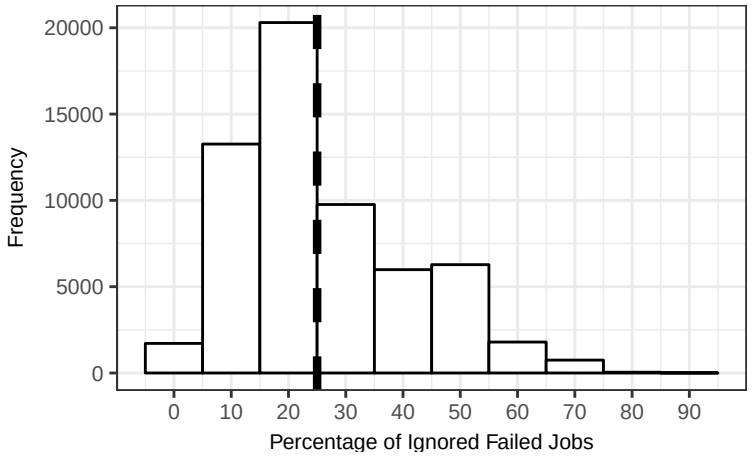


Figure 6.2: Percentage of ignored failed jobs in passing builds that had at least one ignored failed job across all projects.

On the other hand, the `allow_failure` setting can be misused. For example, in the *zdavatz/spreadsheet*[9] project, the `allow_failure` property, which is set in the initial build specification of the project, is never removed from the build specification throughout the five-year history of the project.[10] Furthermore, in our corpus, we detect 23 projects that had the `allow_failure` property set in all of their builds. These projects were not short-lived, with 31 to 769 builds in each project (median of 151). This suggests that although the intended purpose of the `allow_failure` property is to temporarily hide breakages, development teams do not always disable this property after it has been set, leaving the breakages hidden.

Passing build outcomes do not always indicate that the build was entirely clean.

6.3.2 Passively Ignored by Developers

Motivation. Build breakage is considered to be distracting because it draws developer attention away from their work to fix build-reported issues [6, 41, 95]. If development can proceed without addressing a build breakage, we suspect that the breakage is not distracting. Since these *passively ignored breakages* may introduce noise in analyses that assume that all breakages are distracting. We set out to analyze how often breakages are passively ignored.

Approach. To detect passively ignored breakages, we construct and analyze the directed graph of revisions from the version history that have been built using Travis CI.

[9] `https://github.com/zdavatz/spreadsheet`
[10] `https://github.com/zdavatz/spreadsheet/blame/master/.travis.yml`

[1] **Build Filtering.** We start by selecting the `git_trigger_commit` and the `git_prev_built_commit` fields of each build from TravisTorrent. The `git_trigger_commit` field refers to the revision within the repository that is being built. The `git_prev_built_commit` field refers to the revision that was the target of the immediately preceding build. Multiple builds may be associated with one `git_trigger_commit` because developers can configure Travis CI to run builds at scheduled time intervals, even if no new commits have appeared in the repository.[11] Builds can also be triggered by the Travis CI API, regardless of whether there are new commits in the repository.[12] We remove such duplicate builds by checking for builds that have *event_type* property set to `cron` or `api`. This reduces the number of builds from 680,209 to 676,408. Travis CI also triggers builds when *Git* tags are created even if the tagged commit has already been built. We remove builds that were triggered by tag pushes by checking for non-null values for the *tags* property. This reduces the number of builds to 659,048. However, there are multiple builds remaining for one `git_trigger_commit` because manual build invocations can be made via the Travis CI web interface. These manual invocations cannot be distinguished from regular builds that were triggered by Git pushes. Therefore, when multiple builds are encountered for one `git_trigger_commit`, the earliest build is selected. This reduces the number of builds to 610,550.

[2] **Graph Construction.** Nodes in the graph represent build-triggering commits, while edges connect builds chronologically. All nodes are connected by edges from `git_prev_built_commit` node to `git_trigger_commit` node.

[3] **Graph Analysis.** We use the directed graph to identify build-triggering commits from which others branch. We select those branch point build-triggering commits that have a non-passing outcome. Then, we traverse all of the branches of such builds in a breadth-first manner to find the earliest build where the outcome is passing. Finally, we count the number of builds along the shortest path between the breakage branch point and the earliest fix.

[11] `https://docs.travis-ci.com/user/cron-jobs/`
[12] `https://docs.travis-ci.com/user/triggering-builds/`

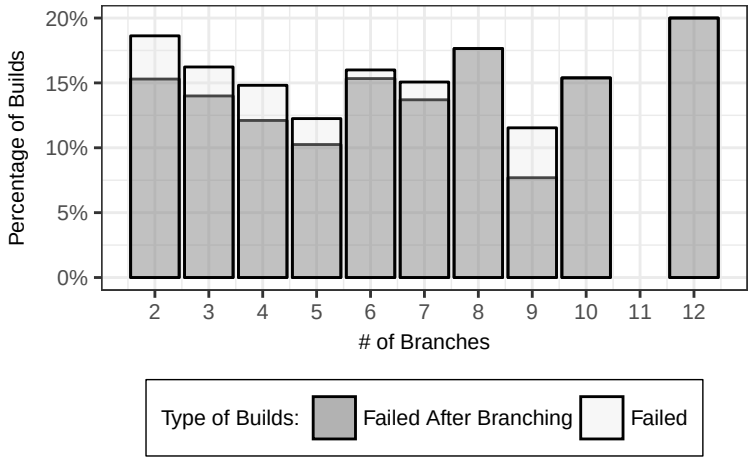


Figure 6.3: Developers branch out into multiple development paths (branches) even after build breakages. Percentage of broken builds at branch points are shown in white. Percentage of broken builds that continued to be broken after branching are shown in grey. There are no broken builds with 11 branches.

Results. Figure 6.3 shows the broken builds that are at the branch points in the version history of the project. To some degree, developers passively ignored these failures by not immediately fixing them and continuing development in multiple paths.

***Observation 2**: Breakages often persist after branching.* Of the 23,068 builds that are triggered by commits at branch points 4,136 (18%) are broken. Of those commits that are branched when the build is broken, 3,426 builds (83%) are not fixed in the immediately subsequent build. These breakages are suspicious because developers have not immediately fixed these breakages and have continued development.

Figure 6.3 shows that commits are branched from up to twelve times when the build was broken. In the 13,102 builds that were not immediately fixed, several commits appear before the fix does. Figure 6.4 shows the maximum and median durations where the projects remained broken in the studied projects.

***Observation 3**: Breakages persist for up to 423 days, and seven days on average, before being fixed.* In one extreme case, the *orbeon/orbeon-forms* project[13] had 485 consecutive build breakages over 423 days before finally the breakage was addressed. Upon further investigation of this breakage, we find that the build is broken due to multiple test failures over time. By analyzing the commit messages of the broken builds in this sequence, we find only 10% of these commits mention fixing the broken build (# of Occurrence of each term: build=3, regression=2,

[13]`https://github.com/orbeon/orbeon-forms`

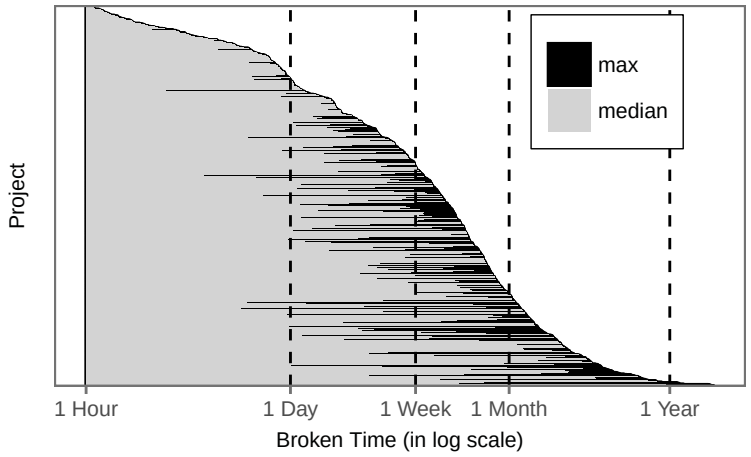


Figure 6.4: In some cases, builds can remain broken for 423 days. The graph shows the maximum and median durations that each project's build remained broken, ordered by the maximum duration.

test=46). However, near the end of the long build breakage sequence, two commits before the build started passing again, the developer has started skipping tests mentioning "For now, don't run integration and database tests".[14] This shows that the build breakages were not the focus of the development activity until the end of the sequence when they turned off the tests that were causing the breakage.

We find that 761 projects have breakages that persist for more than one day, 547 projects have breakages that persist for more than one week, and 227 projects have breakages that persist for more than one month before getting fixed. In eight projects, consecutive build breakages persist for more than one year before getting fixed. The overall median length of the failure sequences is five, while project-specific medians range between 2–29.

In this study, we do not focus on the build breakage differences across branches. In a subsequent study, Vassallo et al. [30] find that broken release branches are present in all studied projects and a median of 11.51% of the `master` branch builds are broken. Moreover, they find that the `master` branch remained broken on average for over six weeks in one project, confirming our observations about long build breakage sequences.

In 83% of branches from broken builds, the breakage persists. These breakages persist for up to 485 commits.

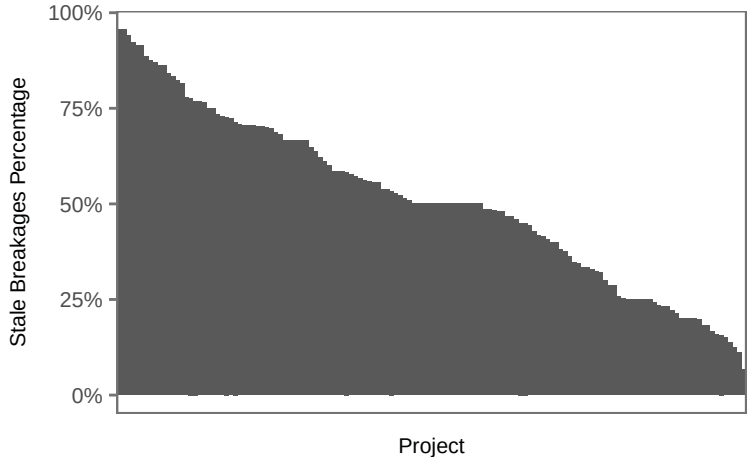


Figure 6.5: Percentage of stale breakages in each project can range from 7% to 96%.

6.3.3 Staleness of Breakage

Motivation. Developers can passively ignore breakages for different reasons. We identify the *staleness* of a build breakage (whether the project has encountered a given breakage in the past) as one of the reasons for ignoring a build breakage. A new breakage is different from a stale breakage because developers may have become desensitized to stale breakages.

Approach. In this section, we investigate how many times developers come across the same breakage repeatedly in the history of a project with respect to the length of build breakage sequences. These stale breakages can occur either consecutively or intermittently. Hence, we extend the *Maven Log Analyzer* developed by Macho et al. [34]. We use it to compare two Travis CI build jobs and check the similarity of the breakages. To make the comparison efficient, this is done in two steps. First, the logs of build jobs are parsed and checked if they are breaking due to the same reason (e.g., compilation failure, test execution failure, dependency resolution failure). If the reason for failures are equal then the details of the failure are also checked (e.g., if both breakages are due to compilation failure, check if the compilation error is the same).

Results. Figure 6.5 shows the percentage of stale build breakages in each project in descending order.

***Observation 4**: 67% of the breakages (6,889 out of 10,816) that we analyze are stale breakages.* On the project level, staleness of breakages ranges from 7% to 96% with a median of 50%. In

[14] `https://github.com/orbeon/orbeon-forms/compare/f137cfb555f1...eb1a8095a025`

the *eirslett/frontend-maven-plugin*[15] project, where we observe the maximum percentage of stale breakages (96%), it was due to the same dependency resolution failure recurring in 23 builds.

> *Two of every three build breakages (67%) that we analyze are stale.*

6.3.4 Signal-To-Noise Ratio

Motivation. In previous analyses, we find that build breakages that are ignored by developers and build successes that include ignored breakages can introduce noise in build outcome data. However, the overall rate of noise in build outcome data is not yet clear. Such an overview is useful for researchers who use build outcome data in their work, to better understand the degree to which noise may be impacting their analyses.

Approach. To quantify the proportion of noise in build outcome data caused by passively and actively ignored build breakage, we adopt the *Signal-To-Noise* ratio (SNR) as follows:

$$SNR = \frac{\#TrueBuildBreakages + \#TrueBuildSuccesses}{\#FalseBuildBreakages + \#FalseBuildSuccesses} \tag{6.1}$$

where *#TrueBuildBreakages* (i.e., signal) is the number of broken builds that are not ignored by developers, *#TrueBuildSuccesses* (i.e., signal) is the number of passing builds without ignored breakages, *#FalseBuildBreakages* (i.e., noise) is the number of broken builds that are ignored by developers, and *#FalseBuildSuccesses* (i.e., noise) is the number of passing builds with ignored breakages.

To compute *#FalseBuildBreakages*, a threshold t_c must be selected such that if the number of consecutive broken builds is above t_c, all builds in such sequences are considered false build breakages. Instead of picking any particular t_c value, we plot an SNR curve as the threshold (t_c) is changed.

Results. Figure 6.6 shows the SNR curve for the subject systems.

***Observation 5**: As t_c decreases from 485 to 1, the SNR decreases from 10.62 to 6.39.* Since *#FalseBuildSuccesses* is not impacted by t_c, the maximum SNR is observed when *#FalseBuild-Breakages* is zero (i.e., when t_c is set to the maximum value). The minimum of SNR is observed

[15] `https://github.com/eirslett/frontend-maven-plugin`

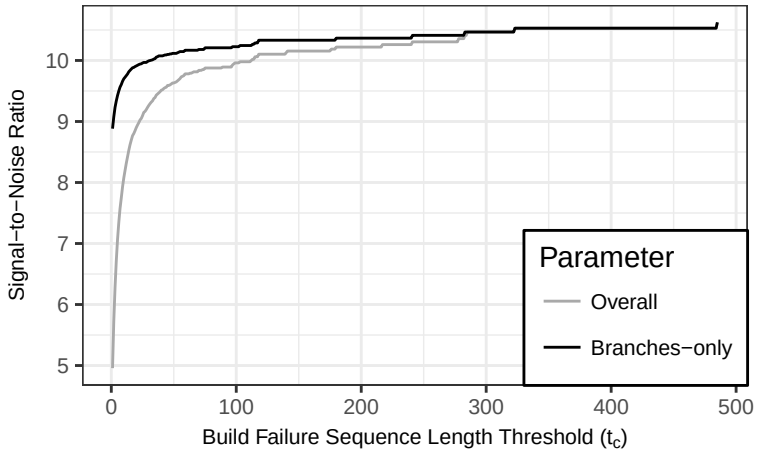


Figure 6.6: For every 11 builds there is at least one build with an incorrect status. The Signal-To-Noise ratio increases when a higher build breakage sequence length is chosen.

when t_c is one and therefore all broken builds that are not immediately fixed are considered false build breakages. If false breakages are defined to be only in consecutive breakages with branches in them, the Signal-to-Noise ratio ranges from 10.19 to 10.62.

> *One in every 7 to 11 builds (9%–14%) is incorrectly labelled. This noise may influence analyses based on build outcome data.*

6.4 Heterogeneity in Build Breakage Data

The way in which builds are configured and triggered vary from project to project. This heterogeneity should be taken into consideration when designing studies of build breakage. Below, we demonstrate build heterogeneity using three criteria.

6.4.1 Matrix Breakage Purity

Motivation. If a software project needs to be tested in multiple environments with different runtime versions, CI services like Travis CI provide the ability to declare these options in a matrix of *runtime, environment,* and *exclusions/inclusions* sections. A build will execute jobs for each combination of included runtimes and environments.

If a build is broken only within a subset of its jobs, the breakage may be platform- or runtime-specific. These environment-specific build breakages may need to be handled differ-

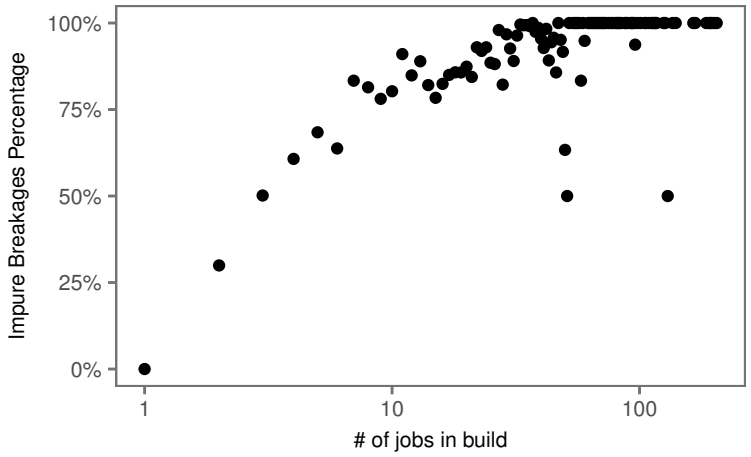


Figure 6.7: Percentage of impure build breakages increases with the number of jobs in each build.

ently from the environment-agnostic breakages. Thus, we want to know the extent to which real breakages are environment specific.

Approach. To study the environments that are affected by a build breakage, we define *Matrix Breakage Purity* as follows:

$$Matrix\ Breakage\ Purity = \frac{\#FailedJobsInBuild}{\#AllJobsInBuild} \tag{6.2}$$

A *Matrix Breakage Purity* below one indicates that the jobs that were run in some environments passed. We compute the *Matrix Breakage Purity* for all builds in our dataset and count the number of builds with values below one. We label all builds that have a *Matrix Breakage Purity* below one as *impure build breakages*.

Results. Figure 6.7 shows how the percentage of impure build breakages varies with respect to the number of jobs per build.

***Observation 6**: At least 44% of broken builds contain passing jobs.* Indeed, environment-specific breakages are almost as common as environment-agnostic breakages.

Given the difference in semantics between pure and impure build breakage, researchers should take this into account when selecting build outcome data for research. For example, in build outcome prediction, if prediction models are trained using data that treats environment-specific and environment-agnostic breakages identically, the model fitness will likely suffer.

Moreover, the insights that are derived from the models will likely be misleading, since the two conflated phenomena will be modelled as one phenomenon.

***Observation 7**: Builds with a greater number of jobs are more likely to suffer from impure build breakages.* Figure 6.7 shows that the number of jobs in a build and the percentage of broken builds that have passing jobs are highly correlated. A Spearman correlation test yields a ρ of 0.8, with $p < 2.2 \times 10^{-16}$. While pure build breakage is common in builds with few jobs, when the number of jobs per build exceeds three, impure build breakage are more frequent than pure ones (i.e., impure breakage percentage > 50%).

Environment-specific breakage is commonplace. Once the number of jobs exceeds three, impure breakages occur more frequently than pure breakages.

6.4.2 Reason for Breakage

Motivation. Builds can break for reasons that range from style violations to test failures. Different types of failures have different implications. For example, while a style violation might be corrected easily, fixing a test failure might require time and effort to understand and address. Since subsequent analyses of build data should handle different types of breakages in different ways, we want to know how types of build breakage vary in reality.

Approach. To analyze the reasons for build breakage in our corpus, we extend the *Maven Log Analyzer* (MLA) [34]. Our extension first parses the Travis CI log file and extracts the sections of the log that correspond to executions of the *Maven* build tool. Then, each of these *Maven* executions are fed to MLA to automatically classify the status of each execution. In addition to the breakage types that were identified in the original work [34], our extended version of MLA also detects the build breakage types that were reported by Vassallo et al. [8] and Rausch et al. [56], as well as ten previously unreported breakage categories.

If MLA classifies all *Maven* executions within a broken build as successful, the build is labelled as a *Non-Maven* breakage. Non-Maven breakages are further classified as *Pre-Maven* if a failing command is detected in the Travis CI log before the *Maven* commands and *Post-Maven* otherwise.

In total, using our extended MLA, we classify 67,267 broken build jobs of projects that use *Maven* as the build tool.

Results. Table 6.1 classifies the broken *Maven* builds by reason.[16]

Observation 8*: Although a large proportion of build breakages are due to the execution of* Ant *from within* Maven*, most of these breakages belong to one project.* Table 6.1 shows that there are 15,850 instances of breakage where the external goal of executing an *Ant* build from within a *Maven* build failed. This accounts for 92.59% of the goal failed breakages in our corpus. However, this is an example of an anomaly that dissipates when examined more closely. Indeed, we find that all of these breakages occur in only two of the studied projects, the overwhelming majority (15,857) of which occur in the *jruby/jruby*[17] project. According to developer discussions, *Ant* is used inside the *Maven* build of *jruby/jruby* for executing tests.[18] However, this complex build setup, which requires 250MB of dependencies, causes build to fail intermittently. The developers hope that the breakages will not occur once the build is completely migrated to *Maven*.

Observation 9*: In our corpus, most breakage is due to commands other than main build tool,* Maven. We observe 41% (27,289) jobs are broken due to reasons other than *Maven* executions failing. This can be either due to a command that was executed by Travis CI (outside of *Maven*) returning an error, the user canceling the build, or Travis CI runtime aborting the build because it exceeded the allocated time.

Observation 10*: Only a small amount of breakage can be automatically fixed by focusing on tool-specific breakage.* For example, 2,257 build jobs are broken because dependency resolution has failed. This suggests that recent approaches that automatically fix dependency-related build breakage [34] will only scratch the surface of the build breakage problem. Moreover, Compilation and Test Execution failures only account for another 29.39% of the breakage in our corpus. Future automatic breakage recovery efforts should look beyond tool-specific breakages to the CI scripts themselves to yield the most benefit for development teams.

> *41% of the broken builds in our corpus failed due to problems outside of the execution of the main build tool. Since tool-specific breakage is rare, future automatic breakage recovery techniques should tackle issues in the CI scripts themselves. Furthermore, to properly analyze a build, a tool must be able to process outputs from multiple technologies.*

[16] More details about reasons for breakage are available online: `https://github.com/software-rebels/bbchch/wiki/Build-Breakages-in-Maven`

[17] `https://github.com/jruby/jruby`

[18] `https://gitter.im/jruby/jruby/archives/2016/05/27`

Table 6.1: Distribution of Build Breakages in *Maven* Projects based on the Categories proposed by Vassallo et al. [8]. and Rausch et al. [56]. Global percentage of each category is shown in brackets.

Category	Subcategory	#	%	Projects
Dependency Resolution*		**2,257**	**(3.41%)**	**18**
Test Execution Failed	Unit	10,759	62.87%	165
	Integration	6,354	37.13%	18
	Total	**17,113**	**(25.89%)**	**171**
Compilation Failed	Production	2,015	87.08%	18
	Test	248	10.72%	12
	Total	**2,314**	**(3.50%)**	**47**
Goal Failed	Pre-processing	44	0.26%	4
	Static-Analysis	210	1.23%	10
	Dynamic-Analysis	8	0.05%	3
	Validation	33	0.19%	5
	Packaging	25	0.15%	4
	Documentation	25	0.15%	7
	Release Preparation	1	0.01%	1
	Deployment - Remote	120	0.70%	9
	Deployment - Local	7	0.04%	1
	Support	3	0.02%	1
	Ant inside Maven*	15,850	92.59%	2
	Run system/Java program*	70	0.41%	2
	Run Jetty server*	8	0.05%	1
	Manage Ruby Gems*	65	0.38%	1
	Polyglot for Maven*	32	0.19%	1
	Total	**17,119**	**(25.90%)**	**47**
Broken Outside Maven	No Log available*	1,554	5.69%	28
	Failed Before Maven*	808	2.96%	3
	Failed After Maven*	7,151	26.20%	46
	Travis Aborted*	16,141	59.15%	172
	Travis Cancelled*	1,635	5.99%	20
	Total	**27,289**	**(41.29%)**	**175**

* New build breakage categories that did not appear in prior work.

6.4.3 Type of contributor

Motivation. Both core and peripheral contributors trigger builds. Since core contributors likely have a deeper understanding of the project than peripheral contributors, builds that are triggered by core contributors might have breakage rates and team responses that differ from those of peripheral contributors. We set out to investigate the differences of build outcome, in these two categories of contributors.

Approach. For analyzing this dimension, we use the two main outcomes of each build (passed or failed) and whether the builds were triggered by a commit that was authored by a core team member. We use the core member indicator from the TravisTorrent dataset,[19] which is set for contributors who have committed at least once within the three months prior to this commit (`gh_by_core_team_member`). Then, we use the broken time and the length of broken build sequences to investigate the relationship between the contributor type and build breakage.

Results. Figure 6.8 shows how the percentage of build outcomes are distributed across projects classified by contributor type.

***Observation 11**: Builds triggered by core team members are break significantly more often than those of peripheral contributors.* A Wilcoxon signed rank test indicates that breakage rates in core contributors are higher than those of peripheral contributors ($p = 1.28 \times 10^{-8}$); however, the effect size is negligible (Cliff's $\delta = 0.13$). Due to having more experience, core team members in the development teams are assigned to complex tasks, which may explain why breakage rates tend to be a little higher. The Wilcoxon test is inconclusive when comparing rates of passing builds among core and peripheral contributors.

Figure 6.9 shows how long build breakages persist classified by contributor type. Figures 6.9a and 6.9b show the length of build breakage sequences in terms of commits and time, respectively.

***Observation 12**: Breakages that are caused by core contributors tend to be fixed sooner than those of peripheral contributors.* A Wilcoxon signed rank test indicates that breakages caused by core contributors tend to persist for significantly less time than those of peripheral contributors ($p = 1.86 \times 10^{-7}$); however, the effect size is negligible (Cliff's $\delta = 0.09$). Another Wilcoxon signed rank test indicates that breakages of core contributors persist for fewer con-

[19] `https://travistorrent.testroots.org/page_dataformat/`

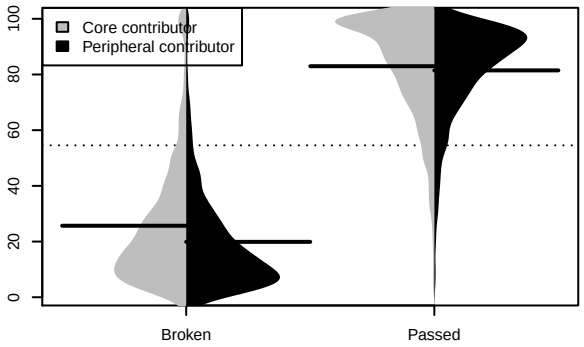
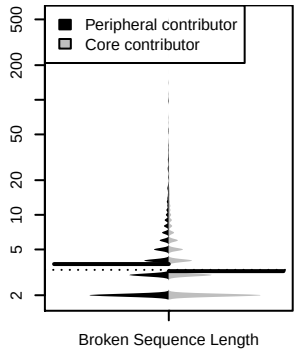
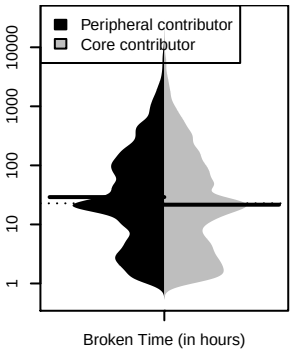


Figure 6.8: Percentage of broken and passing builds classified by contributor type. Horizontal black lines show the median values.



(a) Chains of consecutive breakages caused by peripheral contributors tend to be longer.



(b) Build breakages caused by peripheral contributors take more time to repair.

Figure 6.9: Build breakages caused by peripheral contributors remain broken significantly longer than those of core contributors. Horizontal black lines show the median values.

secutive builds than those of peripheral contributors ($p = 1.81 \times 10^{-8}$); however, the effect size is also negligible (Cliff's $\delta = 0.09$).

The longer time taken by peripheral contributors might be due to multiple attempts of trial and error before fixing a breakage, while core members might be able to identify the root cause of the breakage sooner. Therefore, it may be worthwhile for the researchers working on automatic build breakage repair to focus on build breakages that are caused by peripheral contributors.

Broken builds that are caused by core contributors tend to be fixed sooner than those of peripheral contributors.

6.5 Implications

We now present the broader implications of our observations for researchers and tool builders.

6.5.1 Research Community

Build outcome noise should be filtered out before subsequent analyses. Passing builds might contain breakages that are ignored. Long sequences of repeated breakages might be ignored by the developers as false breakages. If the noise due to false successes and false breakages is not filtered out, the results from prediction models may lead to spurious or incorrect conclusions.

Heterogeneity of builds should be considered when training build outcome prediction models. Some breakages are limited to specific environments while others are not. The reason for breakages vary from trivial issues like style violations to complex test failures. Breakages are often not caused by development mistakes, but by resource limitations in the CI environment. Indeed, build outcome includes many complex categories that can not be accurately represented in the prediction models using only a "broken" or "clean" label.

6.5.2 Tool Builders

Automatic breakage recovery should look beyond tool-specific insight. While recently proposed tools can automatically recover from tool-specific build breakage [34], we find that this category only accounts for a small proportion of CI build breakage in our corpus. Future efforts in breakage recovery should consider CI-specific scripts, for example, detecting those scripts that are at risk of exceeding the allocated time prior to execution.

Richer information should be included in build outcome reports and dashboards. Currently, build tools and CI services provide users with dashboards that show passing builds in green and broken builds in red. However, we found hidden breakages among passing builds and non-distracting breakages among broken builds. Moreover, heterogeneity of breakages introduce further complexities. Build outcome reporting tools and dashboards should consider providing more rich information about hidden, non-distracting, and stale breakages, as well as breakage purity and type.

6.6 Threats to Validity

Construct Validity. Threats to construct validity refer to the relationship between theory and observation. It is possible that there are build failure categories that our scripts are unable to detect. By implementing the categories that were reported in prior work [8, 34, 56] and then manually checking a subset of logs along with their detected failure categories, we ensure most of the *maven* plug-ins and their failure categories are covered by our scripts.

There are likely to be other factors that introduce noise and variability in build outcomes. As an initial study, we focus on the aspects that we think demonstrate noise and heterogeneity of builds in this work. Our list of aspects is not intended to be exhaustive.

Internal Validity. Threats to internal validity are related to factors, internal to our study, that can influence our conclusions. In the analysis of passively ignored breakages, we associate continuous breakage of a build with developers ignoring the breakage. However, developers may be unsuccessfully attempting to fix these breakages during the breakage chain. We do not suspect that this is the most frequent explanation because we find several cases where the initial breakage has several branches (implying that several developers inherited the breakage). Although it can be assumed that these branches are created to fix the build, it is unlikely that twelve branches are created only for bug fixing. We further investigate the staleness of breakages, observing that the same build breakages are often repeated in these long chains.

External Validity. Threats to external validity are concerned with the generalizability of our findings. We only consider open source projects that use the Travis CI service and are hosted on GitHub. However, because GitHub is one of the most popular hosting platforms for open source software projects and Travis CI is the most widely adopted CI service among open source projects, our findings are applicable to a large number of open source projects. Similar to that, we only consider projects that use *Maven* for analyzing reasons for build breakage. However, *Maven* is one of the most popular build tools for Java projects [96] and therefore our findings are widely applicable. Nonetheless, replication studies using data from other hosting platforms, other CI services, and other build tools may provide additional insight.

6.7 Chapter Summary

Automated builds are commonly used in software development to verify functionality and detect defects early in software projects. An off-the-shelf usage of build outcome data is implic-

itly susceptible to harmful assumptions about build breakage. By empirically studying build jobs of 1,276 open source projects, we investigate whether two assumptions hold. First, that build results are not noisy; however, we find in every eleven builds, there is at least one build with a misleading or incorrect outcome on average. Second, that builds are homogeneous; however, we find breakages vary with respect to the number of impacted jobs and the causes of breakage.

Researchers who make use of build outcome data should make sure that noise is filtered out and heterogeneity is accounted for before subsequent analyses are conducted. Build reporting tools and dashboards should also consider providing a richer interface to better represent these characteristics of build outcome data.

In future work, we plan to study how much of an impact noise and heterogeneity can have on common analyses of historical build data. We also plan to investigate whether breakage type varies with respect to contributor type and other commit factors.

Part III

Efficiency in CI/CD Services

Chapter 7

CI/CD Service Providers' Perspective

7.1 Introduction

The main goal of CI is providing fast feedback to developers, allowing them to verify whether their changes cleanly integrate with changes that other team members have submitted. Indeed, the benefits of CI, such as perceived increases in developer productivity and improved software quality, have been observed by the software development community [9]. Both open source [5] and proprietary [6, 67] software organizations have dedicated resources for maintaining CI pipelines for this purpose. These benefits not only improve the organizations that make use of CI, but they can also impact software users through the delivery of higher quality deliverables.

Dedicated cloud-based CI providers such as CircleCI,[1] Travis CI,[2] CloudBees[3] have offered CI services for software organizations to get the benefits of CI without the hassle of provisioning, operating, and maintaining CI infrastructure on their own.

The broad adoption of CI services has presented new opportunities for research on CI/CD. Researchers have interpreted the outcome [53] and duration [54] of the builds of these CI providers from the perspective of the CI users, discussing challenges and benefits of adopting CI [59]. However, the perspective of the CI provider has remained largely unexplored. Focusing on build data from the perspective of CI service providers is also important for capacity planning and identifying opportunities to improve existing provider solutions. Identifying opportunities for managing resources efficiently will help CI service providers to keep oper-

[1] https://circleci.com/
[2] https://www.travis-ci.com/
[3] https://www.cloudbees.com/

ational costs low while delivering fast and reliable CI services. Improving the usability of the service by making use of research findings can help to attract new users and to retain existing ones.

To study CI from the perspective of service providers, we conduct a case study of CircleCI– one of the most popular CI service providers for projects hosted on GitHub.[4] We focus on the CircleCI service in this study due to its availability of reliable build data for a long period. Our dataset includes 23.3 million builds spanning 7,795 open source projects that use the CircleCI service 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: Analyzing usage metrics over an extended period of time may help CI service providers to understand how the feature additions and improvements to the service have impacted the growth of the user base over time and thus allow them to plan accordingly. Studying the growth of CircleCI along multiple productivity metrics 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 the number of monthly users and the total number of builds that are invoked every month. However, this growth has stagnated since the mid-2020. At least 14% of the projects that were inactive on CircleCI during the year 2020 have started using another CI service. Although median values for build duration and throughput have remained stable (under 200 seconds per build and 30 builds per month) during this period, the same metrics have grown rapidly for the most active users of the service (up to 25 minutes per build and 900 builds per month). We also observe a high success rate (67%–92%) and a low Mean Time To Recovery (MTTR of an hour) among the studied builds.

Recommendations: Additional resource usage due to the growth could be reduced by using techniques that have been proposed in research for the acceleration and skipping of CI builds. The high success rate shows that there is a large pool of safe candidate builds to be skipped if they can be identified in advance. Automatic repair techniques can be used to reduce MTTR further.

RQ2: How is time spent during signal-generating builds?

Motivation: By understanding the time consumption of different steps in the CI pipeline,

[4] https://github.blog/2017-11-07-github-welcomes-all-ci-tools/

service providers can identify resource bottlenecks and estimate the operational costs. Identifying the stages that are slowing down the CI pipeline can allow researchers and tool 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 users.

RQ3: Why are some builds unable to provide a signal?
Motivation: Studying why certain builds fail early without providing a signal can help the CI service providers to mitigate such instances, eventually reducing the number of builds that are unable to provide a signal. Researchers can also find opportunities for early detection and remediation of non-signal-providing builds.
Results: Most of the builds are unable to provide a signal because they are prematurely cancelled (3.64% of all builds/50% of non-signal-generating builds). The next most common reasons for builds failing to provide a signal are configuration errors and infrastructure provisioning issues in the CircleCI environment.
Recommendations: Future research should focus on simplifying the configuration and increasing the resiliency of CI services.

Based on our observations, we propose that tool builders should allocate more resources to improve the efficiency and robustness of CI services by adopting existing software engineering research. Furthermore, researchers should also aim to optimize the resource usage of CI services to handle their growth.

The remainder of the chapter is organized as follows. Section 7.2 describes the terminology and metrics related to the CI process. Section 7.3 provides an overview of the design of our study, while Section 7.4 presents the results. Section 7.5 discusses the broader implications of our observations. Section 7.6 presents the threats to the validity of the study. Section 7.7 draws conclusions and proposes directions for future work.

7.2 Core Concepts in Modern CI

Software organizations increasingly use cloud-based CI service providers because these cloud CI services allow them to easily configure CI pipelines based on their needs, and provision

their resources based on demand without the hassle of maintaining their own server infrastructure.

The **organizations** that are using CI services may have multiple **projects** that are kept in source code repositories, each with their own CI **workflows**. A workflow is comprised of one or more **jobs**. CI services typically support specifying workflows in a configuration file (e.g., `.circleci/config.yml` for CircleCI, `.travis.yml` for Travis CI) using a YAML-based DSL. Although this configuration file is primarily used for specifying the sequence of commands to be executed during builds (i.e., invocations of the workflow), it supports further customization such as:

Parallelism: Number of parallel instances of a job to run.

Environment: A map of environment variable names and values to be set during build execution.

Resource Class: Amount of CPU and RAM allocated to each build job.

Organizations that use CI services are billed based on **resource usage** (i.e., a combined metric of build minutes, compute resource type, and concurrently available resources) and **user seats** (i.e., number of team members with the permission to trigger CI builds).

CI workflows can be configured to invoke builds based on development events (e.g., when a pull request is created/updated or a push to a central repository is performed), on a schedule (e.g., every evening after development wraps up), or manually (e.g., on-demand to retry a build without changing the code or programmatically via an API request). Once a build request is received by the CI service via any of these methods, depending on the subscribed plan of the organization 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 CI build.

7.2.1 CI Build Outcomes

CI builds are executed with the expectation that they will produce a signal, which indicates whether the proposed changes to the codebase are ready to be integrated (CI) and/or delivered (CD). However, in practice, build outcomes are not always conclusive. Therefore, we categorize build outcomes as either:

Signal-generating builds. The CI builds that executed until a meaningful outcome, i.e., pass or fail, is produced. If a build passes, CI users know that the proposed changes to the

codebase have at least passed baseline checks (i.e., no faults were uncovered by the CI workflow). If the build fails, CI users can diagnose the problems with their proposed changes while design decisions are still fresh in their minds.

Non-signal-generating builds. CI builds that are terminated before completion. A build could be abruptly stopped before completion due to a user aborting the build, a configuration error, or an infrastructure provisioning issue. Non-signal-generating builds are unable to provide a meaningful signal about the changes to the CI user.

In an ideal CI pipeline, all builds are signal generating. If non-signal-generating builds occur often, CI users may have to spend their considerable amounts of time diagnosing problems in configuration and resource allocation. Furthermore, it is desirable for signal-generating builds to finish executing as quickly as possible to avoid impeding the development progress of CI users.

7.2.2 CI Indicators

CI providers have proposed indicators that track performance in CI pipelines. In a recent report, CircleCI[5] proposes four such indicators:

Build Duration. The time taken for a CI build to execute. A short build duration may indicate that too few tests are executed during the CI process. A long build duration may affect developer productivity because developers will switch contexts to other work while waiting – a costly action to take for knowledge workers like software engineers [97]. CircleCI recommends a build duration of five and ten minutes which typically strikes the right balance.

Mean Time to Recovery (MTTR). The average time between the end times of a failing build and the subsequent successful build. A long MTTR means that the status of the build for a project remains broken for considerable amounts of time, whereas a short MTTR means that developers can quickly resolve build failures and return to their regular tasks without too much of a disruption. CircleCI recommends to keep the MTTR under an hour.

Success Rate. The proportion of signal-generating builds with passing outcomes during a given period of time. The importance of the success rate depends on the team size and

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

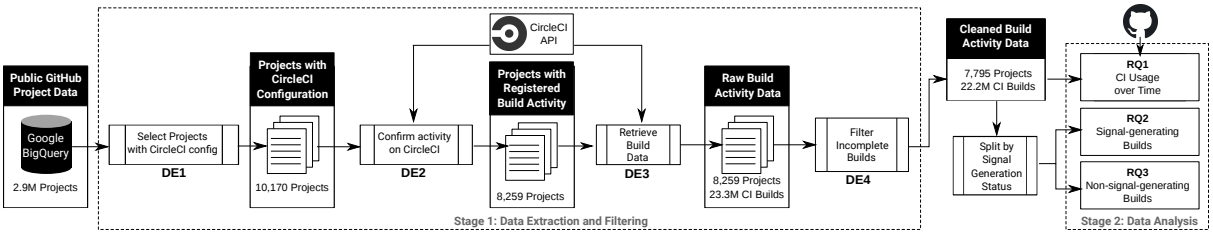


Figure 7.1: An overview of the approach we followed for data analysis

branching model of a software project. A low success rate of builds in the main branch of development in a multi-person software team means that the team is blocked from contributing to the project most of the time. However, the success rate in a feature branch where only one developer is actively working may not be that important. CircleCI recommends a success rate of 90% or better in the main branch.

Throughput. The number of CI builds that are performed during a given period of time. Depending on the development model and the team size, throughput may vary across projects. However, this metric gives an indication of expected total server load and network bandwidth usage for the CI service provider. CircleCI does not suggest a specific number of daily workflows to strive for the software teams.

These four indicators provide insight into the maturity of CI adoption among the CircleCI user population. Individual software organizations also may find these indicators useful to understand how their CI usage pattern compares with others. CircleCI have started providing these metrics to their users recently on a per-workflow basis within the CircleCI web interface. We use these four indicators in our study to characterize the growth of CI usage over the years.

7.3 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).

7.3.1 Subject Systems/Communities

With the popularity of CI as a software development practice, many cloud-based providers have offered CI services. In a Forrested Research report,[6] five leaders were identified among cloud-native CI tool providers after evaluating 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. Out of these services, according to GitHub marketplace statistics,[7] CircleCI has the highest number of installs (748k installs to-date) in the CI category. 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.[8]

Stage 1: Data Extraction and Filtering

Figure 7.1 provides and overview of our data extraction and filtering approach. We describe each step below. In order to arrive at reliable conclusions representing the full workload of a typical CI service provider, it is important that we access all publicly available build data for projects that use CircleCI.

Therefore, we start by querying for projects that use CircleCI in the public GitHub dataset on Google BigQuery,[9] 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 7.1). From this query, we retrieve a corpus of 10,170 projects with a CircleCI configuration file, out of 2,991,522 open source projects in the dataset.

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

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

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

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

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

at least one CI build in their lifetime (see DE2 of Figure 7.1). A corpus of 8,259 projects survive this filter.

After that, via the CircleCI API, we download metadata of all builds across these filtered projects. a total of 23,330,690 build records have been retrieved (see DE3 of Figure 7.1). Then, we remove builds that have incomplete fields and builds that were started after December 31st, 2020 (see DE4 of Figure 7.1). We chose this cutoff date to allow only completed years into our dataset for analysis. We use the 22,238,413 unique builds spanning 7,795 projects for further analysis.

Stage 2: Data Analysis

Figure 7.1 provides an overview of our data analysis approach. We describe each step below.

To answer RQ1, we compute the project level values for the four indicators defined in Section 7.2. Then, we plot the growth of CircleCI usage and indicators for the studied period of 2012–2020 using bar charts and line graphs.

To answer RQ2, we focus on signal-generating builds. Here, we use the Scott-Knott Effect Size Difference (ESD) test [86] to cluster build steps into statistically distinct ranks because this test considers the effect size when clustering build steps into ranks. Furthermore, we use Mann–Whitney U test [98], and Cliff's Delta [99] because these tests allow us to compare the runtime distribution of build steps between heavy users and other users without making the assumption that values are normally distributed.

To answer RQ3, first, we manually inspect a sample of non-signal-generating builds. Then, we formulate queries to find the frequency of occurrence of different non-signal-generating builds in the full dataset.

7.4 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 that we use to address it, followed by the results that we observe.

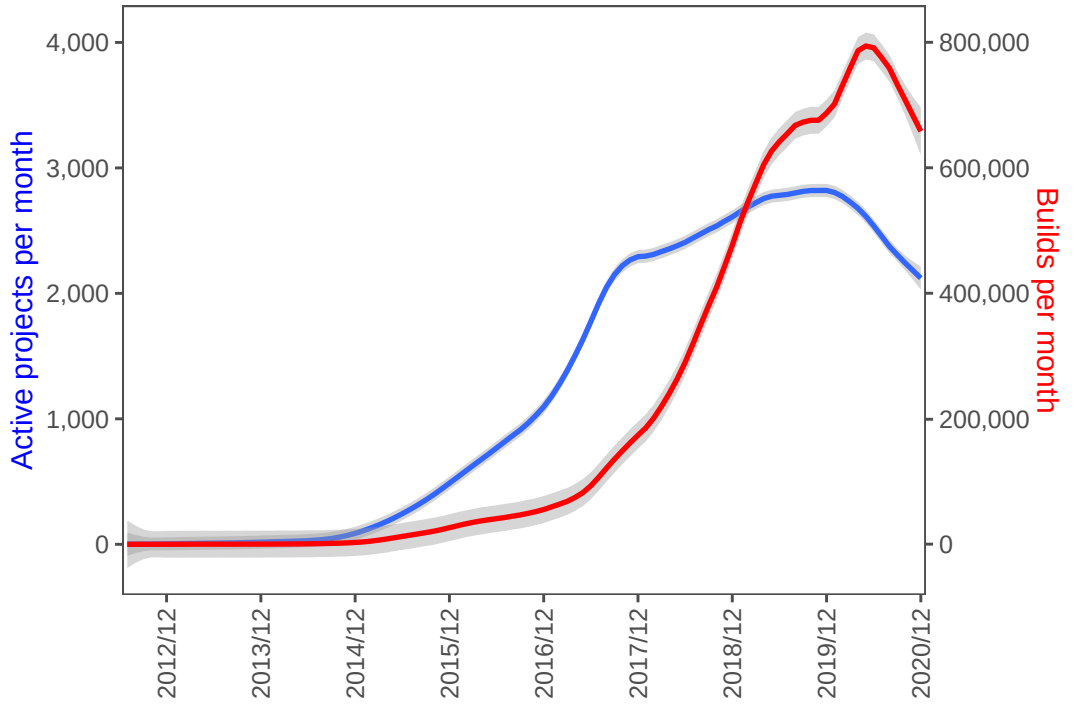


Figure 7.2: The growth of CircleCI usage during the period of 2012–2020. The number of projects that used CircleCI as the CI provider in each month is shown in blue. The number of CircleCI builds of these projects during each month of the studied period is shown in red. Both lines are Loess-smoothed curves with gray shaded areas indicating the 95% confidence interval.

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

RQ1: Approach. We plot the number of projects that are using CircleCI as their CI provider and the number of total builds that are run on CircleCI during each month throughout the studied period from 2012 to 2020. Then we plot the growth of CircleCI usage based on four indicators previously proposed by CircleCI,[5] namely: (a) Build Duration, (b) Mean Time to Recovery (MTTR), (c) Success Rate, and (d) Throughput.

RQ1: Results. Figure 7.2 shows the growth of CircleCI usage during the period of 2012–2020, 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 the studied projects grew over the years, reached a peak of 872,842 builds during the month of April 2020, and then declined.* We suspect that this trend can be explained by the number of active projects that were using CircleCI, which started plateauing in 2019 and declined in mid-2020. Competitive pricing

Table 7.1: Top five CI services used by projects that became inactive on CircleCI. Note that the sum of percentages exceed 100% because some projects are configured to use more than one CI service.

CI Service	#	%
Travis CI	214	50%
GitHub Actions	148	35%
AppVeyor	69	16%
Scrutinizer	11	2%
SemaphoreCI	10	2%

and new features offered by other service providers such as GitLab CI and GitHub Actions—GitHub's own automation service—may have contributed to this exodus users from CircleCI. For instance, compared to the 250 free minutes of build time per month in CircleCI, GitHub Actions provide 20 free parallel builds and unlimited build minutes for every open source project. There are also other community and technical factors at play. For example, Widder et al. [100] observe that projects with more pull requests tend to be less likely to abandon a CI service suggesting that the projects get value out of CI by using it to evaluate external contributions via pull requests. Similarly, they find that projects with a longer build duration 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 that were inactive on CircleCI 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.* Table 7.1 shows the top five CI services that were used by projects that became inactive on CircleCI. The inactive projects on CircleCI had configured at least 20 different CI services to report results back to GitHub.

Figure 7.3 shows the number of builds on the two different CircleCI platforms during the 2012–2020 time period. The second version of the CircleCI platform[10] provided users with

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

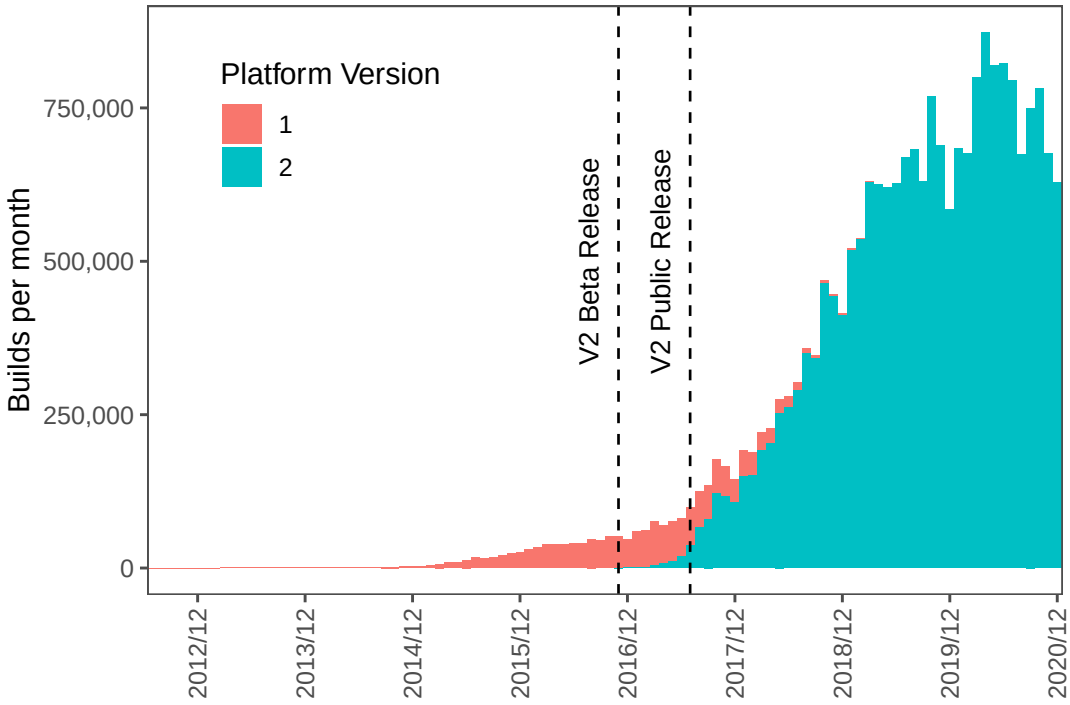
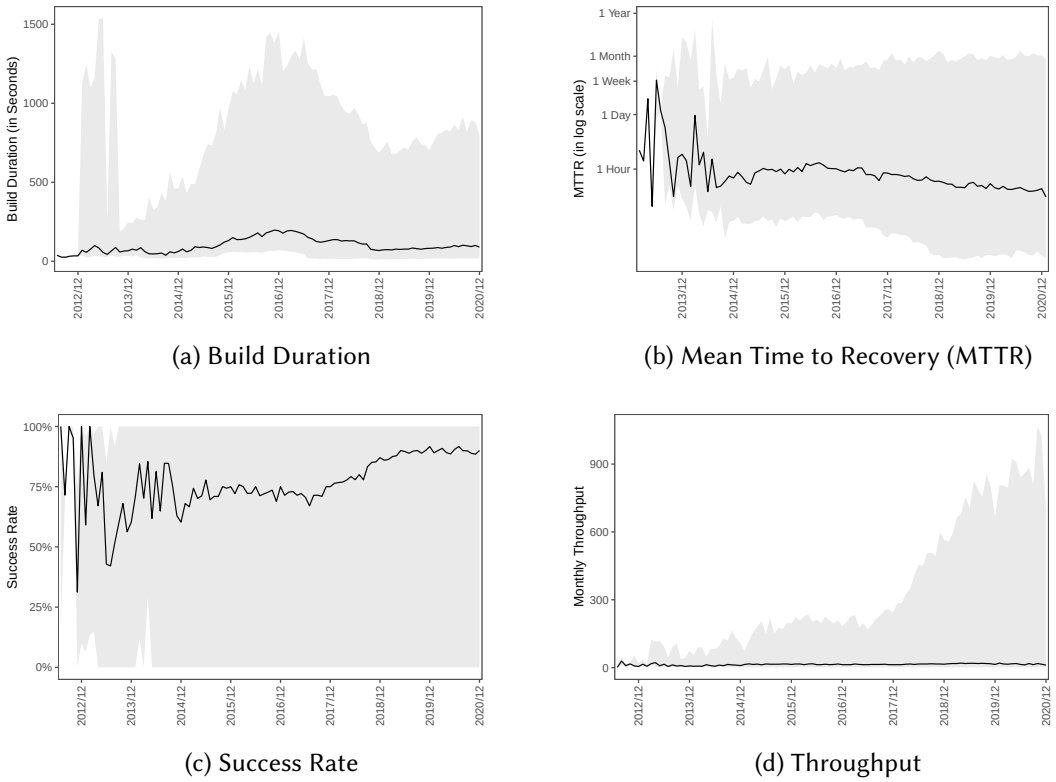


Figure 7.3: Number of builds on CircleCI platforms 1 and 2 during the 2012–2020 time period.

additional technical capabilities such as 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 users in July 2017. Figure 7.3 shows that these dates coincide with sharp changes in the trend of CircleCI build activity.

***Observation 3**: While median project-level indicator values have remained stable, indicators like throughput and build duration have been increasing in the heaviest users of the platform.* Figure 7.4 shows the evolution of four project-level indicators over the studied period. The black lines and gray bands show the median values and 90% confidence intervals of each metric across the projects, respectively.

Figure 7.4a shows that the median build duration stayed below 200 seconds throughout the studied period. However, 5% of studied projects took at least 25 minutes to build. These large projects might benefit from CI acceleration techniques that have been proposed in prior research [101].



(a) Build Duration

(b) Mean Time to Recovery (MTTR)

(c) Success Rate

(d) Throughput

Figure 7.4: 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.

Figure 7.4b shows that the median MTTR fluctuates before December of 2014 because of the low number of observations. However, later in the studied period, the median MTTR stayed in the range of 12–87 mins. For the slowest 5% of projects, the MTTR was at least one week. As shown in prior research [53], 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 the broken builds. For the benefit of software teams who are truly struggling to repair builds, research in automatically repairing build breakage [33, 43], providing developer assistance for build breakage resolution [32], and more broadly automated program repair [102] can be incorporated into CI services to fix build breakages quickly thereby reducing the MTTR.

Figure 7.4c shows that although the median success rate fluctuates before December of 2014, the rate gradually increases later in the studied period and ranges between 67%–92% of all builds. This demonstrates that in the majority of CI builds and at an increasing rate, newly introduced changes are not causing any build failures. If these changes that are not triggering

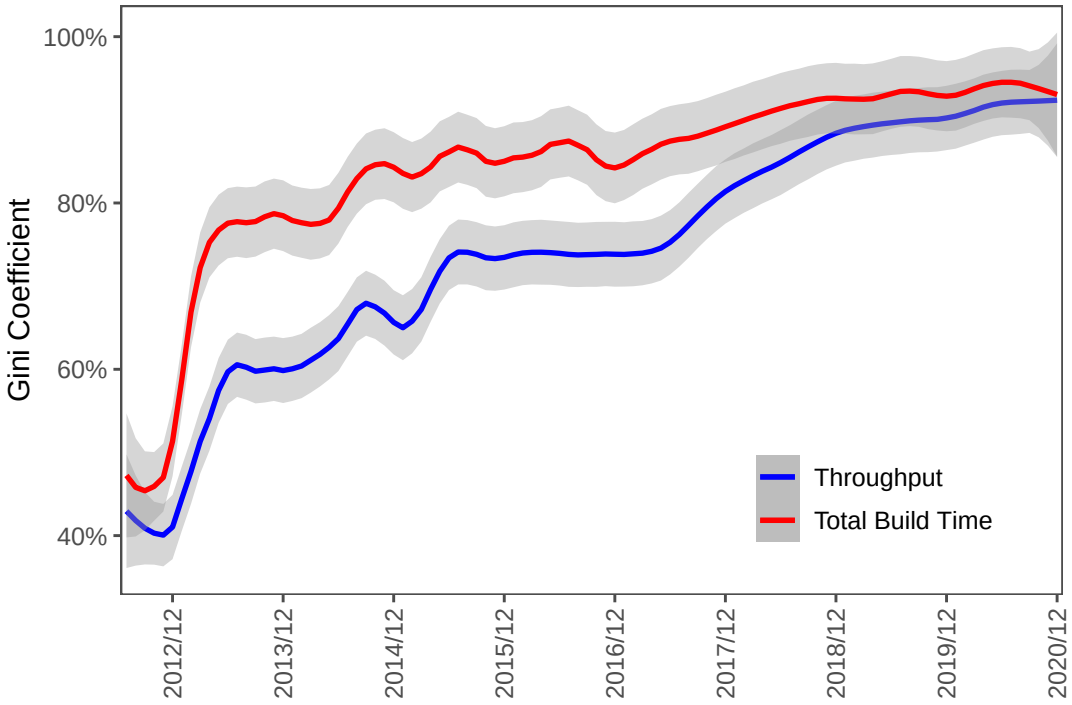


Figure 7.5: Gini coefficient computed using throughput and total build time during the 2012–2020 time period. Inequality of CI users is increasing.

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. [66] have a proposed machine learning approach to identifying such commits that can be skipped. Similarly, Jin and Servant [103] have proposed to reduce the high cost of CI by running fewer builds, while running as many failing builds as early as possible. 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 7.4d shows that the median throughput, i.e., the number of builds that are executed, 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.

***Observation 4**: Inequality of build execution and resource consumption has steadily increased over time.* To further investigate the imbalance of CI usage across users, we compute the Gini coefficient [104], a popular measure of inequality, during each month of the studied period. We calculate the Gini coefficient of the throughput and total build time. The Gini

Table 7.2: Distribution of Build Outcome. Global percentage of each category is shown in brackets.

	Outcome	#	%
Signal-generating	Success	18,831,874	80.72%
	Failed	2,806,792	12.03%
	Sub Total	**(21,638,666)**	**92.75%**
Non-signal-generating	Canceled	849,954	3.64%
	Infrastructure fail	10,993	0.05%
	Timedout	17,917	0.08%
	No tests	36,184	0.16%
	Null	776,976	3.33%
	Sub Total	**(1,692,024)**	**7.25%**

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 resource consumption perspective.

Figure 7.5 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 mean that there is a smaller number of projects that run a larger proportion of builds and consume a greater proportion of the build time per month. Since these heavy users run CI builds frequently, and put a heavy burden on CI services' resources, it is important for CI providers like CircleCI to invest in approaches that optimize the workloads of heaviest users.

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

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

RQ2: Approach. 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

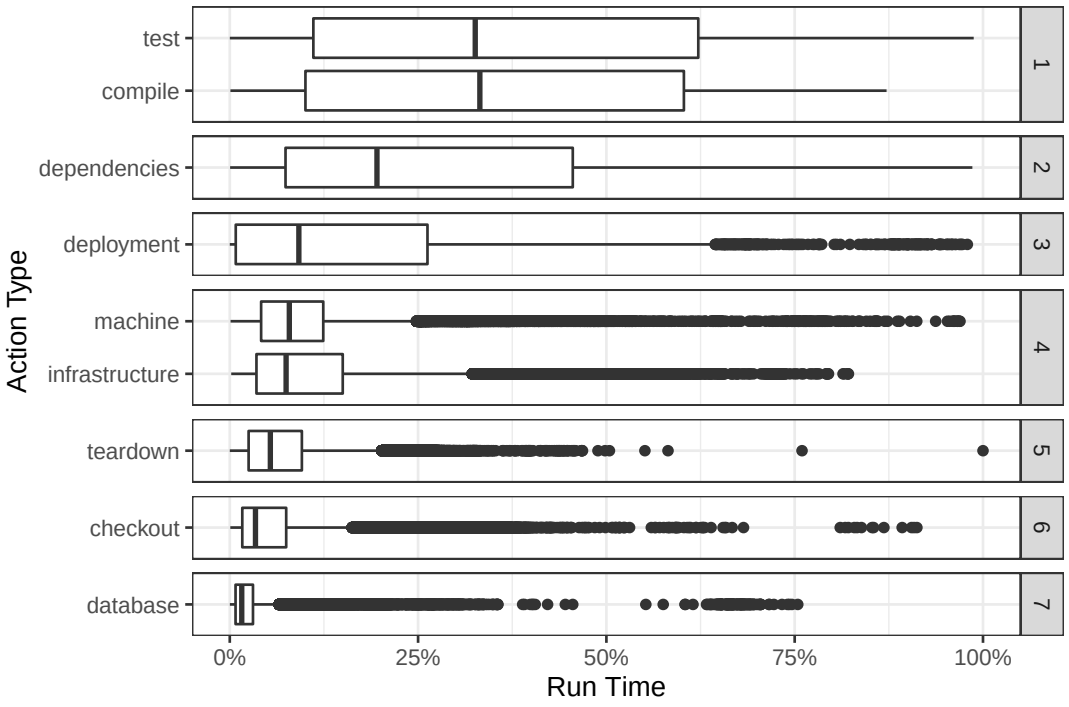


Figure 7.6: Runtime percentage of each action type in signal-generating builds.

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. To answer RQ2, we focus our analysis on the signal-generating builds.

The CircleCI API response provides start and end times of 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 [86] to cluster 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 users is different from non-heavy users. For this purpose, we label projects as heavy users or non-heavy users. We consider projects that consume a large proportion of the build time as heavy users. After identification of heavy users, we conduct Mann–Whitney U tests

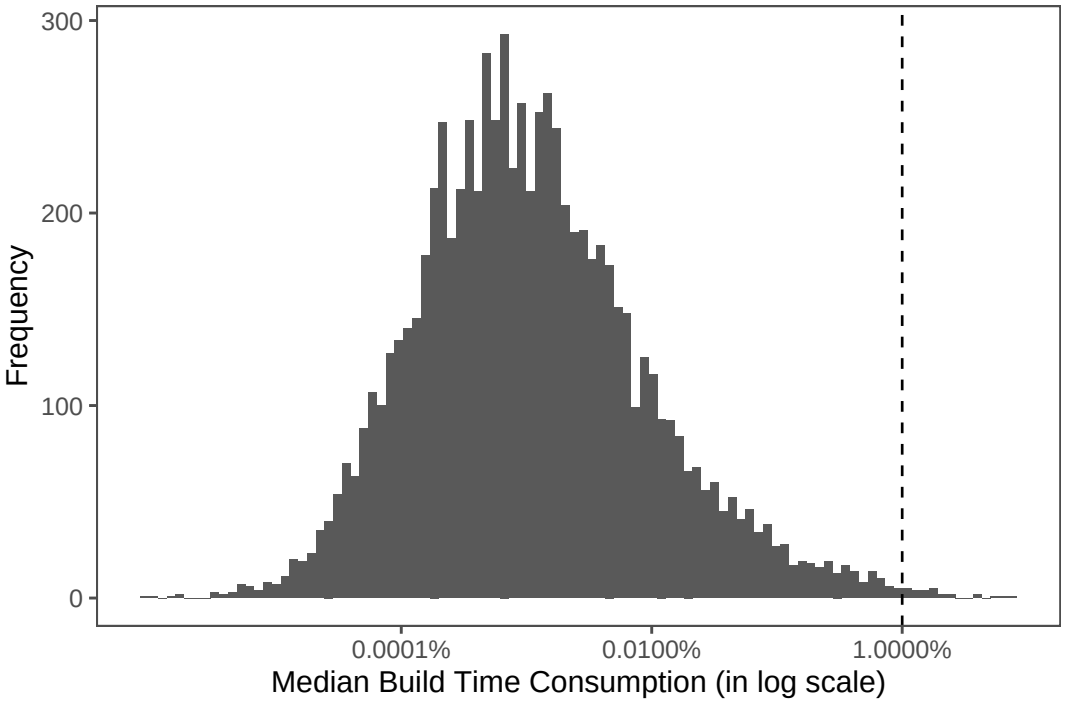


Figure 7.7: Distribution of median monthly build time consumption. Dashed black line at 1% marks the threshold for selecting heavy CI users.

and compute Cliff's delta for build time percentage of action types between heavy and non-heavy users.

RQ2: Results. The top three rows of Table 7.2 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 7.6 shows the run time percentage taken up by 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 the downloading of dependencies, which has a median of 19.5% and is a statistically distinct rank lower than compilation and testing. Therefore, focusing efforts to reduce the time taken for compilation and testing stages during the build will provide the most value for CI service providers. Approaches such as incremental builds, caching, build step skipping, and test selection are applicable here. We explore and evaluate two such CI acceleration techniques in Chapter 8.

However, as we observe in RQ1, the heaviest users exhibit different behaviour than the rest of the users. Figure 7.7 shows the distribution of monthly build time consumption across

Table 7.3: Domains of projects that heavily use CircleCI.

Domain	# Projects
Application Framework	6
Cloud Computing	4
Computer Security	3
Blockchain	3
Development Tools	3
Package Management	2
Machine Learning	2
Web Application	2
IoT	1
Database	1
Total	**27**

all studied projects. From there, we pick 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 users.

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

Table 7.3 shows that the 27 heavy users of CircleCI belong to a broad variety of domains, including cloud computing, security, and machine learning.

***Observation 6**: For all action types except deployment and compile, there were statistically significant differences between heavy users' builds and other builds, in terms of the runtime percentage.* Figure 7.8 shows the runtime percentage of each action type in signal-generating builds in heavy CI users vs others. The results of applying Mann-Whitney U test to the runtime percentage of each action type for the builds of same two user groups shows that there are significant differences in the usage patterns and resource consumption of heavy CI users compared to other users ($p < 2.2 \times 10^{-16}$) for all action types except *deployment* and *compile*, after applying Holm-Bonferroni correction [105]. The *deployment* and *compile* action types had insufficient evidence (i.e., one and zero observations respectively) among the builds of heavy users to reach any conclusions. Based on Cliff's delta, effect sizes were large for *in-*

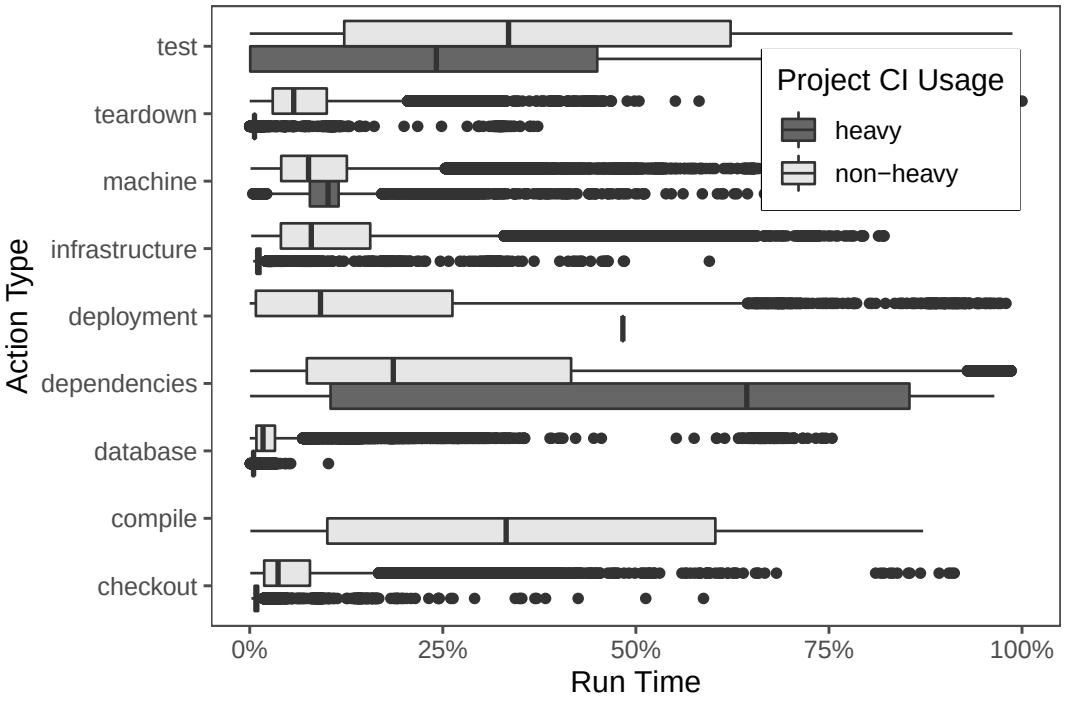


Figure 7.8: Runtime percentage of each action type in signal-generating builds in heavy CI users vs others.

frastructure, checkout, database, and *teardown* action types. The effect size was medium for *dependencies* action type, while for *machine* and *test* action types the effect sizes were small.

Because heavy users are likely to be lucrative as well, it may be worthwhile for the CI services to provide dedicated resources for the heavy users based on their requirements. For example, heavy users are spending more time downloading dependencies during the CI process compared to other users. Therefore, the CI service can provide more bandwidth during the dependency installation of the heavy users.

Approaches to make testing and compiling faster will benefit a large proportion of CI users. The heaviest users (and CI providers as a consequence) will benefit most from more bandwidth during dependency installation.

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

RQ3: Approach. First, we enumerate the outcome of builds that failed to generate a signal. With the help of more granular outcomes such as `infrastructure_fail` provided by the CircleCI API, build outcome enumeration in this section goes beyond the four basic outcomes introduced in Section 2.4. Then, we randomly select 150 random builds that terminated with non-signal-generating build outcomes. Then, one author manually analyzed the CircleCI API responses, build logs, and git commits of this sample of non-signal-generating builds to identify the circumstances that led to the non-signal-generating outcome. Next, another author checked the sample to confirm the findings. Any disagreements in the findings were resolved through discussions between the authors. After completing this step, we formulated queries to detect patterns in the CircleCI API responses to count the number of instances of each non-signal-generating incident.

RQ3: Results. The bottom six rows of Table 7.2 show the outcomes of non-signal-generating builds.

***Observation 7**: 50% of non-signal-generating builds (849,954 of 1,692,024) were abruptly terminated with a build outcome of `cancelled`.* These can be cancelled either on users' request 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, saving resources. However, we only found 64,077 builds cancelled automatically leaving 785,877 builds as cancelled by the users. Further research is needed to understand the reasons behind this high proportion of user-cancelled builds.

***Observation 8**: 36,184 builds terminated with a build outcome of `No tests`.* 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 avoid this behaviour interfering with their workflow, CircleCI users had to include a no-op command in the testing phase of their CI configuration. In version 2.0 of the CircleCI platform, this requirement was removed such that a test section was not necessary in the CI configuration. We were able

to verify that all builds in our dataset with the `No tests` outcome was executed on the 1.0 platform confirming that phenomenon.

***Observation 9**: 10,993 builds were unable to provide a signal due to infrastructure failures in the CI service.* According to the CircleCI support forum discussions, builds can terminate with the outcome `infrastructure_fail` due to faults in CircleCI internal infrastructure or other services that are contacted during the builds (e.g., GitHub and AWS). Although builds that terminate due to infrastructure failures are restarted automatically, it wastes time and resources. Investing in resources for increasing resiliency and fault tolerance in their back-end services may help CircleCI to mitigate these occurrences.

***Observation 10**: 17,917 builds timed out without running to completion.* If there is no output from any of the commands during the CI process for ten minutes, CircleCI considers it as timed out and the build is killed. We observe that builds can be timed out at any stage of the CI process (e.g., setting up the environment, installing dependencies, testing) in practice. Currently, users can extend the timeout in the CI configuration if they expect a build step to continue longer than ten minutes without any output. However, as a future user experience improvement, CircleCI could suggest to increase the timeout automatically if a build step regularly exceeds the time limit.

***Observation 11**: 776,976 builds were terminated prematurely due to other reasons and the build outcome was reported as `NULL`.* In 32,749 of these builds, builds were blocked by CircleCI because users were requesting to trigger builds beyond the resource limits that were allowed by their subscribed plans (e.g., no more user seats available on the plan, Docker Layer Caching feature is not available on the plan, the trial period has ended, needs more containers than plan allows, access to large machines is not available on the plan). At least 31,238 builds were not run to completion because the CI process was not properly configured (e.g., missing CI configuration file, specifying an unsupported Xcode version, GitHub was missing a CircleCI SSH key). Providing more documentation and guiding the users through the configuration process may reduce future occurrences of the builds terminating due to misconfiguration and users unknowingly requesting builds beyond their allocated resource limits.

> *Most builds are unable to provide a signal due to user interruptions. However, configuration errors and resource allocation issues on the services' side are also prevalent causing the builds to terminate prematurely.*

7.5 Practical Implications

Tool builders can use existing research to improve CI services. Rapidly growing build throughput and build durations of active CI users mean that build acceleration research can be used by CI services to reduce costs. A high success rate in builds means that service providers have a growing pool of candidate builds if they are using techniques proposed in research literature to skip builds. Automatic repairing techniques from the research literature can also be implemented by the CI providers in their product offerings to reduce the MTTR and therefore increasing developer productivity.

Researchers can focus on compilation and testing stages to accelerate CI. Reducing build durations will lead to increased developer productivity and savings on computational resources for the CI providers. Based on our observations the CI builds are spending a large proportion of the build run time in compilation and testing stages. Therefore, if researchers can identify ways to optimize compilation and testing steps effectively reducing their run time or skipping such steps altogether, it will greatly reduce the overall build duration providing the highest return on investment. Incremental builds (i.e., identifying steps that are impacted by the updated sources and run only these steps to generate correct build artifacts) is a well-known technique in this regard. In Chapter 8 we discuss challenges and benefits involved in enabling such techniques on continuous integration systems.

Further research to understand user cancellation of builds may help to improve user experience. Because most common reason for abruptly terminating CI builds is cancellation by the user, more research is needed to characterize this behaviour and identify ways to reduce user cancellation. More documentation and tutorials may help users to avoid cancelling started builds.

Investments to stabilize configuration and resource allocation may improve the robustness of CI services. Builds are also abruptly stopped due to configuration errors and server allocation issues. By investing on infrastructure and research efforts to mitigate build termination due to internal errors, the robustness and reliability of the CI service will improve. Furthermore, it will improve user experience helping to retain existing users and to attract new users for the CI service.

7.6 Threats to Validity

This section describes the threats to the validity of results in our case study.

7.6.1 Construct Validity

We use the mapping of commands and action types provided by CircleCI API to determine the CI stage that each command belongs to and then compute where time is spent during each signal-generating build. The accuracy of this mapping depends on the technique followed by CircleCI to label each command with an action type. To mitigate this threat, we manually inspected a sample of 50 commands and their assigned action types for consistency.

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

7.6.2 Internal Validity

We manually analyze API responses, build logs, and respective source code changes to characterize reasons that causes builds to terminate without providing a signal. There could be other reasons that caused the builds to abruptly terminate, yet are not revealed by the inspected artifacts. Although such instances cannot be fully accounted for, two authors independently verified the samples to make sure the findings are realistic.

7.6.3 External Validity

External validity concerns the generalizability of our results to other systems. In this study, we focus only on open source subject systems, which are hosted on GitHub and use CircleCI as the CI service provider. However, since CircleCI is one of the prominent CI services and provides very similar functionality to other leading CI service providers we believe our findings are applicable to other CI providers as well.

7.7 Chapter Summary

We bridge a gap in CI research literature by reporting on the perspective of the CI service provider. We analyze 23.3 million builds spanning 7,795 open source projects that use CircleCI service for nearly eight years and make the following observations:

- In terms of total number of builds and the number of users, the CI service has grown rapidly over the last few years. However, the growth has stagnated over the last year.
- Build duration (95th percentile = 25mins) and throughput (95th percentile = 900 builds per month) has grown to be very high for active users demonstrating opportunities for using build acceleration techniques.
- MTTR of one hour shows that automatic build repair will be beneficial for the developers to reduce the time taken for resolution.
- The high success rate (67%–92%) of the CI builds shows that there is room for automatically skipping builds to further reduce the load on CI servers.
- The compilation and testing actions consume a high proportion of the build times in the CI pipeline of signal-generating builds (Median 33%). This demonstrates two bottlenecks in the cloud CI services that can be optimized to improve efficiency.
- Non-signal-providing builds are terminated mostly due to user cancellation. However, configuration errors and resource allocation issues are also commonplace, highlighting the need for improving the robustness of CI services.

Our observations suggest that there are many existing research findings from research literature that CI service providers' can use to provide a better experience for software teams. Researchers can focus on the compilation and testing stages of the CI services to improve the efficiency. Also, research on configuration and resource allocation problems in large-scale distributed systems may improve the robustness of CI services.

Chapter 8

Accelerating Continuous Integration & Continuous Delivery

Note. An earlier version of the work in this chapter appears in the *IEEE Transactions on Software Engineering (TSE)* journal [101].

8.1 Introduction

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 [9] and improve software quality [7]. Open source [5] and proprietary [6, 67] 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 [30, 40, 84]. Indeed, Widder et al. [59] 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 [67] 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 [72–83]. Indeed, build dependency graphs may be *overspecified* [106, 116], leading to acceleration behaviour that is suboptimal (i.e., an unnecessary dependency forces potentially parallelizable steps to be executed sequentially), or worse, *underspecified* [75], 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 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:

[1] `https://yourbase.io/`

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.

Chapter organization. The remainder of the chapter is organized as follows: We demonstrate the potential of Kotinos using an example scenario in Section 8.2. Section 8.3 describes the Kotinos approach to accelerating CI builds. Section 8.4 presents our study of the fre-

Table 8.1: 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**

quency of activation of Kotinos acceleration types **(RQ1)**, while Section 8.5 reports on the extent to which Kotinos accelerations save time **(RQ2)**. Section 8.6 presents our study of the costs of the proposed accelerations **(RQ3)**. Section 8.7 discusses the broader implications of our study. Section 8.8 outlines the threats to validity. Finally, Section 8.9 draws conclusions.

8.2 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 8.1 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 8.1 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.

8.3 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 configuration file. These jobs are placed into a queue of pending jobs. When job processing nodes become available, they execute jobs from this queue. In this chapter, we focus on reducing the duration of the job processing phase.

We propose two acceleration techniques. Listings 8.3.1 and 8.3.2 provide a running example of the CI configuration of the *Wallaby* project (`https://github.com/reinteractive/wallaby`). Listing 8.3.1 shows the original steps specified for the Travis CI service. We migrated this CI configuration to the format of Kotinos (Listing 8.3.2), which simplifies parsing and enables acceleration features; however, the existing build system (Rake and Bundler in this case) remains unmodified.

```
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 8.3.1: Travis CI Configuration

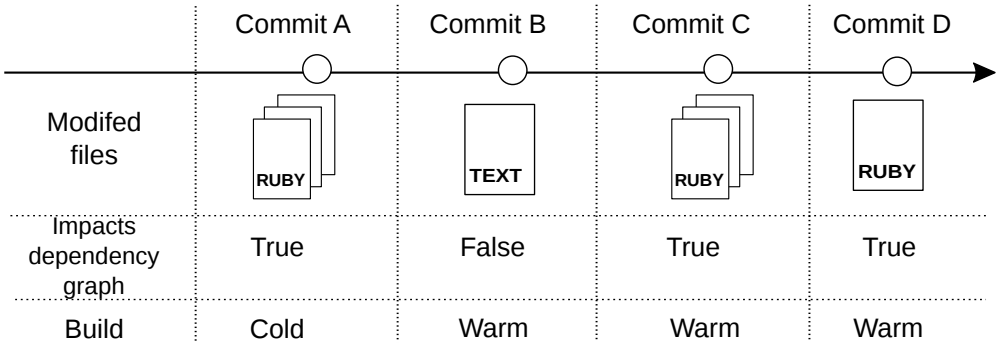


Figure 8.1: An example of commits in chronological order. In Commit A, all source code files are added. In Commit B, the `README.md` file is modified. In commits C and D, source code files that can affect multiple tests are modified.

8.3.1 Caching of the Build Environment (L1)

Before invoking the commands specified in a project's CI configuration, build job processing nodes need to be initialized and prepared. First, a programming language 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 8.1. 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 8.3.2). 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.

```
dependencies:
 build:
 - ruby:2.6.0
 - node:10.15.1

build_targets:
- commands:
  - apt-get update
  - apt-get install -y postgresql-client libpq-dev
  - gem install bundler
  - bundle install
  - bundle exec rake db:setup
  - bundle exec rake db:migrate
  - bundle exec rake spec

  name: daily_ci
  container:
   image: yourbase/yb_ubuntu:16.04

  environment:
  - DB=postgresql
  - BUNDLE_GEMFILE=gemfiles/Gemfile.rails-5.0
  - PGUSER=ci
  - PGPASSWORD=ci
  - RAILS_ENV=test
```

Listing 8.3.2: Kotinos Configuration

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.

8.3.2 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 [108]. 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., makefiles).

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 8.1, 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. [65, 75] 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 $\omega(G_{n-1}, G_n)$, 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

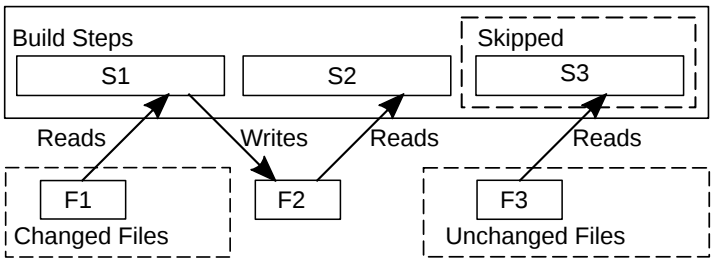


Figure 8.2: An example of how the Build Dependency Graph is used to identify which steps to skip. *S1* and *S2* cannot be skipped because they are **directly** and **transitively dependent** on the changed file *F1*, respectively. However, *S3* can be skipped because it does not depend on any changed files.

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 `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 8.2 provides an example of a BDG and highlights the behaviour when one file (file F1) is modified 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 file *F1*. Therefore, *S1* cannot be skipped.

Transitive dependency (S2). Step S1 reads from *F1* and writes to file *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 file *F3*, which was not modified by the change set being built. Since step *S3* does not have a direct or transitive dependency on modified files, step *S3* can be safely skipped.

8.4 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 first introduce the subject systems, then describe our approach to addressing RQ1, and finally, present our observations.

Subject Systems. The top three rows of Table 8.2 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 primarily use the commercial CI service that implements Kotinos techniques at its core. 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.

***Observation 1**: At least 87.9% of the builds in the studied systems are accelerated.* Table 8.3 shows the percentages of studied builds that activate the different types of acceleration. We find that 87.9%, 98.9%, and 97.6% of the studied builds are accelerated by at least one type of ac-

Table 8.2: 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

celeration 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

Table 8.3: 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

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 files 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 [88].

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 find a cached image of a predecessor for the commit that is being built.

***Observation 3**: Although less frequently activated than environment caching, step skipping accelerations are activated regularly as well.* Table 8.3 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.*

8.5 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.

8.5.1 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 8.4. 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 8.3 shows the distributions of build durations. Based on the p-values after applying Holm-Bonferroni correction [105], 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., $\delta > 0.58$), 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.

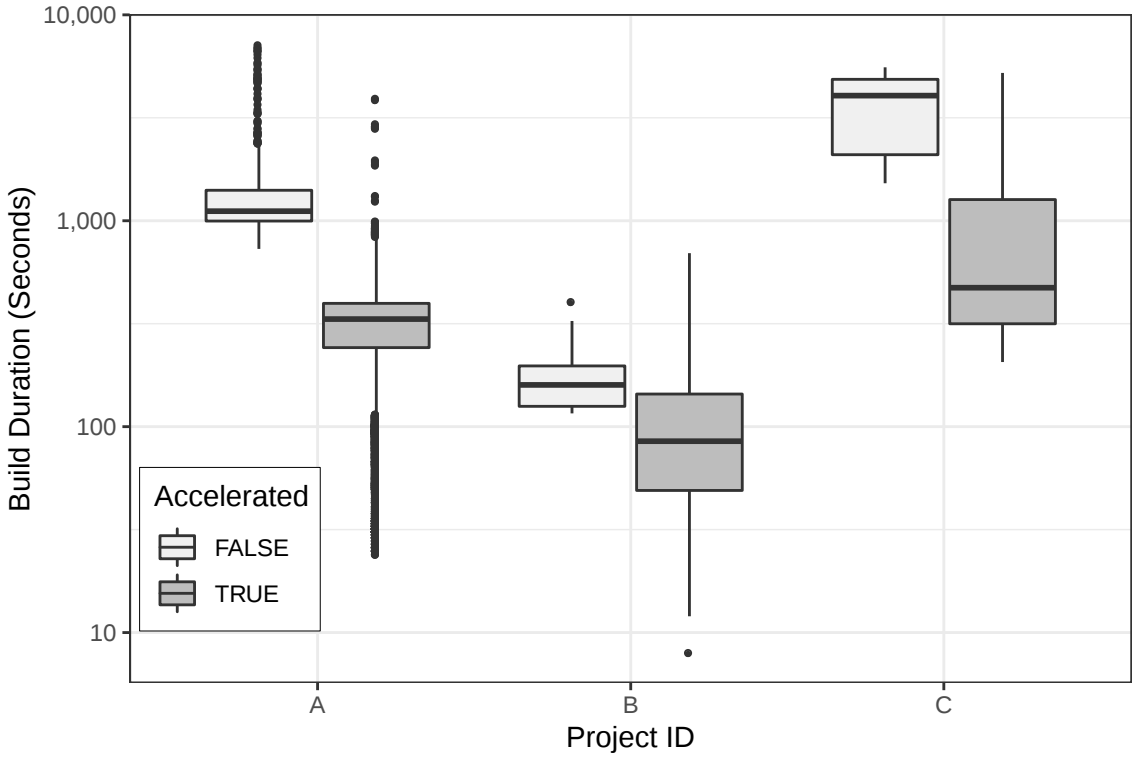


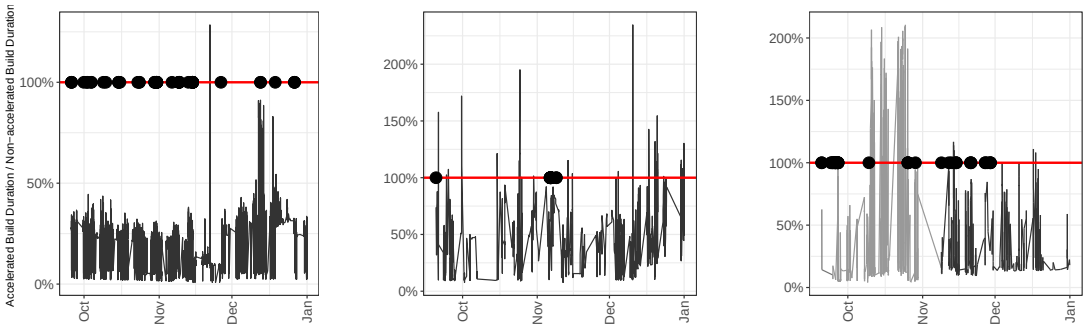
Figure 8.3: Distribution of durations in accelerated and non-accelerated builds across the three subject systems.

8.5.2 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.



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

Figure 8.4: 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 configuration. Project C has modified the CI configuration in mid-November as shown by the different shades of gray.

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 8.4 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 8.4 illustrate when warm builds

[2] `https://doi.org/10.6084/m9.figshare.12106845`

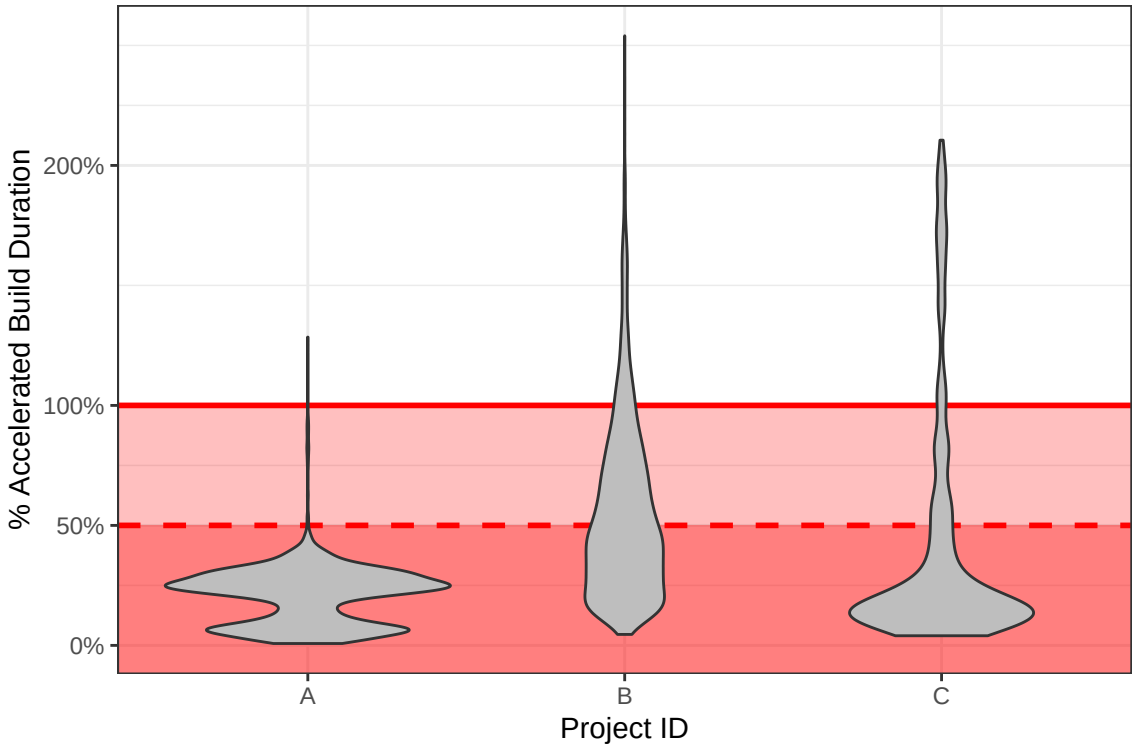


Figure 8.5: The gray-shaded violin plots show the kernel probability density of warm build durations as a percentage of cold builds across the three subject systems. The solid red line indicates the cold build duration (baseline). The builds in red-shaded area (below the solid red line) are faster than their cold counterparts. The builds in dark red-shaded area complete within 50% of their cold counterparts.

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 8.4c. 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 8.5 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 8.5, 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.

8.5.3 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 8.2 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 [86]. Finally,

[3] `https://www2.circleci.com/forrester-wave-leader-2019.html`

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 8.6 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 *RobotScouter*), 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 8.7 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.*

8.6 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 [67] and build correctness [109]. In prior sections, we quantified the benefits of Kotinos. In this section, we set out to quantify the costs in terms of resource utilization and correctness.

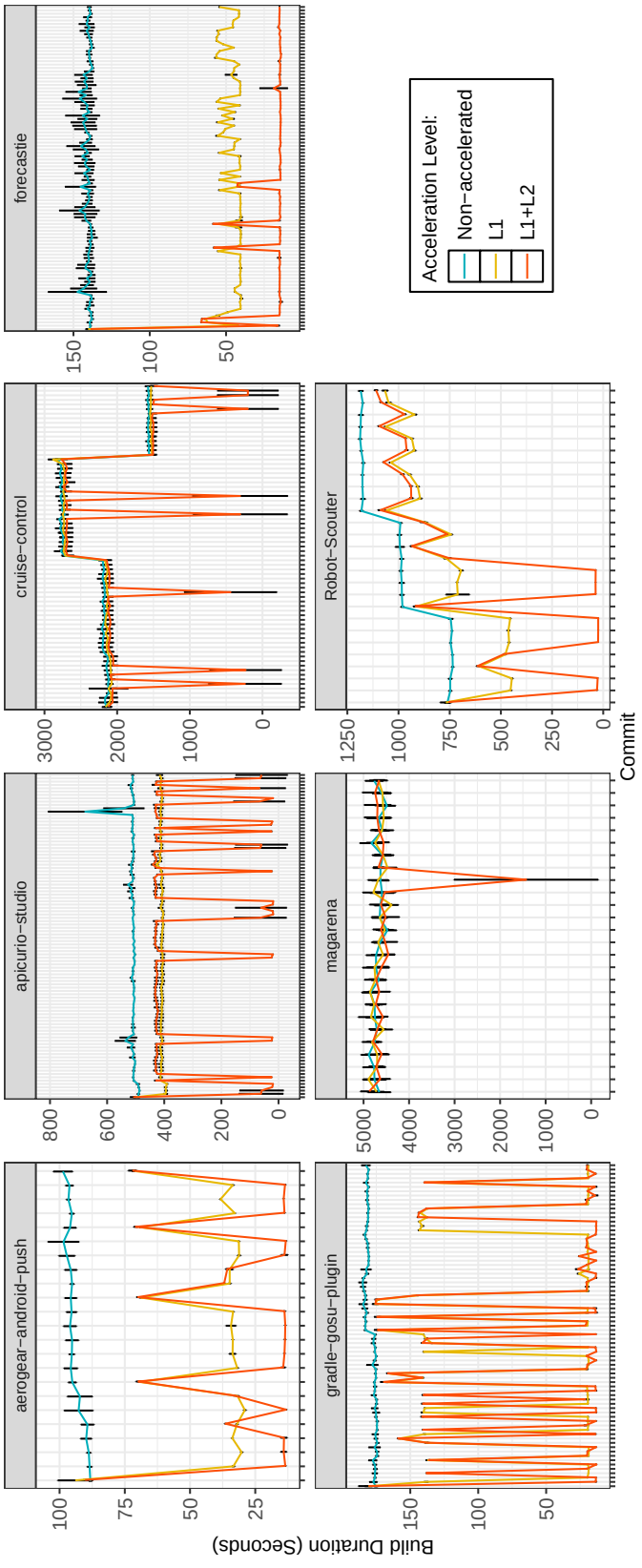


Figure 8.6: Median build time for each acceleration level in the open source subjects. Black vertical bars indicate the 95% confidence interval. (Acceleration Levels: L1 = Caching of the build environment, L2 = Skipping of unaffected build steps).

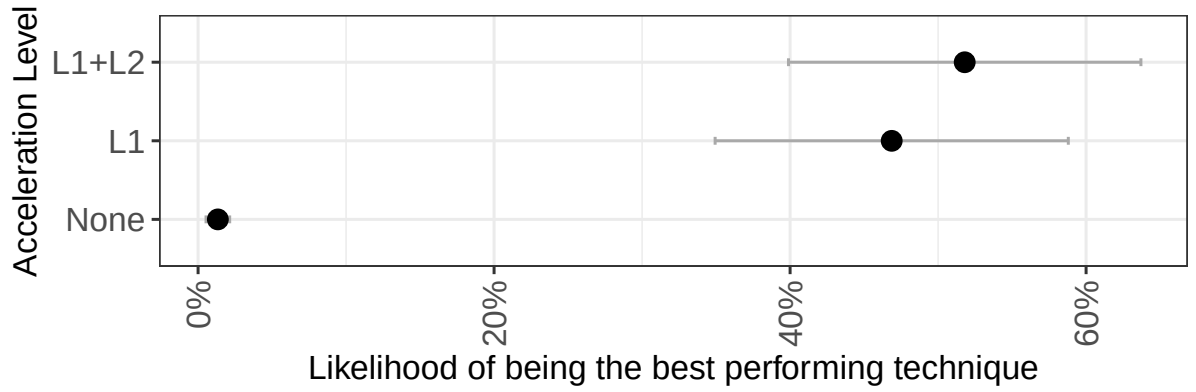


Figure 8.7: The likelihood of each acceleration technique appearing in the top rank. Circles indicate the median, while the error bars indicate the 95% confidence interval.

8.6.1 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 first measure the resource overhead.

Approach. We study the seven open source systems from Section 8.5. 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 Kotinos processes: (1) system call monitoring; (2) graph creation/update; and (3) graph traversal. To compute storage usage, we first 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 Kotinos. The size of the image at this point is recorded as the baseline. Then, each studied commit is built using Kotinos, 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 8.4 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-Scouter* and *magarena* systems have greater CPU and memory usage (median 693 MB – 2.2 GB). The logs indicate that these two subject systems made fivefold as many system calls as other systems, leading to a larger memory footprint when parsing system calls and inferring the BDG.

Table 8.4: 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

Table 8.5: 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 8.5 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 identified that *Maven* generated a JAR file that included all the transitive dependencies of the final 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 file system of Docker, the size of the image on disk is further reduced, effectively minimizing storage costs.

8.6.2 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., "flaky") 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.*

8.7 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 [97, 110].

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 benefit. As long as a software project has a build script that specifies 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 benefiting from modern CI acceleration.

Software teams can immediately benefit 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 specifications. The users only need to introduce a high-level configuration file (e.g., Listing 8.3.2), which invokes steps in existing build files. This approach of not modifying existing project artifacts means that teams are able to immediately derive benefits from migration to Kotinos without a substantial initial investment (e.g., migration of build tools [76, 86]) and with minimal disruptions to development activities.

8.8 Threats to Validity

This section describes the threats to the validity of our case study-based evaluations in Sections 8.4 and 8.5.

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 benefit systems implemented using a broad variety of languages and build tools.

Construct Validity. Threats to construct validity refer to the relationship between theory and observation. Infrastructure used by Kotinos could be very different from the CircleCI one. In other words, it may be possible that, if leveraging the infrastructure of CircleCI or other cloud services, the benefits could be less evident. To mitigate this, we run non-accelerated builds in a non-instrumented environment with vanilla OS to replicate the Circle CI infrastructure as much as possible. Therefore, the performance results relative to the non-accelerated environment should generalize, although the absolute values may differ.

8.9 Chapter Summary

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 specified dependencies in build configuration files; 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 significant 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 specification file 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 file 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 files 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 re-created 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 benefits (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 fulfill the above conditions can benefit from build acceleration.

Chapter 9

Final Conclusion & Future Work

This chapter provides a summary and concluding remarks regarding the contributions of this thesis in Section 9.1, while Section 9.2 discusses future work.

9.1 Thesis Summary

In this thesis, we investigate the **robustness** and **efficiency** of CI/CD services in five empirical studies. For this purpose, we leverage data stored in software repositories and within the CI/CD services. Below, we reiterate our main findings.

9.1.1 Usage of Features in CI/CD Environments

As a community, knowing how CI/CD is being used in practice is important for several reasons. Researchers will be able to target elements of CI/CD that are of greater impact to users of CI/CD. Service providers and tool builders will be able to tailor their solutions to fit the needs of target users.

Analysis of the features that are adopted by projects reveals that explicit deployment code is rare. The CI/CD code that configures job processing nodes appear in the most projects and accounts for the most modifications. Supporting the configuration of job processing nodes or reducing the complexity of deployment configuration via tooling would have the most immediate impact. Moreover, research and tooling for CI/CD configurations should focus on the creation of an initial specification rather than supporting specification maintenance because the configuration files are modified rarely.

9.1.2 Misuse of Features in CI/CD Environments

Like other software artifacts, CI/CD specifications can be misused. If improperly configured, CI/CD jobs may have unintended behaviour, resulting in broken or incorrect builds. Violating the semantics of CI/CD specifications could also introduce maintenance and comprehensibility problems. Furthermore, the CI/CD providers may be unable to optimize the provisioning of CI/CD job processing nodes for specifications where semantics are violated.

Hansel, our anti-pattern detector, can detect misuse and misconfiguration of CI/CD specifications. Hansel-detected anti-patterns can be removed (semi-)automatically with Gretel to avoid the consequences of CI/CD misuse and misconfiguration. Automated fixes for CI/CD anti-patterns are often accepted by developers and integrated into their projects, implying that Hansel & Gretel produce patches that are of value to active development teams.

9.1.3 Noise and Heterogeneity in CI/CD Build Data

Without a closer analysis of the nature of CI/CD build outcome data, practitioners and researchers are likely to make two critical assumptions: (1) build results are not noisy; however, passing builds may contain failing or skipped jobs that are actively or passively ignored; and (2) builds are equal; however, builds vary in terms of the number of jobs and configurations.

To investigate the degree to which these assumptions about build breakage hold, we perform an empirical study of 3.7 million build jobs spanning 1,276 open source projects. We find that: (1) 12% of passing builds have an actively ignored failure; (2) 9% of builds have a misleading or incorrect outcome on average; and (3) at least 44% of the broken builds contain passing jobs, i.e., the breakage is local to a subset of build variants. Like other software archives, build data is noisy and complex. Analysis of CI/CD build data requires nuance.

9.1.4 CI/CD Service Providers' Perspective

Identifying opportunities for managing resources efficiently will help CI/CD service providers to keep operational costs low while delivering fast and reliable CI/CD services. By making use of research findings, CI/CD providers can improve the usability of the service helping to attract new users and to retain existing ones.

Our observations suggest that CI/CD providers would benefit most from CI/CD build acceleration approaches that tackle long build durations and high throughput rates of heavy

CI/CD users. Approaches to make testing and compiling faster will benefit a large proportion of CI/CD users. Furthermore, we find that future investments to stabilize configuration and resource allocation may improve the robustness of CI/CD services.

9.1.5 Accelerating Continuous Integration / Continuous Delivery

While recent work has made several important advances in the acceleration of CI/CD 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 thesis, 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/CD 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. Results of our empirical evaluation suggest that migration to Hansel yields substantial benefits with low resource overhead and minimal investment of effort (e.g., no migration of build systems is necessary).

9.2 Future Work

We believe that this thesis has made contributions toward improving the robustness and efficiency of CI/CD services. Nevertheless, there is plenty of room for future research. We outline several promising avenues for future research below.

- In Chapter 4 we present the domains of subject systems that we study for CI/CD usage. As future work, we plan to study if the domain is a decisive factor that correlates with their practice of CI/CD. This includes studying how CI/CD is used in certain domains where practising CI/CD is challenging (e.g., testing in games or mobile applications) and the impact of programming languages/tools used in the domain for CI/CD. Furthermore, we plan to investigate ML-intensive systems as one of the studied domains since the role of CI/CD can be unique and challenging in a MLOps context.
- In Chapter 5, we deduce CI/CD specification anti-patterns by referring to the formal Travis CI documentation and informal documentation from the Travis CI user community (e.g., blogs, posts on Q&A sites, such as StackOverflow). The list of anti-patterns that we present in the paper is not exhaustive. Building upon our set of anti-patterns

as a starting point, future work can define anti-patterns using other data sources, e.g., developer surveys. Moreover, we plan to extend our study to systematically understand the impact of such misuse.

- Although Hansel & Gretel only support anti-pattern detection and removal in projects that use the Travis CI service, other popular CI/CD services, such as CircleCI, Wercker, and AppVeyor also use YAML DSLs for specifying CI/CD configuration. Thus, the anti-patterns that we define in Chapter 5 may also apply to these services. Future work may extend Hansel & Gretel to support these other CI/CD service providers.
- Hansel & Gretel support anti-pattern detection and removal in projects that use NPM to manage dependencies. Future work may add support for Yarn and other package managers.
- In Chapter 6, we define seven metrics to analyze build breakage. Future work can build upon and extend our initial set of metrics to study noise and heterogeneity. Furthermore, future work may quantify the impact that noise and heterogeneity can have on common analyses of historical build data. Moreover, we plan to study and the purpose of `allow_failures` in builds.
- In Chapter 7, we observe that a larger proportion of the CI/CD runtime is spent on the compilation and testing stages in signal-generating builds. Future work may focus on techniques for accelerating testing and compilation steps in the CI/CD pipeline to yield the largest reductions in CI/CD workload costs for service providers and feedback delays for users.
- Furthermore, in Chapter 7, we identify that approaches for automatic program repair and build acceleration will help to reduce CI/CD operational costs and to improve user experience. Future work may further empirically validate these findings.
- Currently, the CI/CD acceleration approach that we present in Chapter 8 requires the projects to fulfill several conditions in order to achieve robust acceleration. For example, candidate projects should use a build tool that supports deterministic dependency resolution. Test suites in the projects should be isolated and idempotent. Conditional behaviour during the builds should be kept to a minimum. Future work may look into approaches that will allow users to relax these conditions.
- Expanding on our observations of cloud CI/CD services in this thesis, we plan on an in-depth study of migration between CI tools and technologies. Ultimately, we aim to integrate the tools developed in different chapters as a comprehensive CI solution.

Bibliography

[1] Chuck Rossi, Elisa Shibley, Shi Su, Kent Beck, Tony Savor, and Michael Stumm. Continuous deployment of mobile software at Facebook (showcase). In: *Proceedings of Symposium on Foundations of Software Engineering (FSE)*, 2016. DOI: 10.1145/2950290.2994157.

[2] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. Usage, costs, and benefits of continuous integration in open-source projects. In: *Proceedings of International Conference on Automated Software Engineering (ASE)*, pp. 426–437. 2016. DOI: 10.1145/2970276.2970358.

[3] Bram Adams and Shane McIntosh. Modern release engineering in a nutshell: why researchers should care. In: *Proceedings of International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 5, pp. 78–90. 2016. DOI: 10.1109/SANER.2016.108.

[4] Paul M. Duvall, Steve Matyas, and Andrew Glover. *Continuous Integration: Improving Software Quality and Reducing Risk*. Pearson Education, 2007.

[5] Chris AtLee, Lukas Blakk, John O'Duinn, and Armen Zambrano Gasparnian. Firefox release engineering. In: Amy Brown and Greg Wilson (eds.), *The Architecture of Open Source Applications: Structure, Scale, and a Few More Fearless Hacks*, Creative Commons, 2012. URL: http://www.aosabook.org/en/ffreleng.html.

[6] Hyunmin Seo, Caitlin Sadowski, Sebastian Elbaum, Edward Aftandilian, and Robert Bowdidge. Programmers' build errors: a case study (at Google). In: *Proceedings of International Conference on Software Engineering (ICSE)*, pp. 724–734. 2014. DOI: 10.1145/2568225.2568255.

[7] Ade Miller. A hundred days of continuous integration. In: *Proceedings of the Agile Conference*, pp. 289–293. 2008.

[8] Carmine Vassallo, Gerald Schermann, Fiorella Zampetti, Daniele Romano, Philipp Leitner, Andy Zaidman, Massimiliano Di Penta, and Sebastiano Panichella. A tale of CI build failures: an open source and a financial organization perspective. In: *Proceedings*

of International Conference on Software Maintenance and Evolution (ICSME), pp. 183–193. 2017. DOI: 10.1109/icsme.2017.67.

[9] Bogdan Vasilescu, Yue Yu, Huaimin Wang, Premkumar Devanbu, and Vladimir Filkov. Quality and productivity outcomes relating to continuous integration in GitHub. In: *Proceedings of Joint Meeting on European Software Engineering Conference and Symposium on Foundations of Software Engineering (ESEC/FSE)*, pp. 805–816. 2015. DOI: 10.1145/2786805.2786850.

[10] Jez Humble and David Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Pearson Education, 2010.

[11] Gustavo Pinto, Fernando Castor, Rodrigo Bonifacio, and Marcel Rebouças. Work practices and challenges in continuous integration: a survey with Travis CI users. *Software: Practice and Experience* 48(12), 2018, pp. 2223–2236. DOI: 10.1002/spe.2637.

[12] Yang Luo, Yangyang Zhao, Wanwangying Ma, and Lin Chen. What are the factors impacting build breakage? In: *Proceedings of the Web Information Systems and Applications Conference*, 2017. DOI: 10.1109/wisa.2017.17.

[13] Mathias Meyer. Continuous integration and its tools. *IEEE Software* 31(3), 2014, pp. 14–16. DOI: 10.1109/MS.2014.58.

[14] Daniel Ståhl and Jan Bosch. Modeling continuous integration practice differences in industry software development. *Journal of Systems and Software (JSS)* 87, 2014, pp. 48–59. DOI: 10.1016/j.jss.2013.08.032.

[15] Gustavo Pinto, Marcel Reboucas, and Fernando Castor. Inadequate testing, time pressure, and (over) confidence: a tale of continuous integration users. In: *Proceedings of International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*, pp. 74–77. 2017. DOI: 10.1109/chase.2017.13.

[16] Omar Elazhary, Colin Werner, Ze Shi Li, Derek Lowlind, Neil A. Ernst, and Margaret-Anne Storey. Uncovering the benefits and challenges of continuous integration practices. *IEEE Transactions on Software Engineering (TSE)*, 2021. DOI: 10.1109/tse.2021.3064953.

[17] Michael Hilton, Nicholas Nelson, Timothy Tunnell, Darko Marinov, and Danny Dig. Trade-offs in continuous integration: assurance, security, and flexibility. In: *Proceedings of Joint Meeting on European Software Engineering Conference and Symposium on*

Foundations of Software Engineering (ESEC/FSE), pp. 197–207. 2017. DOI: 10.1145/3106237.3106270.

[18] Moritz Beller, Georgios Gousios, and Andy Zaidman. Oops, my tests broke the build: an explorative analysis of travis CI with GitHub. In: *Proceedings of International Conference on Mining Software Repositories (MSR)*, pp. 356–367. 2017. DOI: 10.1109/msr.2017.62.

[19] Sebastian Elbaum, Gregg Rothermel, and John Penix. Techniques for improving regression testing in continuous integration development environments. In: *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*, pp. 235–245. 2014. DOI: 10.1145/2635868.2635910.

[20] José Campos, Andrea Arcuri, Gordon Fraser, and Rui Abreu. Continuous test generation: enhancing continuous integration with automated test generation. In: *Proceedings of International Conference on Automated Software Engineering (ASE)*, pp. 55–66. 2014. DOI: 10.1145/2642937.2643002.

[21] Stefan Dösinger, Richard Mordinyi, and Stefan Biffl. Communicating continuous integration servers for increasing effectiveness of automated testing. In: *Proceedings of International Conference on Automated Software Engineering (ASE)*, pp. 374–377. 2012. DOI: 10.1145/2351676.2351751.

[22] Fabrizio Cannizzo, Robbie Clutton, and Raghav Ramesh. Pushing the boundaries of testing and continuous integration. In: *Proceedings of the Agile Conference*, pp. 501–505. 2008. DOI: 10.1109/Agile.2008.31.

[23] William J. Brown, Hays W. McCormick III, and Scott W. Thomas. *AntiPatterns and Patterns in Software Configuration Management*. John Wiley & Sons, Inc., 1999.

[24] Rian Shambaugh, Aaron Weiss, and Arjun Guha. Rehearsal: a configuration verification tool for puppet. In: *Proceedings of International Conference on Program. Language Des. and Implementation (PLDI)*, pp. 416–430. 2016. DOI: 10.1145/2908080.2908083.

[25] Tushar Sharma, Marios Fragkoulis, and Diomidis Spinellis. Does your configuration code smell? In: *Proceedings of International Conference on Mining Software Repositories (MSR)*, pp. 189–200. 2016. DOI: 10.1145/2901739.2901761.

[26] Eduard Van der Bent, Jurriaan Hage, Joost Visser, and Georgios Gousios. How good is your puppet? an empirically defined and validated quality model for puppet. In: *Proceedings of International Conference on Softw. Anal., Evolution Reengineering (SANER)*, pp. 164–174. 2018. DOI: 10.1109/SANER.2018.8330206.

[27] Akond Rahman and Laurie Williams. Characterizing defective configuration scripts used for continuous deployment. In: *Proceedings of International Conference on Softw. Testing, Validations, and Verification (ICST)*, pp. 34–45. 2018. DOI: 10.1109/ICST.2018.00014.

[28] Ajay Kumar Jha, Sunghee Lee, and Woo Jin Lee. Developer mistakes in writing Android manifests: an empirical study of configuration errors. In: *Proceedings of International Conference on Mining Software Repositories (MSR)*, pp. 25–36. 2017. DOI: 10.1109/msr.2017.41.

[29] Jürgen Cito, Gerald Schermann, John Erik Wittern, Philipp Leitner, Sali Zumberi, and Harald C. Gall. An empirical analysis of the Docker container ecosystem on GitHub. In: *Proceedings of International Conference on Mining Software Repositories (MSR)*, pp. 323–333. 2017. DOI: 10.1109/MSR.2017.67.

[30] Carmine Vassallo, Sebastian Proksch, Harald C. Gall, and Massimiliano Di Penta. Automated reporting of anti-patterns and decay in continuous integration. In: *Proceedings of International Conference on Software Engineering (ICSE)*, pp. 105–115. 2019. DOI: 10.1109/ICSE.2019.00028.

[31] Fiorella Zampetti, Carmine Vassallo, Sebastiano Panichella, Gerardo Canfora, Harald Gall, and Massimiliano Di Penta. An empirical characterization of bad practices in continuous integration. *Empirical Software Engineering (EMSE)* 25(2), 2020, pp. 1095–1135. DOI: 10.1007/s10664-019-09785-8.

[32] Carmine Vassallo, Sebastian Proksch, Timothy Zemp, and Harald C. Gall. Un-break my build: assisting developers with build repair hints. In: *Proceedings of International Conference on Program Comprehension (ICPC)*, pp. 41–51. 2018. DOI: 10.1145/3196321.3196350.

[33] Foyzul Hassan and Xiaoyin Wang. HireBuild: An Automatic Approach to History-Driven Repair of Build Scripts. In: *Proceedings of International Conference on Software Engineering (ICSE)*, pp. 1078–1089. 2018. DOI: 10.1145/3180155.3180181.

[34] Christian Macho, Shane McIntosh, and Martin Pinzger. Automatically repairing dependency-related build breakage. In: *Proceedings of International Conference on Software Analysis, Evolution, Reengineering (SANER)*, pp. 106–117. 2018. DOI: 10.1109/SANER.2018.8330201.

[35] Carmine Vassallo, Sebastian Proksch, Anna Jancso, Harald C. Gall, and Massimiliano Di Penta. Configuration smells in continuous delivery pipelines: a linter and a six-month study on GitLab. In: *Proceedings of Joint Meeting on European Software Engineering Conference and Symposium on Foundations of Software Engineering (ESEC/FSE)*, pp. 327–337. 2020. DOI: 10.1145/3368089.3409709.

[36] Radu Marinescu. Detection strategies: metrics-based rules for detecting design flaws. In: *Proceedings of International Conference on Software Maintenance (ICSM)*, pp. 350–359. 2004. DOI: 10.1109/ICSM.2004.1357820.

[37] Joshua Garcia, Daniel Popescu, George Edwards, and Nenad Medvidovic. Toward a catalogue of architectural bad smells. In: *Proceedings of International Conference on the Quality of Software Architectures: Architectures for Adaptive Software Systems*, pp. 146–162. 2009. DOI: 10.1007/978-3-642-02351-4_10.

[38] Naouel Moha, Yann-Gaël Guéhéneuc, Laurence Duchien, and Anne-Francoise Le Meur. Decor: a method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering (TSE)* 36(1), 2010, pp. 20–36. DOI: 10.1109/TSE.2009.50.

[39] Mansoor Alicherry and Angelos D. Keromytis. DoubleCheck: multi-path verification against man-in-the-middle attacks. In: *Proceedings of International Symposium on Computers and Communications (ISCC)*, 2009. DOI: 10.1109/iscc.2009.5202224.

[40] Akond Rahman, Chris Parnin, and Laurie Williams. The seven sins: security smells in infrastructure as code scripts. In: *Proceedings of International Conference on Software Engineering (ICSE)*, pp. 164–175. 2019. DOI: 10.1109/icse.2019.00033.

[41] Noureddine Kerzazi, Foutse Khomh, and Bram Adams. Why do automated builds break? an empirical study. In: *Proceedings of International Conference on Software Maintenance and Evolution (ICSME)*, 2014. DOI: 10.1109/icsme.2014.26.

[42] Michele Tufano, Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. There and back again: can you compile that

snapshot? *Journal of Software: Evolution and Process* 29(4), 2016, e1838. DOI: 10.1002/smr.1838.

[43] Foyzul Hassan, Shaikh Mostafa, Edmund S.L. Lam, and Xiaoyin Wang. Automatic building of java projects in software repositories: a study on feasibility and challenges. In: *Proceedings of International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2017. DOI: 10.1109/esem.2017.11.

[44] Ahmed E. Hassan and Ken Zhang. Using decision trees to predict the certification result of a build. In: *Proceedings of International Conference on Automated Software Engineering (ASE)*, 2006. DOI: 10.1109/ase.2006.72.

[45] Timo Wolf, Adrian Schroter, Daniela Damian, and Thanh Nguyen. Predicting build failures using social network analysis on developer communication. In: *Proceedings of International Conference on Software Engineering (ICSE)*, 2009. DOI: 10.1109/icse.2009.5070503.

[46] Irwin Kwan, Adrian Schroter, and Daniela Damian. Does socio-technical congruence have an effect on software build success? a study of coordination in a software project. *IEEE Transactions on Software Engineering (TSE)* 37(3), 2011, pp. 307–324. DOI: 10.1109/tse.2011.29.

[47] Panagiotis Dimitropoulos, Zeyar Aung, and Davor Svetinovic. Continuous integration build breakage rationale: travis data case study. In: *Proceedings of International Conference on Infocom Technologies and Unmanned Systems (Trends and Future Directions) (ICTUS)*, 2017. DOI: 10.1109/ictus.2017.8286087.

[48] John Downs, Beryl Plimmer, and John G. Hosking. Ambient awareness of build status in collocated software teams. In: *Proceedings of International Conference on Software Engineering (ICSE)*, 2012. DOI: 10.1109/icse.2012.6227165.

[49] Tijs van der Storm. Backtracking incremental continuous integration. In: *Proceedings of European Conference on Software Maintenance and Reengineering (CSMR)*, 2008. DOI: 10.1109/csmr.2008.4493318.

[50] Mahdis Zolfagharinia, Bram Adams, and Yann-Gaël 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), 2019, pp. 3933–3971. DOI: 10.1007/s10664-019-09709-6.

[51] Taher Ahmed Ghaleb, Daniel Alencar da Costa, Ying Zou, and Ahmed E. Hassan. Studying the impact of noises in build breakage data. *IEEE Transactions on Software Engineering (TSE)*. DOI: 10.1109/tse.2019.2941880.

[52] Moritz Beller, Georgios Gousios, and Andy Zaidman. TravisTorrent: synthesizing travis CI and GitHub for full-stack research on continuous integration. In: *Proceedings of International Conference on Mining Software Repositories (MSR)*, pp. 447–450. 2017. DOI: 10.1109/msr.2017.24.

[53] Keheliya Gallaba, Christian Macho, Martin Pinzger, and Shane McIntosh. Noise and heterogeneity in historical build data: an empirical study of Travis CI. In: *Proceedings of International Conference on Automated Software Engineering (ASE)*, pp. 87–97. 2018. DOI: 10.1145/3238147.3238171.

[54] Taher Ahmed Ghaleb, Daniel Alencar da Costa, and Ying Zou. An empirical study of the long duration of continuous integration builds. *Empirical Software Engineering (EMSE)* 24(4), 2019, pp. 2102–2139. DOI: 10.1007/s10664-019-09695-9.

[55] Thomas Durieux, Rui Abreu, Martin Monperrus, Tegawendé F. Bissyandé, and Luis Cruz. An analysis of 35+ million jobs of Travis CI. In: *Proceedings of International Conference on Software Maintenance and Evolution (ICSME)*, pp. 291–295. 2019. DOI: 10.1109/ICSME.2019.00044.

[56] Thomas Rausch, Waldemar Hummer, Philipp Leitner, and Stefan Schulte. An empirical analysis of build failures in the continuous integration workflows of java-based open-source software. In: *Proceedings of International Conference on Mining Software Repositories (MSR)*, pp. 345–355. 2017. DOI: 10.1109/msr.2017.54.

[57] Wagner Felidré, Leonardo Furtado, Daniel Alencar da Costa, Bruno Cartaxo, and Gustavo Pinto. Continuous integration theater. In: *Proceedings of International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pp. 1–10. 2019. DOI: 10.1109/ESEM.2019.8870152.

[58] Yangyang Zhao, Alexander Serebrenik, Yuming Zhou, Vladimir Filkov, and Bogdan Vasilescu. The impact of continuous integration on other software development practices: a large-scale empirical study. In: *Proceedings of International Conference on Automated Software Engineering (ASE)*, pp. 60–71. 2017. DOI: 10.1109/ase.2017.8115619.

[59] David Gray Widder, Michael Hilton, Christian Kästner, and Bogdan Vasilescu. A conceptual replication of continuous integration pain points in the context of Travis CI. In: *Proceedings of Joint Meeting on European Software Engineering Conference and Symposium on Foundations of Software Engineering (ESEC/FSE)*, pp. 647–658. 2019. DOI: 10.1145/3338906.3338922.

[60] Yue Yu, Huaimin Wang, Vladimir Filkov, Premkumar Devanbu, and Bogdan Vasilescu. Wait for it: determinants of pull request evaluation latency on GitHub. In: *Proceedings of International Conference on Mining Software Repositories (MSR)*, pp. 367–371. 2015. DOI: 10.1109/msr.2015.42.

[61] Carlene Lebeuf, Elena Voyloshnikova, Kim Herzig, and Margaret-Anne Storey. Understanding, debugging, and optimizing distributed software builds: a design study. In: *Proceedings of International Conference on Software Maintenance and Evolution (ICSME)*, pp. 496–507. 2018. DOI: 10.1109/icsme.2018.00060.

[62] Graham Brooks. Team pace keeping build times down. In: *Agile 2008 Conference*, pp. 294–297. 2008. DOI: 10.1109/agile.2008.41.

[63] Qi Cao, Ruiyin Wen, and Shane McIntosh. Forecasting the duration of incremental build jobs. In: *Proceedings of International Conference on Software Maintenance and Evolution (ICSME)*, pp. 524–528. 2017. DOI: 10.1109/icsme.2017.34.

[64] Michele Tufano, Hitesh Sajnani, and Kim Herzig. Towards predicting the impact of software changes on building activities. In: *Proceedings of International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*, pp. 49–52. 2019. DOI: 10.1109/ICSE-NIER.2019.00021.

[65] Rabe Abdalkareem, Suhaib Mujahid, Emad Shihab, and Juergen Rilling. Which commits can be CI skipped? *IEEE Transactions on Software Engineering (TSE)*, 2019. DOI: 10.1109/tse.2019.2897300.

[66] Rabe Abdalkareem, Suhaib Mujahid, and Emad Shihab. A machine learning approach to improve the detection of CI skip commits. *IEEE Transactions on Software Engineering (TSE)*, 2020. DOI: 10.1109/tse.2020.2967380.

[67] Hamed Esfahani, Jonas Fietz, Qi Ke, Alexei Kolomiets, Erica Lan, Erik Mavrinac, Wolfram Schulte, Newton Sanches, and Srikanth Kandula. CloudBuild: Microsoft's distributed and caching build service. In: *Proceedings of International Conference on Software Engineering Companion*, pp. 11–20. 2016. DOI: 10.1145/2889160.2889222.

[68] Yingling Li, Junjie Wang, Yun Yang, and Qing Wang. Method-level test selection for continuous integration with static dependencies and dynamic execution rules. In: *Proceedings of the International Conference on Software Quality, Reliability and Security*, pp. 350–361. 2019. DOI: 10.1109/qrs.2019.00052.

[69] August Shi, Peiyuan Zhao, and Darko Marinov. Understanding and improving regression test selection in continuous integration. In: *Proceedings of International Symposium on Software Reliability Engineering (ISSRE)*, 2019. DOI: 10.1109/issre.2019.00031.

[70] Shin Yoo and Mark Harman. Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability* 22(2), 2012, pp. 67–120. DOI: 10.1002/stv.430.

[71] Atif Memon, Zebao Gao, Bao Nguyen, Sanjeev Dhanda, Eric Nickell, Rob Siemborski, and John Micco. Taming google-scale continuous testing. In: *Proceedings of International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, 2017. DOI: 10.1109/icse-seip.2017.16.

[72] Shane McIntosh, Bram Adams, Meiyappan Nagappan, and Ahmed E. Hassan. Mining co-change information to understand when build changes are necessary. In: *Proceedings of International Conference on Software Maintenance and Evolution (ICSME)*, 2014. DOI: 10.1109/icsme.2014.46.

[73] Xin Xia, David Lo, Shane McIntosh, Emad Shihab, and Ahmed E. Hassan. Cross-project build co-change prediction. In: *Proceedings of International Conference on Software Analysis, Evolution, Reengineering (SANER)*, 2015. DOI: 10.1109/saner.2015.7081841.

[74] Christian Macho, Shane McIntosh, and Martin Pinzger. Predicting build co-changes with source code change and commit categories. In: *Proceedings of International Conference on Software Analysis, Evolution, Reengineering (SANER)*, 2016. DOI: 10.1109/saner.2016.22.

[75] Cor-Paul Bezemer, Shane McIntosh, Bram Adams, Daniel M. German, and Ahmed E. Hassan. An empirical study of unspecified dependencies in make-based build systems. *Empirical Software Engineering (EMSE)* 22(6), 2017, pp. 3117–3148. DOI: 10.1007/s10664-017-9510-8.

[76] Milos Gligoric, Wolfram Schulte, Chandra Prasad, Danny van Velzen, Iman Narasamdya, and Benjamin Livshits. Automated migration of build scripts using dynamic analysis and search-based refactoring. In: *Proceedings of International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*, pp. 599–616. 2014. DOI: 10.1145/2660193.2660239.

[77] Roman Suvorov, Meiyappan Nagappan, Ahmed E. Hassan, Ying Zou, and Bram Adams. An empirical study of build system migrations in practice: case studies on KDE and the linux kernel. In: *Proceedings of International Conference on Software Maintenance (ICSM)*, pp. 160–169. 2012. DOI: 10.1109/icsm.2012.6405267.

[78] Gerald Schermann and Philipp Leitner. Search-based scheduling of experiments in continuous deployment. In: *Proceedings of International Conference on Software Maintenance and Evolution (ICSME)*, pp. 485–495. 2018. DOI: 10.1109/icsme.2018.00059.

[79] Ozan Günalp, Clement Escoffier, and Philippe Lalanda. Rondo: a tool suite for continuous deployment in dynamic environments. In: *Proceedings of International Conference on Services Computing (SCC)*, 2015. DOI: 10.1109/scc.2015.102.

[80] Somit Gupta, Lucy Ulanova, Sumit Bhardwaj, Pavel Dmitriev, Paul Raff, and Aleksander Fabijan. The anatomy of a large-scale experimentation platform. In: *Proceedings of International Conference on Software Architecture (ICSA)*, 2018. DOI: 10.1109/icsa.2018.00009.

[81] Tony Savor, Mitchell Douglas, Michael Gentili, Laurie Williams, Kent Beck, and Michael Stumm. Continuous deployment at Facebook and OANDA. In: *Proceedings of International Conference on Software Engineering Companion*, 2016. DOI: 10.1145/2889160.2889223.

[82] Eero Laukkanen, Juha Itkonen, and Casper Lassenius. Problems, causes and solutions when adopting continuous delivery—a systematic literature review. *Information and Software Technology (IST)* 82, 2017, pp. 55–79. DOI: 10.1016/j.infsof.2016.10.001.

[83] M. Leppänen, S. Mäkinen, M. Pagels, V. P. Eloranta, J. Itkonen, M. V. Mäntylä, and T. Männistö. The highways and country roads to continuous deployment. *IEEE Software* 32(2), 2015, pp. 64–72. DOI: 10.1109/MS.2015.50.

[84] Keheliya Gallaba and Shane McIntosh. Use and misuse of continuous integration features: an empirical study of projects that (mis)use Travis CI. *IEEE Transactions on Software Engineering (TSE)*, 2018. DOI: 10.1109/tse.2018.2838131.

[85] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M. German, and Daniela Damian. The promises and perils of mining GitHub. In: *Proceedings of International Working Conference on Mining Software Repositories (MSR)*, pp. 92–101. 2014. DOI: 10.1145/2597073.2597074.

[86] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E. Hassan, and Kenichi Matsumoto. An empirical comparison of model validation techniques for defect prediction models. *IEEE Transactions on Software Engineering (TSE)* 43(1), 2017, pp. 1–18. DOI: 10.1109/tse.2016.2584050.

[87] Andrew John Scott and M Knott. A cluster analysis method for grouping means in the analysis of variance. *Biometrics* 30(3), 1974, pp. 507–512.

[88] Bram Adams, Kris de Schutter, Herman Tromp, and Wolfgang de Meuter. The evolution of the linux build system. *Electron. Commun. of the ECEASST* 8, 2008.

[89] Shane McIntosh, Bram Adams, and Ahmed E. Hassan. The evolution of Java build systems. *Empirical Software Engineering (EMSE)* 17(4-5), 2012, pp. 578–608. DOI: 10.1007/s10664-011-9169-5.

[90] Matthew B. Miles, A. Michael Huberman, and Johnny Saldaña. *Qualitative data analysis: A methods sourcebook*. Sage, 2013.

[91] Mahdis Zolfagharinia, Bram Adams, and Yann-Gaël Guéhéneuc. Do not trust build results at face value - an empirical study of 30 million CPAN builds. In: *Proceedings of International Conference on Mining Software Repositories (MSR)*, pp. 312–322. 2017. DOI: 10.1109/msr.2017.7.

[92] Audris Mockus. Software support tools and experimental work. In: Victor R. Basili, Dieter Rombach, Kurt Schneider, Barbara Kitchenham, Dietmar Pfahl, and Richard W. Selby (eds.), *Empirical Software Engineering Issues. Critical Assessment and Future Directions: International Workshop, Dagstuhl Castle, Germany, June 26-30, 2006. Revised Papers*, pp. 91–99. 2007. DOI: 10.1007/978-3-540-71301-2_25.

[93] Peter Kampstra. Beanplot: a boxplot alternative for visual comparison of distributions. *Journal of Statistical Software* 28(Code Snippet 1), 2008, pp. 1–9. DOI: 10.18637/jss.v028.c01.

[94] E. Burton Swanson. The dimensions of maintenance. In: *Proceedings of International Conference on Software Engineering (ICSE)*, pp. 492–497. 1976. URL: http://dl.acm.org/citation.cfm?id=800253.807723.

[95] Adriaan Labuschagne, Laura Inozemtseva, and Reid Holmes. Measuring the cost of regression testing in practice: a study of Java projects using continuous integration. In: *Proceedings of Joint Meeting on European Software Engineering Conference and Symposium on Foundations of Software Engineering (ESEC/FSE)*, pp. 821–830. 2017. DOI: 10.1145/3106237.3106288.

[96] Shane McIntosh, Meiyappan Nagappan, Bram Adams, Audris Mockus, and Ahmed E. Hassan. A Large-Scale Empirical Study of the Relationship between Build Technology and Build Maintenance. *Empirical Software Engineering (EMSE)* 20(6), 2015, pp. 1587–1633.

[97] Andre N. Meyer, Laura E. Barton, Gail C. Murphy, Thomas Zimmermann, and Thomas Fritz. The work life of developers: activities, switches and perceived productivity. *IEEE Transactions on Software Engineering (TSE)* 43(12), 2017, pp. 1178–1193. DOI: 10.1109/tse.2017.2656886.

[98] 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*, 1947, pp. 50–60. URL: http://www.jstor.org/stable/2236101.

[99] Norman Cliff. Dominance statistics: ordinal analyses to answer ordinal questions. *Psychological Bulletin* 114(3), 1993, pp. 494–509. DOI: 10.1037/0033-2909.114.3.494.

[100] David Gray Widder, Michael Hilton, Christian Kästner, and Bogdan Vasilescu. I'm leaving you, Travis: a continuous integration breakup story. In: *Proceedings of International Conference on Mining Software Repositories (MSR)*, pp. 165–169. 2018. DOI: 10.1145/3196398.3196422.

[101] Keheliya Gallaba, Yves Junqueira, John Ewart, and Shane Mcintosh. Accelerating continuous integration by caching environments and inferring dependencies. *IEEE Transactions on Software Engineering (TSE)*, 2020. DOI: 10.1109/TSE.2020.3048335.

[102] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. Automated program repair. *Communications of the ACM* 62(12), 2019, pp. 56–65. DOI: 10.1145/3318162.

[103] Xianhao Jin and Francisco Servant. A cost-efficient approach to building in continuous integration. In: *Proceedings of International Conference on Software Engineering (ICSE)*, 2020. DOI: 10.1145/3377811.3380437.

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

[105] Sture Holm. A simple sequentially rejective multiple test procedure. *Scandinavian Journal of Statistics*, 1979. URL: http://www.jstor.org/stable/4615733.

[106] Mohsen Vakilian, Raluca Sauciuc, J. David Morgenthaler, and Vahab Mirrokni. Automated decomposition of build targets. In: *Proceedings of International Conference on Software Engineering (ICSE)*, 2015. DOI: 10.1109/icse.2015.34.

[107] Shane McIntosh, Bram Adams, Meiyappan Nagappan, and Ahmed E. Hassan. Identifying and Understanding Header File Hotspots in C/C++ Build Processes. *Automated Software Engineering* 23(4), 2015, pp. 619–647. DOI: 10.1007/s10515-015-0183-5.

[108] Stuart I. Feldman. Make — a program for maintaining computer programs. *Software: Practice and Experience* 9(4), 1979, pp. 255–265. DOI: 10.1002/spe.4380090402.

[109] Bram Adams, Kris de Schutter, Herman Tromp, and Wolfgang de Meuter. MAKAO (demo). In: *Proceedings of International Conference on Software Maintenance (ICSM)*, pp. 517–518. 2007.

[110] Manuela Züger and Thomas Fritz. Interruptibility of software developers and its prediction using psycho-physiological sensors. In: *Proceedings of International Conference on Human Factors in Computing Systems (CHI)*, pp. 2981–2990. 2015. DOI: 10.1145/2702123.2702593.