Use and Misuse of Continuous Integration Features

An Empirical Study of Projects that (mis)use Travis CI

Keheliya Gallaba, *Student Member, IEEE,* and Shane McIntosh, *Member, IEEE*

Abstract—Continuous Integration (CI) is a popular practice where software systems are automatically compiled and tested as changes appear in the version control system of a project. Like other software artifacts, CI specifications require maintenance effort. Although there are several service providers like TRAVIS CI offering various CI features, it is unclear which features are being (mis)used. In this paper, we present a study of feature use and misuse in 9,312 open source systems that use TRAVIS CI. Analysis of the features that are adopted by projects reveals that explicit deployment code is rare—48.16% of the studied TRAVIS CI specification code is instead associated with configuring *job processing* nodes. 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 in the corpus (9.60%), and achieves a recall of 82.76% in a sample of 100 projects. Furthermore, we propose GRETEL—an anti-pattern removal tool for TRAVIS CI specifications, which can remove 69.60% of the most frequently occurring anti-pattern automatically. Using GRETEL, we have produced 36 accepted pull requests that remove TRAVIS CI anti-patterns automatically.

Index Terms—Continuous integration, Anti-patterns, Mining software repositories

1 INTRODUCTION

CONTINUOUS Integration (CI) is a software development practice in which the latest code changes are regularly downloaded onto dedicated machines to validate that the codebase still compiles, and that unit and integration tests still pass. 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 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. Since the build process that is being invoked tends to evolve [1], [22], 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, 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. CI configuration can also be *misused*, e.g., when unsupported or deprecated commands are used.

In this paper, we set out to study how CI features are being used and misused. First, 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 [15], 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

• *K. Gallaba and S. McIntosh are with the Department of Electrical and Computer Engineering, McGill University, Canada.*
E-mail: keheliya.gallaba@mail.mcgill.ca, shane.mcintosh@mcgill.ca

Manuscript received date; revised date.

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

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* [16]—the process of automatically releasing code that integrates cleanly.
- **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.

To study misuse, we define four anti-patterns: (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:

- **RQ4** *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:

- **RQ5** *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.
- **RQ6** *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 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. Moreover, 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.

Paper organization. The remainder of the paper is organized as follows. Section 2 describes the modern CI process. Section 3 outlines the design of our study of CI feature usage, while Section 4 presents the results. Sections 5 and 6 outline the motivation for and design of our study of CI misuse, respectively, while Section 7 presents the results. Section 8 discusses the broader implications of our results. Section 9 discloses the threats to the validity of our study. Section 10 situates this paper with respect to the related work. Finally, Section 11 draws conclusions.

2 MODERN CI PROCESS

The main goal of CI is automating the integration of software as soon as it is developed so that it can be released rapidly and reliably [11]. Figure 1 provides an overview of the cycle. We describe each step below.

- **Build-triggering events**: In projects that adopt CI, the cycle begins with a build-triggering event. These events can occur in the development, review, or integration stages. 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 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.
- **Build job processing service**: Build jobs in the pending queue will be allocated to build job processing nodes for processing. 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) [16], 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 is also popular nowadays (e.g., Slack, web dashboards).

Operating and maintaining CI infrastructure is a burden for modern software organizations. As organizations grow,

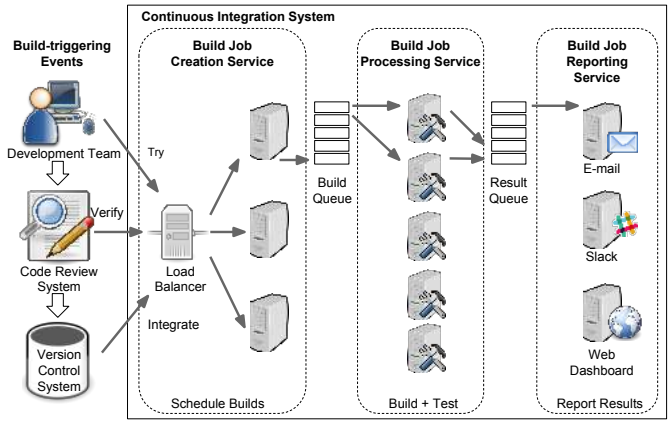


Fig. 1: Main components of a continuous integration system: Build job creation, processing, and reporting.

their CI 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 load.

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

2.1 Configuring Travis CI

TRAVIS CI 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 stored in a `.travis.yml` file, which appears in the root directory of a GITHUB repository. The `.travis.yml` file can also specify programming language runtimes, and other environment configuration settings that are needed to execute build jobs.

Figure 2 shows that `.travis.yml` files consist of node configuration and build process configuration sections. We describe each section below.

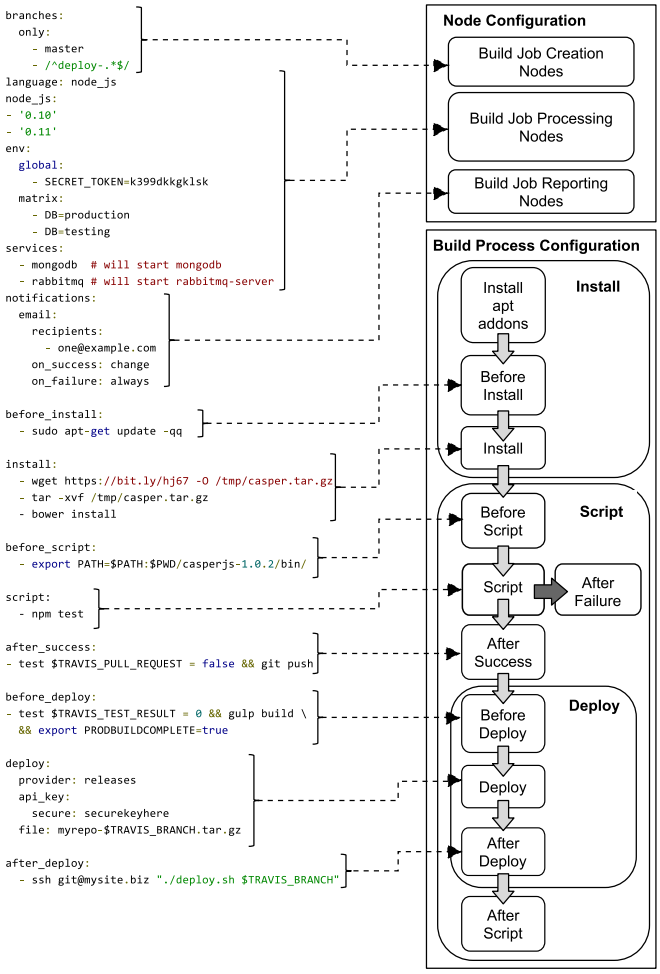


Fig. 2: A `.travis.yml` configuration file (left) and how it maps to the TRAVIS CI life cycle (right). The file consists of (1) *node configuration*, which specifies how the nodes for build job creation, processing, and reporting are configured; and (2) *build process configuration*, which specifies the commands that are executed in the `install`, `script`, and `deploy` phases and their sub-phases.

2.1.1 Node Configuration

This section specifies how CI 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 branches where commits should create 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 installed), 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.
- **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`).

2.1.2 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`,

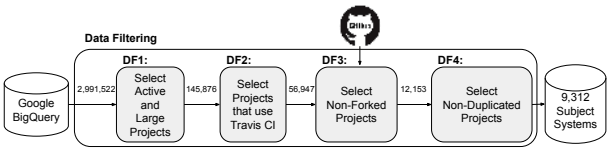


Fig. 3: An overview of our data filtering approach.

`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 [16], where regression-free changes are released to system users.

2.2 Research Questions

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 and GRETEL) will be able to tailor their solutions to fit the needs of target users.

Hilton et al. [15] 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.
- **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 section 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 churn is dispersed among the sections.

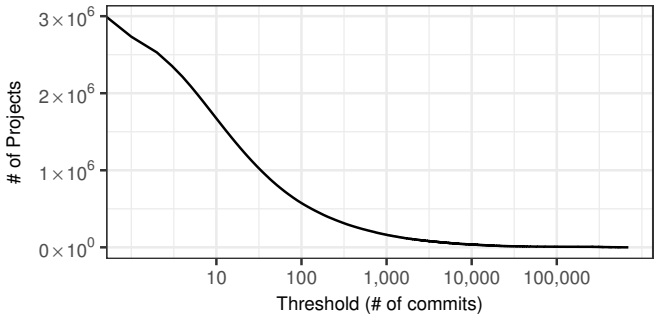


Fig. 4: Threshold plot for commit activity.

3 CI USAGE STUDY DESIGN

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

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.

3.2 Data Filtering

While GITHUB is a large corpus, it is known to contain projects that have not yet reached maturity [18]. To prevent the bulk of immature projects from impacting our conclusions, we first apply a set of filters to our GITHUB data. Figure 3 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 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 5 again plots threshold values against the number of surviving systems. Selecting a threshold of 500 files further reduces the corpus to 145,876 projects.

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

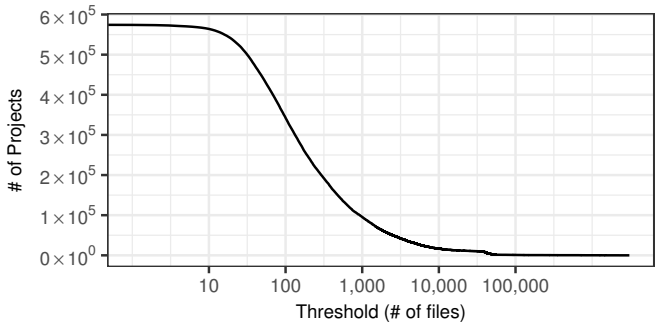


Fig. 5: Threshold plot for project size.

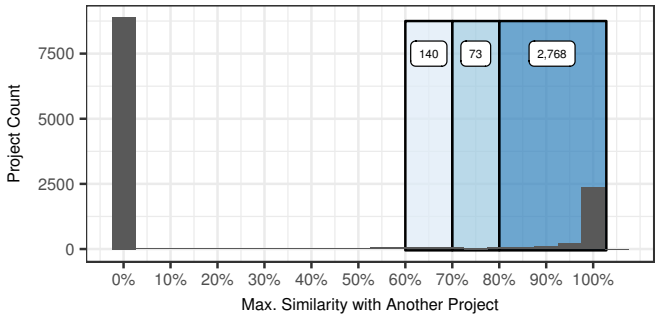


Fig. 6: A histogram of the maximum commit similarity among the candidate repositories.

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 8.4.

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 candidate 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 6 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.

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

TABLE 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**

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 1 shows that our corpus contains a broad variety of subject systems, including games, and web and mobile apps.

4 RESULTS OF CI USAGE STUDY

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.

TABLE 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

Results. Table 2 shows the ten most popular languages in our corpus of studied projects. Hilton et al. [15] 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 [15] 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 requests). ANDROID does not appear in the list by GITHUB because it is 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 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 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.

Since some languages may require more files than others, we repeat our analysis with four file count threshold values (DF1). Figure 7 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.

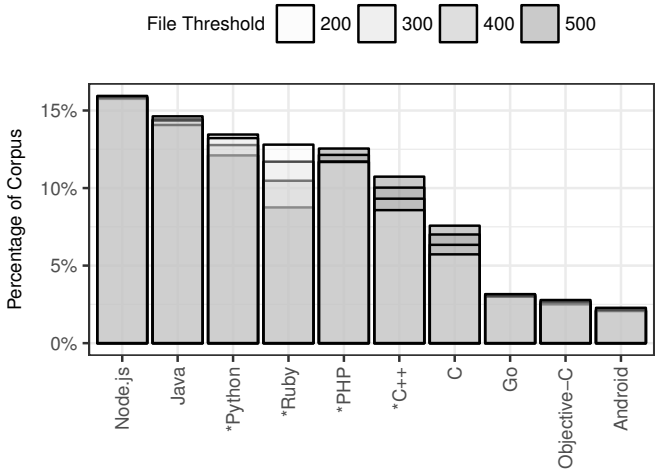


Fig. 7: The percentage of the corpus that uses the ten most popular languages. Asterisks (*) denote languages that change ranks when the file count threshold changes (DF1).

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, build status notification, or other. Table 3 shows our mapping of `.travis.yml` tags to these sub-categories.

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 [31]—an enhancement to the Scott-Knott test [27], which also considers the effect size when clustering CI sections into statistically distinct ranks.

Results. Table 4 shows the popularity of the sections, as well as their overall length and proportion within the corpus.

***Observation 3**: For CI node configuration, sections that are related to* job processing *nodes appear in the most projects.* Table 4 shows that 8,852 (95.06%) of the studied `.travis.yml`

10. https://octoverse.github.com/
11. http://stackoverflow.com/insights/survey/2017
12. https://www.theregister.co.uk/2016/03/23/npm_left_pad_chaos/

TABLE 3: The identified build process 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`

TABLE 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

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 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 8a 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 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 8b shows the distribution of commands after removing zero-length sections. The difference in the `deploy` phases in Figure 8a (with zeros) and Figure 8b (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.*

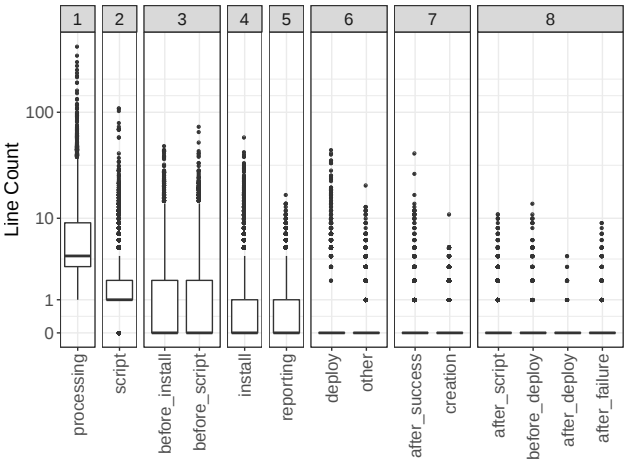
(RQ3) Which sections in the CI specifications induce the most churn?

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 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.

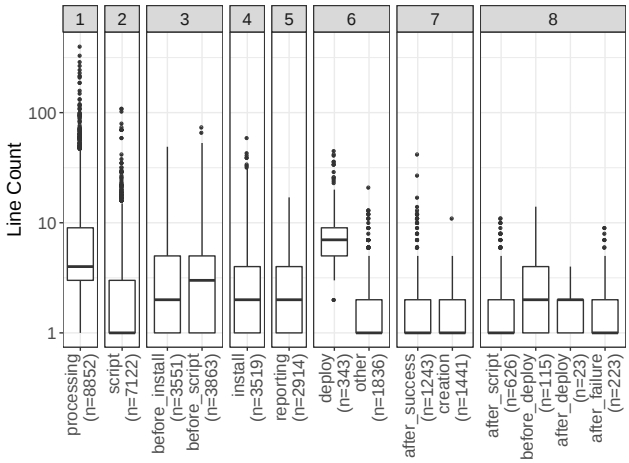
Results. Figure 9a shows the churn of each phase over time. The `.travis.yml` files in the subject systems are modified 18.06 times on average in their lifetime, with a maximum of 366 changes in the *lolli42/TYPO3.CMS-Catharsis* project;[16] however, the churn in each phase is very low. In more than 75% of the studied projects, configuration sections are modified fewer than 10 times. These results complement the work of Hilton et al. [15], 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/FrameworkBench-*

13. https://github.com/jruby/activerecord-jdbc-adapter
14. https://github.com/joshuarowley42/BigBox-Pro-Marlin
15. https://github.com/oden-lang/oden
16. https://github.com/lolli42/TYPO3.CMS-Catharsis

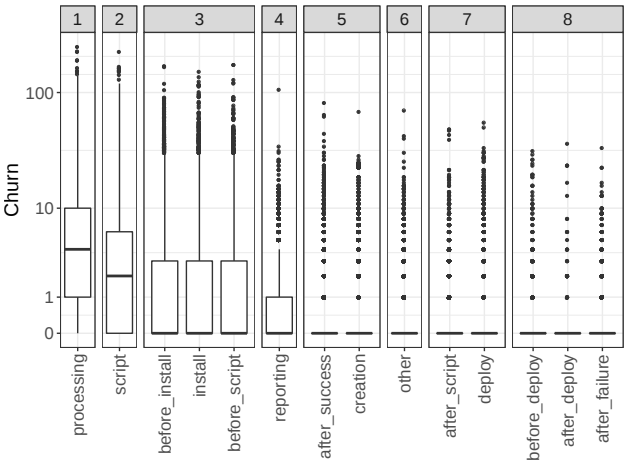


(a) The distribution for all projects.

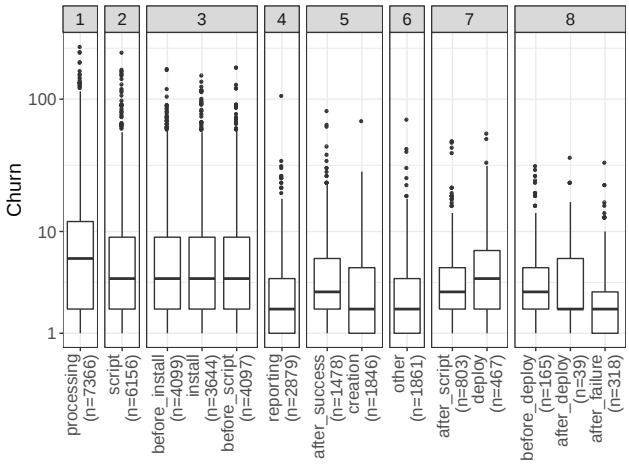


(b) The distribution after removing zero-length sections.

Fig. 8: Line counts in each section of the `.travis.yml` file.



(a) The distribution for all projects.



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

Fig. 9: The churn of each section in the `.travis.yml` file.

marks,[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 requires job processing nodes to be configured differently. This complements our earlier observations that most of the effort in configuring CI is spent on the processing node configuration.

Hilton et al. [15] 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 granularities of our analyses. For example, in our analysis, we study distributions 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 9a 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 9b. 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.*

5 Anti-patterns in CI 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

17. https://github.com/TechEmpower/FrameworkBenchmarks

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,[18] an online service and an open source tool that scans `.travis.yml` files for mistakes (e.g., YAML formatting issues, missing mandatory fields). If the issues are fixed, TRAVISLINT can prevent configuration errors from breaking project builds.

5.1 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:

- **RQ4** *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".
- **RQ5** *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.
- **RQ6** *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.

6 CI MISUSE 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[19] 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 [17], [29], we first read the rules implemented by TRAVISLINT,[18] formal TRAVIS CI documentation,[20] informal documentation from the 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 related best practices and deduce corresponding anti-patterns (i.e., cases where best practice are being violated).

18. https://docs.travis-ci.com/user/travis-lint
19. https://pypi.python.org/pypi/ruamel.yaml
20. https://docs.travis-ci.com/

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,[21] 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.[22] 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.

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.[23,24]

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

21. https://sandstorm.io/install
22. https://www.idontplaydarts.com/2016/04/detecting-curl-pipe-bash-server-side/
23. https://annevankesteren.nl/2017/01/secure-secure-shell
24. https://docs.travis-ci.com/user/ssh-known-hosts/#Security-Implications

(`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.

Detection. In Section 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.

We consider instances of well-bounded commands that we find in other phases to be instances of this anti-pattern. Table 5 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 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.

7 CI MISUSE STUDY RESULTS

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

(RQ4) How prevalent are anti-patterns in CI specifications?

Observation 9*: 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,[25] which provides the Russian version of the AngularJS documentation, has three types of anti-patterns (the only missing anti-pattern is #3).

Observation 10*: In a sample of 100 projects,* HANSEL *achieves a recall of 82.76%.* To estimate the recall of HANSEL, we manually identify anti-patterns in a randomly selected sample of 100 `.travis.yml` files from our corpus. We apply HANSEL to the same sample, and compute the $recall = \frac{\#\ anti\text{-}patterns\ found\ by\ \text{HANSEL}}{\#\ anti\text{-}patterns\ found\ manually}$.

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).

25. https://github.com/AngularjsRUS/angular-doc

TABLE 5: 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`

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). This can easily be remedied by adding the missing binding.

Observation 11*: 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.[26] In fact, the METEOR documentation instructs users to install the framework using this method (`curl https://install.meteor.com|/bin/sh`).[27]

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[22] or how networking interruptions during the download command may lead to partial execution of the installation script.[28] On the other hand, members of the SANDSTORM project defend script redirection for cases where script downloads are served strictly over HTTPS.[29] 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-type Linux distributions, respectively). Moreover, discussion threads on HACKERNEWS[30] 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 12*: 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.

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 13*: 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 6).

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

26. https://www.meteor.com
27. https://www.meteor.com/install
28. https://www.seancassidy.me/dont-pipe-to-your-shell.html
29. https://sandstorm.io/news/2015-09-24-is-curl-bash-insecure-pgp-verified-install
30. https://news.ycombinator.com/item?id=12766049

TABLE 6: 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 7: Commands that appear in unrelated phases.

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

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 14*: The most common variant of anti-pattern 4 is using `install` phase commands in the `script` phase.* Table 7 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 developers 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.

```
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
```

Fig. 10: 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.

Observation 15*: 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) 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, 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.*

(RQ5) 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 ran-

domly select a subset of candidates for removal and manually classify them until we achieve *saturation* [24], 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.[18] We then apply GRETEL to the valid candidates in order to remove 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 16*: 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 10 shows an example where a state-altering command impacts the removal of an anti-pattern, taken from *lamkeewei/battleships*,[31] a tool for building 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 17*: 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 18*: In 29 instances, limitations in the* ruamel.yaml *framework*[19] *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.

31. https://github.com/lamkeewei/battleships

Seven (2.80%) of the remaining projects use the YARN package manager[32] 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.*

(RQ6) 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 19*: 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. In another two pull requests, 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 that uses `install`-related commands in the `before_script` phase.[33] 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.*

32. https://yarnpkg.com/en/

33. https://docs.travis-ci.com/user/languages/javascript-with-nodejs/#Using-Gulp

8 Further Insights into CI Misuse

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

8.1 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`.[34] FACEBOOK, the organization that maintains `xctool`, have deprecated it as of 2016.[35] 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[36] 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 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,[37] 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.

8.2 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[38] 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.

TABLE 8: 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

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 8 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.

8.3 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[39] uses the `before_install`, `before_script`, `script`, 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%).

8.4 Applicability to Other CI 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

34. https://github.com/facebook/xctool
35. https://github.com/facebook/xctool/blob/master/README.md#features
36. https://blog.travis-ci.com/after_script_behaviour_changes/
37. https://circleci.com/docs/2.0/migrating-from-1-2/
38. https://github.com/huginn/huginn
39. https://github.com/aescobarr/natusfera

this paper may also apply to these services. For example, CIRCLECI uses a `config.yml` file[40] 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.

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.[41]

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, we can map commands to CIRCLECI phases where they should appear.

9 THREATS TO VALIDITY

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

9.1 Internal Validity

The list of anti-patterns that we present in the paper is not exhaustive. However, to the best of our knowledge, this paper 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 [9], 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. 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.

9.2 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 [15]. Therefore, our findings are applicable to a large proportion of open source projects. Moreover, given the similarities among the popular CI services (see Section 8.4), our observations are likely applicable to some degree. Nonetheless, replication studies using other CI services may yield further insight.

40. https://circleci.com/docs/2.0/

41. https://discuss.circleci.com/t/add-known-hosts-on-startup-via-config-yml-configuration/12022

9.3 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.

CI use and misuse 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 6 and 7), where the impact of selecting different thresholds is shown to be minimal.

10 RELATED WORK

In this section, we situate our work with respect to the literature on continuous integration and configuration smells.

10.1 Continuous Integration

As a relatively new practice in software development, CI has only just begun to attract the attention of software engineering researchers [2].

Recent work has characterized CI practices and outcomes along different dimensions. Meyer [23] 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 [30] also provided a systematic overview of CI practices and their differences from a technical perspective. Vasilescu et al. [32] 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. For example, Hilton et al. [15] 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. In a recent qualitative study, Hilton et al. [14] also found that, when adopting CI, developers face trade-offs between speed and certainty, accessibility and security, and configurability

and ease of use. Laukkanen et al. [19] 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. [20] found that domain-imposed restrictions, resistance to change, customer needs, and developers' skill and confidence are adoption obstacles for continuous deployment.

Other works focus on improving specific stages of the CI process. Beller et al. [4] 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. [4] 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. [12] 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 [6], [10] 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 [7].

Our goal in this paper 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.

10.2 Software Configuration Smells

To the best of our knowledge, this paper 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. [5] published a catalog of anti-patterns and patterns for software configuration management. Shambaugh et al. [28] proposed REHEARSAL, a verification tool for PUPPET configurations. Sharma et al. [29] 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. [9] surveyed developers and used the findings to develop a PUPPET code quality analysis tool. Rahman and Williams [26] 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. [17] proposed a static analysis tool for detecting errors in configuration files of ANDROID apps. In an exploratory empirical study, Cito et al. [8] assessed the quality of DOCKER configuration files on GITHUB, observing that they violate 3.1 linter rules on average.

Another related context is architectural or design smells. Marinescu [21] has defined detection strategies for capturing important flaws of object-oriented design that were reported in the literature. Garcia et al. [13] 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. [25] 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 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. [28]. Alicherry and Keromytis [3] 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. 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. [29] and the *Silent Failure* problem proposed by Shambaugh et al. [28]. Finally, our anti-pattern 4 (commands unrelated to the phase) is similar to Sharma et al.'s *Misplaced Attribute* and *Multifaceted Abstraction* configuration smells [29]. 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.

11 CONCLUSIONS

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. 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 use and misuse 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).

- 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 9–15).
- 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 16–18).
- Of the 49 pull requests for instances that are removed automatically and to which developers responded, 36 (73.47%) have been accepted (Observation 19).

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. Moreover, 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.

REFERENCES

[1] B. Adams, K. de Schutter, H. Tromp, and W. de Meuter. The evolution of the linux build system. *Electronic Communications of the ECEASST*, 8, 2 2008.

[2] B. Adams and S. McIntosh. Modern release engineering in a nutshell: Why researchers should care. In *Proceedings of the International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 5, pages 78–90, 2016.

[3] M. Alicherry and A. D. Keromytis. DoubleCheck: Multi-path verification against man-in-the-middle attacks. In *Proceedings of the IEEE Symposium on Computers and Communications (ISCC)*, jul 2009.

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

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

[6] J. Campos, A. Arcuri, G. Fraser, and R. Abreu. Continuous test generation: Enhancing continuous integration with automated test generation. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 55–66, 2014.

[7] F. Cannizzo, R. Clutton, and R. Ramesh. Pushing the boundaries of testing and continuous integration. In *Proceedings of the Agile 2008 Conference*, pages 501–505, 2008.

[8] J. Cito, G. Schermann, J. E. Wittern, P. Leitner, S. Zumberi, and H. C. Gall. An empirical analysis of the Docker container ecosystem on GitHub. In *Proceedings of the International Conference on Mining Software Repositories (MSR)*, pages 323–333, 2017.

[9] E. V. der Bent, J. Hage, J. Visser, and G. Gousios. How good is your puppet? an empirically defined and validated quality model for puppet. In *Proceedings of the International Conference on Software Analysis, Evolution and Reengineering (SANER)*, page To Appear, 2018.

[10] S. Dösinger, R. Mordinyi, and S. Biffl. Communicating continuous integration servers for increasing effectiveness of automated testing. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 374–377, 2012.

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

[12] S. Elbaum, G. Rothermel, and J. Penix. Techniques for improving regression testing in continuous integration development environments. In *Proceedings of the SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pages 235–245, 2014.

[13] J. Garcia, D. Popescu, G. Edwards, and N. Medvidovic. Toward a catalogue of architectural bad smells. In *Proceedings of the International Conference on the Quality of Software Architectures: Architectures for Adaptive Software Systems*, pages 146–162, 2009.

[14] M. Hilton, N. Nelson, T. Tunnell, D. Marinov, and D. Dig. Trade-offs in continuous integration: assurance, security, and flexibility. In *Proceedings of Joint Meeting of the European Software Engineering Conference and the International Symposium on the Foundations of Software Engineering*, pages 197–207, 2017.

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

[16] J. Humble and D. Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Pearson Education, 2010.

[17] A. K. Jha, S. Lee, and W. J. Lee. Developer mistakes in writing Android manifests: An empirical study of configuration errors. In *Proceedings of the International Conference on Mining Software Repositories (MSR)*, pages 25–36, 2017.

[18] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian. The promises and perils of mining GitHub. In *Proceedings of the International Working Conference on Mining Software Repositories (MSR)*, pages 92–101, 2014.

[19] E. Laukkanen, J. Itkonen, and C. Lassenius. Problems, causes and solutions when adopting continuous delivery—a systematic literature review. *Information and Software Technology*, 82:55–79, 2017.

[20] 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):64–72, 2015.

[21] R. Marinescu. Detection strategies: Metrics-based rules for detecting design flaws. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 350–359, 2004.

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

[23] M. Meyer. Continuous integration and its tools. *IEEE Software*, 31(3):14–16, 2014.

[24] M. B. Miles, A. M. Huberman, and J. Saldaña. *Qualitative data analysis: A methods sourcebook*. Sage, 2013.

[25] N. Moha, Y.-G. Guéhéneuc, L. Duchien, and A.-F. L. Meur. Decor: A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering (TSE)*, 36(1):20–36, 2010.

[26] A. Rahman and L. Williams. Characterizing defective configuration scripts used for continuous deployment. In *Proceedings of International Conference on Software Testing, Validations, and Verification (ICST)*, page To Appear, 2018.

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

[28] R. Shambaugh, A. Weiss, and A. Guha. Rehearsal: A configuration verification tool for puppet. In *Proceedings of the International Conference on Programming Language Design and Implementation (PLDI)*, pages 416–430, 2016.

[29] T. Sharma, M. Fragkoulis, and D. Spinellis. Does your configuration code smell? In *Proceedings of the International Conference on Mining Software Repositories (MSR)*, pages 189–200, 2016.

[30] D. Ståhl and J. Bosch. Modeling continuous integration practice differences in industry software development. *Journal of Systems and Software*, 87:48–59, 2014.

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

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

Keheliya Gallaba is a PhD student at McGill University, Canada. His PhD thesis aims to improve the robustness and efficiency of continuous integration and continuous deployment tools. He received his BSc degree in Computer Science and Engineering from University of Moratuwa, Sri Lanka and his MASc degree in Electrical and Computer Engineering from University of British Columbia, Canada. More about Keheliya and his work is available online at http://keheliya.github.io/.

Shane McIntosh is an assistant professor in the Department of Electrical and Computer Engineering at McGill University, where he leads the Software Repository Excavation and Build Engineering Labs (Software REBELs). He received his Bachelor's degree in Applied Computing from the University of Guelph and his Master's and PhD in Computer Science from Queen's University, for which he was awarded the Governor General of Canada's Academic Gold Medal. In his research, Shane uses empirical software engineering techniques to study software build systems, release engineering, and software quality. More about Shane and his work is available online at http://rebels.ece.mcgill.ca/.