Which Commits Can Be CI Skipped?

Rabe Abdalkareem, Suhaib Mujahid, Emad Shihab, *Senior Member, IEEE,* Juergen Rilling

Abstract—Continuous Integration (CI) frameworks such as Travis CI, automatically build and run tests whenever a new commit is submitted/pushed. Although there are many advantages in using CI, e.g., speeding up the release cycle and automating the test execution process, it has been noted that the CI process can take a very long time to complete. One of the possible reasons for such delays is the fact that some commits (e.g., changes to readme files) unnecessarily kick off the CI process.
Therefore, the goal of this paper is to automate the process of determining which commits can be CI skipped. We start by examining the commits of 58 Java projects and identify commits that were explicitly CI skipped by developers. Based on the manual investigation of 1,813 explicitly CI skipped commits, we first devise an initial model of a CI skipped commit and use this model to propose a rule-based technique that automatically identifies commits that should be CI skipped. To evaluate the rule-based technique, we perform a study on unseen datasets extracted from ten projects and show that the devised rule-based technique is able to detect and label CI skip commits, achieving Areas Under the Curve (AUC) values between 0.56 and 0.98 (average of 0.73). Additionally, we show that, on average, our technique can reduce the number of commits that need to trigger the CI process by 18.16%. We also qualitatively triangulated our analysis on the importance of skipping the CI process through a survey with 40 developers. The survey results showed that 75% of the surveyed developers consider it to be nice, important or very important to have a technique that automatically flags CI skip commits. To operationalize our technique, we develop a publicly available prototype tool, called CI-SKIPPER, that can be integrated with any git repository and automatically mark commits that can be CI skipped.

Index Terms—Continuous Integration, Travis CI, Build Status, Mining Software Repository.

1 INTRODUCTION

CONTINUOUS integration (CI) is becoming increasingly popular in modern software projects. CI platforms automate the process of building and testing these projects. Previous research showed that CI, amongst other things, increases developers' productivity and helps improve software quality [38]. Due to their many advantages, Hilton *et al.* showed that CI is used by both, the open source community and in industrial software projects [18], [19] and that as much as 40% of 34,544 of analyzed popular GitHub projects use CI [19].

Despite CI's many benefits and wide popularity, it also has several drawbacks. CI's process can take a very long time to complete [26], especially for large projects [8]. This can be particularly problematic for developers who need the CI process to complete after each commit. This long waiting time is mainly due to the fact that the CI process needs to automatically clone the source code into a clean virtual machine, set up the required environment, initiate the build, run the tests and output the result of the build to the developers after each commit. This waiting time can affect both, the speed of software development and the productivity of the developers (Duvall *et al.* [11], p. 87).

Most of the previous work on CI focused on the study of its usage and benefits (e.g,. [22], [38]). Other work examined the reason for failing builds [31], and even tried to predict the results of the build result [17]. However, very few studies tried to improve the efficiency of the CI process. We believe that doing so can reap benefits, especially for large projects that use CI. Since the CI process is triggered by commits, we believe that trying to reduce the number of commits that kick off the CI process will have the biggest impact (though it is not the only way to do so). Our main argument is that *not every commit needs to trigger the CI process*. For instance, developers may modify a project's documentations, which causes the CI process to be triggered. Since such a change does not affect the source code, the result of the build will not change and kicking off the CI process is just a waste of resources. Furthermore, in a discussion channel on Travis CI, many developers argue that the CI process should not be run on every commit, and Travis CI developers are asked to provide an advanced mechanism to automatically CI skip specific commits[1]. Even though, Travis CI actually has a built in functionality that allows developers to skip the CI process for a specific commit, the challenge of which commit to CI skip remains. As we will show later, developers often do not leverage this existing CI skip feature, which indicates that they a) either are unaware of this feature or b) do not know when a commit can be CI skipped.

Therefore, the main *goal* of our work is to automatically detect and label commits that can be CI skipped. We begin by studying 1,813 CI skip commits belong to projects from the TravisTorrent dataset [2], where developers explicitly skip the build when using Travis CI to understand the reasons why developers skip build commits. We found that developers skip the CI process for eight main reasons, of which five can be automated; changes that touch only doc-

- *R. Abdalkareem, S. Mujahid and E. Shihab with the Data-driven Analysis of Software (DAS) Lab at the Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada. E-mail: rab_abdu,s_mujahi, eshihab@encs.concordia.ca*
- *Juergen Rilling is with the Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada. E-mail: juergen.rilling@concordia.ca*

Manuscript received May 29, 2018; revised August 26, 2018.

[1]https://github.com/travis-ci/travis-ci/issues/6301

umentation files, changes that touch only source code comments, changes that modify the formatting of source code, changes that touch meta files, and changes that only prepare the code for release. Based on the automatable reasons, we propose a rule-based technique, which automatically detects and labels commits that can be CI skipped.

To examine the effectiveness and potential effort savings of our proposed technique, we perform an empirical study using 392 open source Java projects. Our study examines two research questions ***RQ1:*** How effective is our rule-based technique in detecting CI skip commits? and ***RQ2:*** How much effort can be saved by marking CI skip commits using our rule-based technique? Our findings show that our rule-based technique can detect and label CI commits with an AUC between 0.56 and 0.98 (average of 0.73). Although these may seem like modest performance numbers, they are quite favourable given the unbalanced nature of the data (i.e., only a small portion of the commits can be CI skipped). Moreover, we find that applying our technique can, on average, CI skip 18.16% of a project's commits, amounting to an average savings of 917 minutes per project.

Moreover, to better understand the importance of the CI skip technique from developers perspective, we conducted a survey with 40 developers. Our survey results showed that 75% of the developers believe it would be nice, important or very important to have a technique that automatically indicates CI skip commits. Finally, we built a prototype tool that implements our rule-based technique and make it publicly available so that developers and the research community can begin to leverage the benefits of this research immediately.

Our work makes the following contributions:

- We first qualitatively examine commits that can be CI skipped. We manually examine more than 1,800 commits to determine the reasons why developers CI skip commits.
- We propose a rule-based technique that can be used to automatically detect and label CI skip commits. Our results show that our technique is effective and can provide effort savings for software projects.
- We perform a survey with 40 open source developers to gain an in-depth understanding about the importance of having a technique to CI skip commits. Developers indicated that they CI skip a commit when they perceive no change in the build results, saving development time and computational resources. At the same time, 75% of the surveyed developers indicate that having an automatic techniques to CI skip commits would be favourable.
- We build a prototype tool, called CI-SKIPPER, that implements our technique and is publicly available for the community to use [2].

Paper organization. The rest of the paper is organized as follows. Section 2 provides background on the CI process, in particular Travis CI. We detail our data collection and the dataset used in our study in Section 3. We describe our approach and the reasons that developers CI skip commits in Section 4. We present our case study results, detailing the effectiveness and effort savings of our technique in Section 5. In Section 6, we present the developers survey about CI skip commits. The shortcomings of our technique and the use of source code analysis are discussed in Section 7. Section 8 presents our prototype tool, CI-SKIPPER. The work related to our study is discussed in Section 9 and the threats to validity in Section 10. Section 11 concludes the paper.

2 BACKGROUND AND TERMINOLOGY

Since the main goal of our study is to detect commits that can be CI skipped, it is important first to provide some background on CI and Travis CI in particular. Travis CI is an online continuous integration service. When a commit is pushed to any branch or a pull request is made to a Git repository, Travis CI starts the continuous integration process. The pushed commits or pull requests can trigger a build that is often referred to as a *build commit*. A build commit can be triggered on a single commit or a group of commits, known as a *set of changes*. In general, the build commit is supposed to build the project and run any specified tests, which should pass.

Given that the CI process requires resources, developers may decide that there is no need to build the project for a commit (for various reasons). A *skipped commit*, is a commit in which a developer explicitly and intentionally requests the CI process to be bypassed. To skip the CI process in Travis CI, a developer adds the term `[skip ci]` or `[ci skip]` in the commit message. It is important to note here that although Travis CI will provide the functionality for a commit or pull request to be CI skipped, the developer needs to explicitly add the aforementioned terms in the commit message.

Once the CI process is triggered with a *build commit*, the repository is first cloned into a clean virtual machine. Then, Travis CI starts the installation phases by installing all the required dependencies for the project and builds it and runs associated test cases. Travis CI can be configured to run multiple jobs (i.,e. n-jobs). A *Job* corresponds to a configuration of the building step (i.e., the SDK version or the DBMS), which helps to reduce the build process time and to improve the virtual machine utilization.[3].

Once the CI process is complete, Travis CI reports its results, which can be one of four: *passed:* the project is successfully built and its tests pass; *failed:* the project fails to build or some of its tests did not pass; *errored:* which can happen for different reasons, but generally means that an error occurred in one or more of the CI phases (e.g., the installation did not complete or a configuration of the build system is missing), and/or *canceled* which means the CI process was canceled, most likely by the developer or the release engineer [1].

3 DATA COLLECTION

The *goal* of our study is to determine the commits that can be CI skipped. Since to the best of our knowledge, no other work examined skipped commits, we collected data of projects that skip some of their commits, which we use later to derive rules that can be used to skip commits. In this section, we detail our data collection and processing steps.

[2]http://das.encs.concordia.ca/publications/which-commits-can-be-ci-skipped/

[3]https://docs.travis-ci.com/user/speeding-up-the-build/

TABLE 1
Percentage of Build Results in All the Java Projects in the TravisTorrent Dataset.

Build Result	Min.	Median($\tilde{x}$)	Mean(μ)	Max.
Passed	0.00%	84.13%	76.37%	100%
Failed	0.00%	7.89%	13.86%	100%
Errored	0.00%	4.87%	9.58%	90.09%
Canceled	0.00%	0.00%	0.19%	5.45%

3.1 The TravisTorrent Dataset

The main data source for our study is the TravisTorrent dataset [2]. TravisTorrent is a publicly available data set that synthesizes data from Travis CI[4] and their corresponding GitHub repositories. We obtained the TravisTorrent data dump (released December 06, 2016) in SQL format. The dataset combines meta-data from three sources, the git version control system, the GitHub website, and Travis CI services [1].

Since we are interested in non-toy projects, and similar to prior research [21], we use a number of criteria to make sure we study real projects. We used the TravisTorrent dataset which identifies projects with at least 50 Travis CI builds and a minimum of 10 watchers on GitHub [2]. Based on this filtering process, TravisTorrent dataset contains 1,283 open source projects, made up of 886 ruby projects, 393 Java projects, and 4 JavaScript projects. In this paper, we focused on the study of the 393 projects written in Java. We chose to focus on projects written in Java, since 1) we manually examined each project and wanted a dataset that is sufficiently large, but at the same time manageable in size to analyze manually, 2) Java is a well-understood language that the authors have expertise in, hence, giving us confidence in the manual analysis, and 3) Java is one of the most popular programming languages on GitHub [38]. That said, it is important to note that our study is not language dependent and our approach & technique can be applied on projects written in any programming language.For our study, we cloned all 393 open source Java projects provided in the TravisTorrent dataset and identified the date when Travis CI was introduced to each project. We did so by determining the commit date on which the `.travis.yml`[5] file was first added. We could not determine the date for one project since its history was modified (i.e., the developers of the project rebased many of the commits). We therefore excluded this project from our dataset, leaving us with a remaining 392 projects. We analyze the commit history of all the branches for each project and extracted various metrics, which include, 1) the type of change performed in each commit (i.e., does the commit modify source code, modify the formatting of the source code, or modify source code comments); 2) the type of file(s) modified in each commit (provided by their file extensions); and 3) identified the commits that contain the keyword `[ci skip]` or its variation `[skip ci]` in its commit message.

[4] https://travis-ci.org/

[5] The `.travis.yml` file is the configuration file used to configure Travis CI in a project.

TABLE 2
The Selected Ten Open Source Java Projects used as a Testing Dataset.

Project	#Commits§	%Skipped Commits
TracEE Context-Log	216	29.63
SAX	372	23.66
Trane.io Future	247	18.62
Solr-iso639-filter	408	41.42
jMotif-GI	345	12.17
GrammarViz	417	13.67
Parallec	129	56.59
CandyBar	242	69.01
SteVe	298	19.46
Mechanical Tsar	388	34.54
Average	**306.20**	**31.88**
Median	**321.50**	**26.65**

§Number of commits after the introduction of Travis CI service to the project.

3.2 Aggregating the Travis CI Results

The TravisTorrent dataset organizes the build results at the *job* level. Every job is associated with a build commit, a set of changes and is associated with other meta data (e.g,. build states, number of test runs). In total, the dataset contains 456,793 jobs that come from 243,811 build commits. Each project has on average 620.4 builds (median = 296). To come up with one result for a build we aggregate the results of all jobs related to a build and provide one status using the build-id in the TravisTorrent dataset. Since several jobs may belong to the same build commit that can have different statuses, we abstracted the job data to the build commit level in order to avoid any ambiguity as to whether the build commit passed or failed. To do so, we looked at the status of every job related to a build commit and if any of them failed during the build, we considered the whole build commit as failed. Also, the same build commit may trigger the build on Travis CI more than one time and could result in different statuses. We found 26,965 duplicated builds with the same build-id and we eliminated these build commits.

Table 1 shows the summary statistics for each of the different build outcomes of all the Java projects in our dataset. We observe that the majority of the builds pass (median 84.13%), and some fail, error, and/or are canceled (medians 7.89%, 4.87%, and 0.0%, respectively)[6].

Finally, since the goal of this study is to examine the commits that can be skipped, we only focus on builds that have a pass or fail status. Thus, we eliminate all builds commits that have a status of *Error* or *Cancel* from our analysis, since we can not determine the actual reason for the build results, and in such cases it is not clear how and if the commit is impacted. In total, we studied 193,833 out of 243,811 build commits from the 392 different projects.

3.3 Test Dataset

To determine how effective the devised technique is in detecting CI skip commits, we need to have a labeled testing dataset that we can apply the devised technique on. We have two main criteria when building the test dataset: first we need a dataset that is different than the dataset used to learn

[6] There is only one project that all its CI commits fail which is the sdywcd/jshoper3x project.

TABLE 3
Summary of the Number of Commits After Introducing Travis CI, the Number and Percentage of Skip Commit for all Studied Java Projects, and for only Projects Using CI Skip.

Measurement	All the Projects				Projects Using CI Skip			
	Min.	Median($\tilde{x}$)	Mean(μ)	Max.	Min.	Median($\tilde{x}$)	Mean(μ)	Max.
#Commits§	38	769.50	1,556	32,370	168	1,276	2,426	32,370
#Skip Commits	0	0	4.62	515	1	6	31.26	515
%Skip Commits	0.00%	0.00%	0.45%	33.64%	0.01%	0.48%	3.06%	33.64%
#Developers	1	33	49.18	612	2	40	67.60	341
Time-frame‡	5	865	850	1,796	16	904.5	897.5	1,600

§Number of commits after the introduction of Travis CI service.
‡Time-frame is measured in the number of days.

our rules from (to test on completely unseen data); second the dataset should have a sufficient number of CI skipped commits (that are explicitly marked by developers). To do so, we resorted to GitHub, and we searched for non-forked Java projects that use Travis CI and where their developers use the `[ci skip]` feature. To search for these projects on GitHub, we first use the BigQuery GitHub dataset, which provides a web-based console to allow the execution of a SQL query on the GitHub data[7]. We search for all non-forked Java projects that 1) contains the keywords `[ci skip] | [skip ci]` in more than 10% of their commit messages; and 2) do not exist in the TravisTorrent dataset. We choose projects with $>$ 10% of skipped commits, since this is a good indicator that the developers of those projects are somehow familiar with the Travis CI skip feature. We also eliminate the projects that exist in the TravisTorrent dataset, since our training data comes from the TravisTorrent dataset.

We found eleven projects that satisfy our selection criteria. However, we eliminated one project where all the CI skip commits were auto-generated, which left us with ten projects. Table 2 presents the project names, the number of commits after the introduction of Travis CI service to the project, and the percentage of CI skipped commits in the ten selected Java projects. In total there are 3,062 (average 306.20 and median 321.50) commits in all the selected projects. The table also shows that the percentage of actual CI skipped commits varies between 12.17 - 69.01% for projects in the testing dataset. It is important to note that we only consider commits after the use of Travis CI in the projects, since it presents the period of the project life where its developers start using the CI service.

4 Investigating the Reasons for [CI Skip] Commits

In this section, we describe the preliminarily analysis that we performed on the TravisTorrent dataset to understand when developers decide to CI skip in real world projects. Having this insight, our goal is to devise a rule-based technique to detect skipped commits. We then come up with a set of rules that is used to devise a rule-based technique to determine commits that can be CI skipped.

4.1 Identifying CI Skip Commits in the TravisTorrent Dataset

As mentioned earlier, developers can explicitly add the keyword `[skip ci]` or its variation `[ci skip]` to tell the Travis CI that they intend to skip a commit. Hence, we mine the commit messages and search for the skip keywords to obtain all of the skipped commits. After mining each project's data, we determine the time when the project started using Travis CI by identifying the commit that first introduced Travis's CI configuration file (`.travis.yml`). Then, we use a string pattern matching technique to automatically detect commits that are CI skipped, i.e., we searched using the term '`[skip ci] | [ci skip]`'. This was fairly straightforward since the skip keywords are very structured.

The goal of this section is to investigate how much this CI skip feature is used. Hence, we measured the number and percentage of skipped commits per project. It turns out that most projects did not `[ci skip]` commits - only 58 out of the 392 projects had one or more skipped commits.

Table 3 presents the statistics for the entire dataset of the 392 projects and the data of the 58 projects that have at least one skipped commit. We observe that overall, the mean number of skipped commits per project is 4.62 commits, which equates to (0.45%) of all commits. However, when we look at the projects that have at least one skipped commit (58 projects), we see that this percentage increases to 3.06% skipped commits on average. In addition, we observe that the 392 analyzed projects have a median of 33 (average = 49.18) developers, while the 58 projects that have at least one CI skip commit have a median of 40 (average = 67.60) developers. Our analysis also shows that the number of developers in our dataset is in the typical range of the number of developers in open source projects hosted on GitHub [3], [35], [36]. Table 3 also shows the time-frame of the studied projects (measured in days). For all projects in our dataset, the median number of days is 865 (average = 850), while for the projects that have at least one CI skip commit the median is 904.5 days (average = 897.5).

It is important to distinguish between the two sets, i.e., projects that have at least one skipped commits and all projects, since prior work showed that most developers may not know about the different features of CI tools, such as the ability to skip commits [19]. In any case, the projects with at least one skipped commit gives us a different view and at least for such projects we know that one or more developers knew about the skip functionality.

In total, we find 1,813 skipped commits in the TravisTorrent dataset. Overall, we observe that the number/percentage of the skipped commits can vary significantly for different projects, however, the detected number of skipped commits is large enough to enable the exploration and extraction of reasons that developers CI skip commits.

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

TABLE 4
The Manually Extracted Reasons for CI Skipped Commits.

Reason	Description	Number (%)	Information Source
Non-Source code files	Developers add or modify non source code file (e.g,. documentation files)	943 (52.01%)	Repository
Version preparation	Developers change the version of the project	274 (15.11%)	Repository
Source code comment	Adding , removing or editing source code comments	109 (6.01%)	Repository
Meta files	Developers modify meta files in the projects (e.g,. git ignore file)	68 (3.75%)	Repository
Formatting source code	Formatting source code without changing the semantic of the code	21 (1.16%)	Repository
Source code	Change that is made to source code of the project, but developers skip the build commit.	191 (10.54%)	Developer
Build change	Developer made change to build system they use.	112 (6.18%)	Developer
Tests	Change that is related to test cases of the project.	18 (1.00%)	Developer
Other		190 (10.48%)	Various

4.2 Reasons for CI Skip Commits

The first author manually analyzed all the 1,813 CI skip commits and identified the reason that developers skipped the commit. The first author applied an iterative coding process [30], where the first author first inspected every skipped commit by looking at its meta-data (e.g., commit message, etc.) and its associated source code in order to determine the reason for the commit being skipped. Every time a new reason for a CI skip is identified, we re-examine all the previously categorized commits to determine if categorization for a commit changed. As a result of this manual analysis, we were able to identify eight different reasons that developer CI skip a commit for.

Since this manual analysis heavily depends on human judgment, our classification is potentially prone to human bias. Thus, to examine the validity of our classification, we extracted a statistically significant sample of 317 (of the 1813 CI skip commits) commits to achieve a confidence level of 95% and a confidence interval of 5%. Then, we had the second author independently classify the 317 commits. After, the second author classified the 317 commits, we measured Cohen's Kappa coefficient to evaluate the level of agreement between the two annotators [7]. Cohen's Kappa coefficient is a well-known statistical method that evaluates the inter-rater agreement level for categorical scales. The resulting coefficient is a scale that ranges between -1.0 and +1.0, where a negative value means poorer than chance agreement, zero indicates exactly chance agreement, and a positive value indicates better than chance agreement. As a result of this process, we found the level of agreement between the two annotators to be +0.96, which is consider to be excellent agreement [13].

Table 4 lists the reasons, provides more detailed description, the number (and percentage) and information source needed for CI skipped commits in the history of the studied Java projects. The information source needed is related to the type of information that one needs to make a decision on whether a commit should be CI skipped or not. For example, if the commit changes non-source code files, one can easily infer such information from the project's repository. However, if the reason depends on the source code being modified, then such information is difficult to infer unless the developer or someone with domain specific knowledge provides it. We discuss this in more detail, later in Section 7.

In devising our rule-based technique to automatically detect CI skip commits, we focus on the five rules that can be inferred from the repository data since such reasons can be automated and applied to a wide number of projects. Below, we provide more details about each reason:

- **R1. Changes that touch documentation or non-source code files (52.01%):** The most common reason for developers to skip a commit build is when developers change, add, or delete non-source code files. For example, commits that change readme files, release notices, and/or adding logo to the projects.
- **R2. Changes related to preparing releases (Version preparation) (15.11%):** For this type of skip commit, developers simply prepare the project for release. For example, developers modify the version number of the project.
- **R3. Changes that only modify source code comments (6.01%):** Developers CI skip commits when they modify comments in the source code. For example, when they change the copyright of a source code file or modify the description of a partial source code.
- **R4. Changes that touch meta files (3.75%):** Developers tend to skip a commit when they change meta data of the project. For example, a developer may change a `.ignore` file, hence, they do not see any reason to build the project.
- **R5. Changes that format source code (1.16%):** Formatting source code is another reason that developers do skip commit for. For example, when a developer tries to improve the readability of the source code, they add a newline and/or spaces to reformat the source code.

In other cases, developers may CI skip for reasons that are not easily identifiable using the repository data. Below we detail the reasons that developers may CI skip a build:

- **D1. Commits that change source code (10.54%):** In this case a developer changed the source code of a project (e.g., add, delete, or/and modify source code), and CI skip the build. We discuss and provide a more comprehensive list of such commits in Section 7.
- **D2. Commits that change the configuration of the build system used in the project (6.18%):** In certain cases, developers change the configuration of the build systems

and CI skip that commit. Although the detection of such a commit can be easily automated, the decision that the developer makes to CI skip or not depends on the developers' themselves. For example, in certain cases, a developer may change the build configuration and decide to CI skip and in another they may not.

- **D3. Changes related to test cases of a projects (1.00%):** In these cases developers CI skip commit that change source code of test cases. For example, a developer adds a new test case to the project and decide not to build the project. Once again, this can be easily automated, however, the reason that a test-related change may get CI skipped or not depends on the developers themselves, which makes it difficult to automate.

There are cases (10.48%) where developers CI skip commits, but we cannot identify the exact reasons or some rare cases that are not worth having a separate category for. Finally, it should be mentioned that a build commit could contain more than one change type, hence the percentages above sum to more than 100%.

The most frequent type of changes that developers tend to skip build commits and can be inferred from the repository data are changing non-source code files, version preparation, source code comments, meta files, and formatting source code. The other three types of changes that developers CI skip a commit for require the developer knowledge which are changing source code, configuration of the build system, and test code.

4.3 Operationalization of Rules to Automatically Detect CI Skip Commits

As mentioned above, we use the CI skip commits to learn the different reasons that such commits exist. We do so in order to come up with a rule-based technique that can be used to automatically mark commits as CI skip commits. Hence, in this section, we detail the 'rules' in our rule-based technique, which are based on the aforementioned reasons.

Our rules are based on the reasons that can be extracted from software repositories (i.e., R1-R5). We focus on these reasons since they can be easily extracted, applied to any software project, and be fully automated. Below, we describe how we operationalized the rules used to detect CI skip commits:

Rule 1: Non-source code files: Similar to prior work [20], [37], we rely on the file extension to identify if a file change is a non-source code change (e.g,. readme file). We came up with a list of file extensions that indicate non-source code files (e.g., .md, .txt, and .png). Then, for each new commit, we check if the files changed are listed in the predefined list of non-source code extensions. In cases where the file does not have any extension, we check if the file is one that is not expected to affect the build (e.g., LICENSE, COPYRIGHT)[8]. In the case of the aforementioned files, we mark the commit with a CI skip.

Rule 2: Version release: We analyze the changed files in a commit and if the commit only modified the version in build configurations files, e.g., Maven or Gradle, then we mark the commit as a release preparation commit. Since such commits need not to be built, we mark the commit as a CI skip commit.

Rule 3: Source code comments: We consider changes that only modify the source code comments as changes that do not effect the build. Hence, we use regular expressions to remove the comments from the modified files. Since we analyze projects written in Java programming language, we considered all files ending with .java to be Java source code files and applied the following regular expression to each line of those files:

```
/\/\/(.*)|\/\*(\*(?!\/)|[^*])*?\*\//
```

We then check if the remaining lines modified by the change are the same, we consider the change as a source code comment change and mark it as a CI skip commit.

Rule 4: Meta files: As we did with non-source code files, we identify meta files by looking at the extensions of the files modified in the commit. We consider a commit in this category if it only modifies meta files in the repository such as .ignore file.

Rule 5: Formatting source code: To identify changes that only modify the format of the source code, we compare the current version of the file with the previous version of the file after removing all white spaces that are ignored by the Java language grammars. To be able to combine this rule withe *Rule 3*, we implement this process after removing the source code comment from both versions of the file. Thus, we use the devised rule-based technique to implement a tool (Details of the tool are in Section 8).

5 Case study Results

After understanding the reasons for CI skip commits and devising the rules to detect such commits, we would like to answer our research questions related to the effectiveness of our technique (RQ1) and the effort savings (RQ2). For each question, we describe the motivation behind the question, the approach to address the question, and present our findings.

5.1 RQ1: How effective is our rule-based technique in detecting CI skip commits?

Motivation: Since building the project after every commit can be wasteful (Duvall *et al.* [11], p. 87), we want to be able to effectively determine commits that can be CI skipped. Since it is now up to the developers to manually CI skip commits, we can use our rule-based technique to help automate this process and even recommend to developers if their commit should be CI skipped or not. Using the reasons we extracted from the current CI skipped commits, we devise a rule-based technique to detect skip commit. Thus, the goal of this research question is to examine how good is the defined rule-based technique in detecting skipped commits.

Approach: To determine how effective the devised rule-based technique is, we run the devised technique on all the projects in the testing dataset described in section 3.3. To evaluate the accuracy of our technique in detecting skip commits, we calculate the standard classification accuracy measures - recall and precision. In our study, recall is the

[8] A complete list of the file extensions can be found here: http://das.encs.concordia.ca/publications/which-commits-can-be-ci-skipped/

TABLE 5
Performance of Applying the Rule-Based Technique.

Project	Precision	Recall	F1-Measure (Relative F1-Measure)	AUC
TracEE Context-Log	0.91	1.00	0.96 (2.6X)	0.98
GrammarViz	0.57	0.89	0.70 (3.3X)	0.89
Parallec	0.80	0.97	0.88 (1.7X)	0.83
SAX	0.46	0.95	0.62 (1.9X)	0.80
jMotif-GI	0.32	0.79	0.46 (2.4X)	0.78
CandyBar	0.84	0.46	0.59 (1.0X)	0.63
Solr-iso639-filter	0.49	0.94	0.64 (1.4X)	0.62
SteVe	0.37	0.28	0.32 (1.1X)	0.58
Mechanical Tsar	0.69	0.20	0.31 (0.8X)	0.58
Trane.io Future	0.26	0.33	0.29 (1.1X)	0.56
Average	**0.57**	**0.68**	**0.58 (1.7X)**	**0.73**
Median	**0.53**	**0.84**	**0.61 (1.5X)**	**0.71**

percentage of correctly classified *Skip Commits* relative to all of the commits that are actually skipped (i.e. Recall $= \frac{TP}{TP+FN}$). Precision is the percentage of detected skipped commits that are actually skipped commits (i.e. Precision $= \frac{TP}{TP+FP}$). Finally, we combine the precision and recall of our defined rule-based technique in detecting skip commits using the well-known F1-measure (i.e. F1-measure $= 2 \times \frac{Precision \times Recall}{Precision + Recall}$).

Since our dataset is unbalanced (i.e., only a small percentage of commits are CI skipped), we would like to put our results in context by comparing it to a baseline that takes this imbalanced data into account. Similar to prior work [9], [32], we calculate the performance of the baseline model as follows: the precision of this baseline model is calculated by taking the total number of CI skip commits over the total number of commits of each project. For example, project `jMotif-GI` has a total number of 345 commits, of those, only 42 commits are commits that are explicitly labeled as CI skip commits. The probability of randomly labeling a commit as a CI skip commit comment is 12.17% (i.e., $\frac{42}{345}$). Similarly, to calculate the recall we take into consideration the two possible classifications available: CI skip or not. Once a prediction is made, there is a 50% chance that the commit will be classified as CI skip commit. Thus, the F1-measure for the baseline of the `jMotif-GI` project is computed as $2 \times \frac{0.1217 \times 0.5}{0.1217+0.5} = 0.098$.

Then, we divide the F1-measure from our technique with the baseline F1-measure and provide a *relative F1-measure*, which tells us how much better our technique does compared to the baseline. For instance, if a baseline achieves a F1-measure of 10%, while the defined rule-base technique achieves a F1-measure of 20%, then the relative F1-measure is given $\frac{20}{10} = 2X$. In other words, the defined rule-based technique performs twice as accurate as the baseline model. It is important to note that the higher the relative F1-measure value the better the model is in detecting CI skip commits.

Additionally, to mitigate the limitation of choosing a fixed threshold when calculating precision and recall, we also present the Area Under the ROC Curve (AUC) values. AUC is computed by measuring the area under the curve that plots the true positive rate against the false positive rate, while varying the threshold that is used to determine if a commit is classified as skipped or not. The advantage of the AUC measure is its robustness toward imbalanced data since its value is obtained by varying the classification threshold over all possible values. The AUC value ranges between 0-1, and a larger AUC value indicates better classification performance.

Results: Table 5 shows the result of the devised rule-based technique. We first present the precision, recall, F1-measure (relative F1-measure shown in parentheses), and AUC in the table. As we can see, the devised rule-based technique achieves an average F1-measure of 0.58 (median = 0.61) and average AUC of 0.73 (median = 0.71). This corresponds to a significant improvement of 72% in F1-measure over our baseline. The AUC at 0.73 is also significantly higher than the 0.50 baseline. As mentioned earlier, although these may seem like modest performance numbers, they are quite significant given the unbalanced nature of the data (i.e., only a small portion of the commits can be CI skipped).

Moreover, we see from Table 5 that for nine of the ten projects, we achieve an improvement in F1-measure and AUC over the baseline. The results show that our rule-based technique is effective in detecting CI skip commits, achieving an AUC of up to 0.98 for the `TracEE Context-Log` project.

However, in one particular project (`Mechanical Tsar`), our technique performs poorly. We manually investigated the data from this project to better understand the reasons for this poor performance. For the `Mechanical Tsar` project, we found a total of 134 commits that are explicitly marked to be CI skipped. The devised rule-based technique correctly identified 26 of these commits. For the remaining 108 commits, what we found is that they were related to either source code changes (which we cannot automatically skip without developer knowledge), changes related to Maven dependencies (which we believe should not be skipped, since they may break the project [31]) and changes related to the configuration of Docker containers (once again, changes that need developer knowledge to safely mark as CI skip).

It is important to note that there are two main factors that impact the performance of our technique. First, we test the technique against commits that are explicitly labeled by the developers to be CI skipped. In many cases, our rule-based technique is correct in flagging a commit to be CI skipped, however, the developer may have forgotten or not known that this commit should be CI skipped (we discuss this point in more detail in Section 10.2). This, of course, would result in a false negative and negatively impact the overall performance of our technique. Second, our technique is rule-based mainly due to the fact that we want it to be easily explainable and easy to apply (as we show later in Section 8).

Our rule-based technique can effectively classify CI skip commits with an average AUC of 0.73 and F1-measure of 0.58, which represents an improvement of 70% over a baseline model.

5.2 RQ2: How much effort can be saved by marking CI skip commits using our rule-based technique?

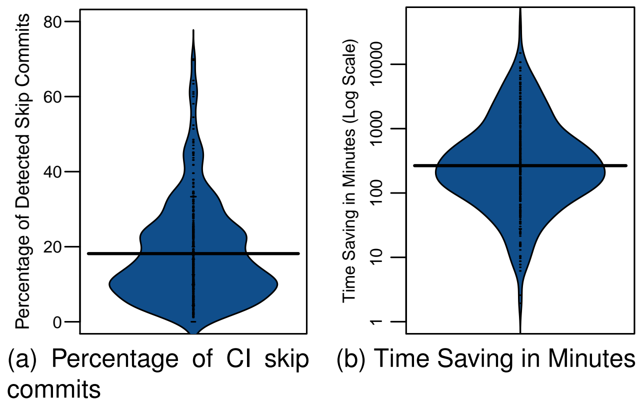
Motivation: After determining the effectiveness of the rule-based technique, we would like to know if applying this technique and marking some of the commits as CI skip commits would save significant effort for the project. Automatically detecting commits that can be CI skipped can reduce the amount of resources needed for the CI process and even speed up the overall development, making code reach its customers faster. Therefore, in this RQ we investigate the amount of effort that can be saved by applying our rule-based technique to the projects in our TravisTorrent dataset.

Approach: To address this research question, we evaluate the effort saving, by applying the defined rule-based technique on real build commits from the TravisTorrent dataset. We perform our analysis on projects from the TravisTorrent dataset since it contains build times and results, which enable us to measure effort. In total, we applied our technique on the 392 Java projects, which contained 193,833 build commits. We consider two complementary ways to measure the effort savings: first, we measure how many builds a project can save if they apply our rule-based technique; and second, we measure how much time a project saves when applying our technique on their commits. The idea is that, if the rule-based technique detects a build commit as a skip commit, the build status will not change, and there would be no need to build.

For every project in the TravisTorrent dataset (392 Java projects), we first apply the rule-based technique on all the commits. We then identify the *set of changes* in a *build commit*. We rely on the build linearization and commit mapping to git approach that is implemented in the TravisTorrent dataset [1], [2]. The approach basically considers the build history of a project on Travis CI as a directed graph and links each build to the commit on git that triggered the build execution. We refer readers to the original paper by Beller *et al.* for a full detailed explanation of the approach used by the TravisTorrent dataset [1], [2]. Second, we aggregate the results of the *set of changes*. This allows us to identify *build commits* (with all their associated changes) that should be skipped, and we predict that this build commit will result in a successful build (i.e, passed status from Travis CI). Then, we compare our predicted skipped build commit with the result from Travis CI.

We follow this methodology since a build commit can be associated with more than one commit (i.e., the set of changes) and we also want to only CI skip commits where the build is successful. Skipping a commit that causes the build to fail is not desirable since it means we may have let a failing build pass by. Finally, we measure the percentage of the build commits that we predict to be *passed*, since different projects will have different number of detected skip commits.

To measure the saved time, we use the time that is required for a build to be finished including setup time, build time and test run time. This time measurement is provided by Travis CI for every build. To put our results in context, we also measure the total time for all build commits and calculate the percentage of time saving of skip commits over all the commits. We argue that when a project does not build on build commits that we identify as skip build commit, this will save the projects's time.



(a) Percentage of CI skip commits (b) Time Saving in Minutes

Fig. 1. Beanplots showing the distributions of effort savings in terms of the number of CI skip commits and saved time for different values of Projects. The horizontal lines represent the medians.

Results: *Percentage of saved build commits:* Figure 1a shows the distribution of the saved build commits in all the studied projects. It shows that on average, 18.16% (median = 15.04) of the build commits in the studied projects can be CI skipped. As the figure shows, for some projects, the percentage of CI skipped commits can be more than 70% of their commits. This finding can have significant implications for software projects, especially given that prior research showed that developers often complain about their CI process slowing them down [18].

We also examined the top projects in terms of the percentage of commits to be CI skipped. We found that all of these projects are real Java projects (i.e., not just toy projects), where our technique can make a difference. For example, the projects, `Money and Currency API`, have more than 1,000 commits and 112 build commits after the introduction of Travis CI to the project, and according to our technique, 63.4% of their build commits can be CI skipped.

Amount of time saved: Figure 1b shows the distribution of time saving (in minutes) for skipped commits in all the studied projects. We find that, on average, 917 (median = 251.40) minutes can be saved per project if the classified commits are CI skipped. This time saving corresponds to an average of 10.70% (median = 15.17%) of the total time for all builds in our dataset. In some cases of certain projects, we can save up to 37,600 minutes (or approximately 626 hours) by CI skipping commits that need not be built.

Once again, we manually examined the top projects in terms of the time savings due to CI skips. We found that all of these projects are real Java projects where our technique can make a difference. At the top of the list is the *GeoServer* project where our technique can save 37,600 minutes since our technique shows that 17.61% of *GeoServer*'s 7,817 commits (2,951 build commits after the introduction of Travis CI) can be CI skipped.

Our rule-based technique can save developers, on average, 18.16% of their builds. These savings equate to an average time savings of 917 minutes in build time per project.

TABLE 6
Background of Survey Participants.

Experience	#	Developers' Position	#	Experience with CI (in years)	#
<1	0	Full-time Developer	30	<1	2
1 - 3	4	Part-time Developer	3	1 - 3	9
4 - 5	5	Freelance Developer	2	4 - 5	10
>5	31	Research Developer	5	>5	19

6 The Developers' Perspective

Although our results showed favourable results in terms of its accuracy and potential time savings (an average of 18.2% of their builds), one question that remains is whether such a technique is really needed by developers. To answer this question, we conducted a developer survey asking developers whether they consider the ability to CI skip commits as being important and why they CI skip commits.

We sent the survey to 512 developers whose names and emails were randomly selected from the 392 projects in the TravisTorrent dataset. In total, we were able to identify 19,231 developers in the dataset. We select a statistically significant sample to attain a 5% confidence interval and a 99% confidence level. This random sampling process resulted in 643 developers from 152 projects. We manually examined all the names and email address of the randomly selected developers to remove any incorrect email or possible duplicated developers and to make sure that we have personalized invitations that will increase the survey participation [34]. This manual analysis left us with 590 developers. Our survey was emailed to the 590 selected developers, however, since some of the emails were returned for different reasons (e.g., the email server name does not exist, out of office replies, etc.), we were able to successfully reach 512 developers. We received 40 responses for our survey after opening the survey for 10 days, i.e., the response rate is 7.81%. This response rate is acceptable comparing to the response rate reported in other software engineering surveys [33].

Survey Design: We designed an online survey that included three main parts. First we asked questions about the participant's background, development history and their experience using different CI services. We also asked whether the participants thought it is important to be able to CI skip commits or not and finally, we asked when/why might these developers CI skip a commit.

Table 6 shows the development experience of the participants, their position, and the number of years they have used CI services. Of the 40 participants in the survey, 31 participants had more than 5 years of development experience, 9 responses had between 1 to 5 years; 30 participants identified themselves as full-time developers and 5 participants as part-time or freelance developers, and the remaining 5 participants stated they are researchers who develop software. As for the experience of using CI services, 19 participants had more than 5 years of using CI, 10 respondents had between 4 to 5 years, 9 others had 1 to 3 years of experience, and finally two participants had less than 1 year of using CI. Overall, the participants are quite experienced in software development and using CI.

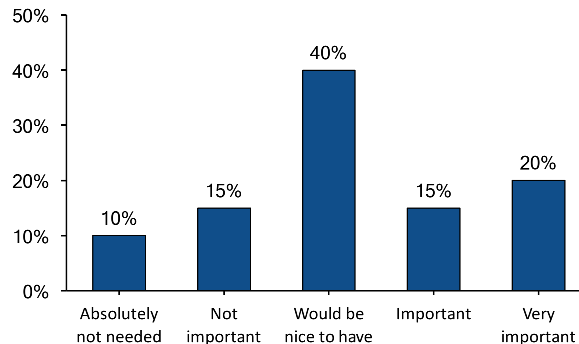


Fig. 2. Survey responses regarding the importance of being able to automatically CI skip a commit.

6.1 How important is it for developers to have the ability to automatically CI skip a commit?

We asked developers how important it is for them to have an automated way to skip the CI process for a commit or a pull request. In essence, our goal was to determine whether having our rule-based technique would be deemed favorable. To avoid bias [23], [24], we asked them to answer the question on a five-point likert-scale, ranging from 1= absolutely not needed to 5= very important. Figure 2 reports the result related to developers's opinions about the importance of having the ability to automatically CI skip a commit. Of the 40 participants, 35% indicated that it is important or very important and 75% indicated that it would nice, important or very important to have a technique that automatically helps them determine a CI skip commit. On the other, the remaining 25% considered such an approach to be not important or absolutely not important. We believe that these results clearly indicate that having a technique to help automatically CI skip commits is needed by developers.

6.2 When do developers skip the CI process?

In addition to simply asking whether it is important to have a technique, we also asked participants in which cases and why they CI skip a commit. We provided a free-form text box for participants to reply in. Since the responses for these two questions are free-text, we collected all of the responses and manually analyze them. The first two authors carefully read the participant's answers and came up with a number of categories that the responses fell under. Next, the same two authors went through the responses and classified them according to the extracted categories. To confirm that the two authors correctly classified the responses to the right category, we measure the classification agreement between the two authors. To do so, we use the well-known Cohen's Kappa coefficient [7]. For the two questions, we found the level of agreement was +0.87 for when developer skip CI process and +0.82 for the question why do developers skip CI process. Finally, for the few cases that annotators failed to agree on, the third author was consulted to resolve the differences and categories these cases.

We were able to identify three main categories for when developers skip the CI process. Also there are some small cases that we group into one category as "Other". In the fol-

lowing, we present these cases and provide some examples of participant replies under each category:

Non-Source Code Changes (71.79%): The majority of the participants indicated that changing non-source code is the main reason when decide to skip CI process. For example, P21 stated: *"documentation updates, trivial updates that don't affect execution"* and P30 *"I usually forget. But a documentation/README change is the most likely."*. Note here that P30 explicitly mentioned that he/she forgets to CI skip, which is exactly why we believe our technique will be very useful since it automatically flags such commits.

No Test Code Coverage (15.38%): The second main case of skipping CI process based on the participant's responses is when the changed source code is not covered with tests. Some example responses that mention these cases are stated by participant P14: *"When tests are not written to work for that particular source branch/repo"*.

Trivial Source Code Changes (2.82%): A less common case for skipping the CI process is when the commit is performing a trivial source code change or fixing a trivial bug. For example, P11 stated that he/she skips the CI process when fixing a small bug; *"small bug fixes"*. Another example P31 stated that *"partial jobs already ok"*.

Other (27.50%): Other cases cited by the participants for when they skip CI process. In these cases developers reported various cases related to the source code changes and/or build scripts such as refactoring. For example, P27 stated that *"build settings (cmake, etc...), refactoring, etc..."*. Other cases such as increasing the development speed or skipping that build that is known to be failing. Some examples of such cases reported by P18 & P40 as follows: P18 *"When something is broken and skipping CI will (probably) fix things faster."* & P40 *"redundant builds and build that are known to fail"*.

Also, three participants stated that they do not CI skip any commit for the open source projects that they contribute to since they are required to run the CI process on all commits. For example, P1 said that *"the main GitHub repos in which I work require a passing CI build to merge PRs so I never skip CI for those repos, even for docs only changes."*

6.3 Why do developers skip the CI process?

We followed the same approach described above to classify the answers of the why question. After the classification of the participants' answers, we extracted three main reasons. There are also a few cases that we group them into one "Other" category:

No Change in Build Result (43.59%): Nearly half of the participants reported that when they expect the build result will not change, they skip the CI process. Some example are reported by P32 and P4 as follows: P32 said that *"Because I have recently seen a successful run (within minutes) and have faith that my docs have not changed any code."* and P4 *"Because rerunning the CI would most likely be redundant as nothing in the result would change."*

Saving Time (35.90%): The second most cited reasons for skipping the CI process is to save development time specially when building the project and running the tests takes a long time. For example, participant P16 mentions that *"To save time. Some projects have CI that takes 30 minutes or more to complete with multiple PRs/branches that are competing for CI resources."*. Other participant believe when the CI process takes a long time, it could block other developers and result in slowing down the development. For example, P26 stated *"Because the build process is quite long and I don't want to block up the queue for someone else who's working."*. These examples clearly show that having such a technique is needed by developers in practice.

Saving Computation Resources (28.21%): In some cases running the CI process on every commit to the project seen as wast of resource by the survey participants. For example, as P5 and P1 state: P5 *"I do not want to waste resources on effectively unchanged codebase"* & P1 *"I skip CI when it's unnecessary so as to not waste computing resources."*

Other (10.00%): In these cases, developers cited different reasons for skipping the CI process such as when they contribute to a small project or they run the build and run the test cases locally. For example, P35 stated *"Only for small projects."*, P37 said *"some because the ci can result in unexpected deploy..."*, and P40 *"If I expect the commit to fail, I might skip the CI build"*.

7 Discussion

In this section, we discuss areas where our rule-based technique can be improved, and present results of how source code analysis techniques may potentially improve performance.

7.1 Special Cases of CI Skip Changes.

Although our rule-based technique is able to significantly outperform the baseline, it still misses some cases of commits that should be CI skipped. Therefore, in this subsection, we manually examine the cases that are missed by our rule-based technique to better understand where (and how) our technique can be improved. We examined commits that were explicitly marked by developers as a CI skip commit (i.,e. contains the keyword `[ci skip]` in their commit message), however, our rules missed. Below, we list the different types of CI skip commits that our rule-based technique missed:

SC1. Renaming variables, methods, or/and classes: In many cases, developers tend to CI skip commits when they rename Java objects (e,.g. variables, methods or/ and classes). The following is an example of a skip commit where a developer commits a change to rename a method name.

```
- public AgreementReport getAgreement() {
+ public AgreementReport
    getAgreementReport() {
return new
    AgreementReport.Builder().compute(stage,
    answerDAO).build();
```

Although it is easy to detect such renaming, it is very difficult to argue that all such cases can be CI skipped. However, one can devise project-specific rules that can learn if all renaming commits in a certain project are skipped, and apply such a rule for that one project. We plan to investigate the introduction of such project-specific rules in the future.

SC2. Optimizing import statements: To use a library in Java (built-in or third-party), developers need to use import statements that specify part of the library or more general declaration. In the skip commit related to this case, developers may decided to optimize the declaration of some Java libraries by specifying parts of the library or make it more general. For example, developers commit a source code change where they specify the type of Java collection they use instead of the general declaration as it is shown in the following commit:

```
- import java.util.*;
+ import java.util.Collection;
+ import java.util.Collections;
+ import java.util.Map;
```

Once again, although it would be trivial to devise a rule that CI skips commits that optimize the import statements, it is not the case that all commits that optimize import statements should be CI skipped. Hence, it is difficult to automate the skipping of this type of commits.

SC3. Java annotation: In Java, source code annotations can be used for several reasons such as documentation or to force the compiler to execute specific code snippets. We found cases that developers skip commits when they add/modify/delete Java annotations. The following shows an example for such a skip commit.

```
@JsonProperty("workerRanker")
+  @SuppressWarnings("unused")
public String getWorkerRankerName() {
return workerRanker == null ? null :
    workerRanker.getClass().getName();
```

Most likely, the developers feel that the project need not be built after such a change. However, in some cases the develops may feel that indeed the project needs to be built.

SC4. Modifying the log or exception message: Developers add logs or exception messages to help in the debugging of their Java programs. However, since often the log message do not affect the functionality of the program, developers tend to CI skip such commits. In the following example, a developers CI skips the commit that modifies the log message and in the other the developers CI skips the commit that modifies the exception message.

```
logger.info(String
, ... ,
-  progressPercent, secondElapsedStr, "",
+  progressPercent, secondElapsedStr,
    hostName,
... );
```

```
catch (Exception e) {
- fail("sholdn't throw an exception,
    exception thrown: \n" +
    StackTrace.toString(e));
+ fail("shouldn't throw an exception,
    exception thrown: \n" +
    StackTrace.toString(e));
e.printStackTrace();
```

Although we can automate this case with a rule, we decided not to since in other cases, developers build the project when they modify the log or exception messages. This is primarily due to the fact that they will often perform more than one modification per commit (e.g., fix a bug and update he log message).

SC5. Refactoring source code: In certain cases, developers perform some refactoring procedure on the Java source code (e,.g. moving method, or split Java classes into two or more classes) and decide to CI skip the commit. Although we believe that the project should be built after a source code refactoring, in some cases developers tend to skip them.

One way to detect some of the aforementioned changes is through the use of source code analysis. In the next subsection, we examine the applicability of using source code analysis to enhance our rule-based technique in detecting CI skip commits.

TABLE 7
Performance of Rule-Based Technique with ChangeDistiller.

Project	Precision	Recall	F1-Measure (Relative F1-Measure)	AUC
TracEE Context-Log	0.91	1.00	0.96 (2.6X)	0.98
GrammarViz	0.57	0.89	0.70 (3.3X)	0.89
Parallec	0.80	0.97	0.88 (1.7X)	0.83
SAX	0.46	0.95	0.62 (1.9X)	0.80
jMotif-GI	0.32	0.79	0.46 (2.4X)	0.78
CandyBar	**0.86**	**0.56**	**0.68 (1.2X)**	**0.68**
Solr-iso639-filter	0.49	0.94	0.64 (1.4X)	0.62
SteVe	**0.43**	**0.34**	**0.38 (1.4X)**	**0.62**
Mechanical Tsar	**0.86**	**0.54**	**0.67 (1.6X)**	**0.75**
Trane.io Future	0.26	0.33	0.29 (1.1X)	0.56
Average	**0.60**	**0.73**	**0.63 (1.9X)**	**0.75**
Median	**0.53**	**0.84**	**0.65 (1.7X)**	**0.77**

7.2 Can Source Code Analysis Enhance the Detection of CI Skip Commits?

As we have seen in the previous subsection, the rule-based technique misses some CI skipped commits. It is evident that such missed cases may be better detected through source code analysis. In this section, we apply and investigate the effectiveness of using source code analysis in detecting CI skip commits.

To perform the source code analysis, we use CHANGEDISTILLER [14], a well-known tool that identifies statement-level structural changes between Java Abstract Syntax Tree (AST) pairs. CHANGEDISTILLER presents the differences between two source code files as edit scripts, or sequences of edit operations (e.g., insertions, deletions, or updates) involving structural entities at varying levels of granularity. It relies on a measure of textual similarity between statement versions to detect cases where a statement was modified.

Since we need to make our decisions at the commit level, we wrote a script to augment the output from CHANGEDISTILLER so that it applies at the commit level. More specifically, we extract all the source code Java file pairs (modified and original) for each file touched in a commit. We then use CHANGEDISTILLER to extract the fine-grained source code changes between each pair of revisions. CHANGEDISTILLER takes as input the pair of revision files and creates two Abstract Syntax Trees (ASTs) that are used to compare these

TABLE 8
Mann-Whitney Test (p-value) and Cliff's Delta (d) for Using Only Our Rule-Based Technique and With the Integration of Source Code Analysis.

Metrics	p-value	d
Precision	0.8201	-0.07 (negligible)
Recall	0.7047	-0.11 (negligible)
F1-Measure	0.5443	-0.17 (small)
AUC	0.7327	-0.1 (negligible)

revisions. As a result, CHANGEDISTILLER outputs a list of fine-grained source code changes (e.g., an update in a method invocation or rename).

We analyzed the result of CHANGEDISTILLER to determine how the additional information from the tool can help enhance our rule-based detection technique. We find two main cases. First, we find cases where the output of CHANGEDISTILLER provides the same information as one of our rules (e.g., formatting changes that are due to white space). In such cases, we do not consider the output to be necessarily useful, since our simple rules that are more lightweight can detect such cases. Second, we find cases where there is a change in the code, but there are no changes in the AST pairs. Note that although there is change in the ASTs, there may be changes in the nodes (e.g., if one of the nodes is renamed). We find that such output can contain useful information since it can indicate an ideal case of a CI skip commit.

Since the first case (i.e., where the output of CHANGEDISTILLER is similar to our rules is not interesting, we focus on the output of the second case. We find that output of CHANGEDISTILLER in the second case can fall into three main types of changes:

- **Simple renaming.** For commits where simple renaming are done, CHANGEDISTLLER will output a change in the nodes of the AST, but no change in the structure of the AST itself. We use this output as an indicator of a commit that can be CI skipped. This case handles a subset of SC1 mentioned in Section 7.1.
- **Java annotations.** We noticed that cases where CHANGEDISTLLER provides no output (neither in the AST or the nodes) can indicate cases where Java annotations are added/modified. We use this no output as an indicator of a commit that can be CI skipped. This case handles SC3 mentioned in Section 7.1.
- **Minor code restructuring.** Again, we noticed that cases where CHANGEDISTLLER provides no output (neither in the AST or the nodes) can indicate cases where minor code restructuring is performed. We use this no output as an indicator of a commit that can be CI skipped. This case handles a subset of SC5 mentioned in Section 7.1.

To determine how much using code analysis can improve our rule-based technique, we perform an experiment to compare the performance improvement of the rule-based technique vs. rule-based and code analysis. We re-ran the experiment using the ten open source Java projects in the testing dataset, listed in Table 2. We measure performance using precision, recall, F-measure, and AUC.

Table 7 shows the result of our rule-based technique with

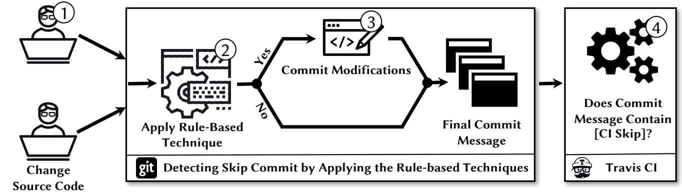


Fig. 3. The workflow of the designed CI-SKIPPER tool.

the integration of source code analysis technique. We find that using source code analysis improves the performance of our rule-based technique for only three projects of the 10 projects, namely `CandyBar`, `SteVe`, and `Mechanical Tsar` (highlighted in bold in Table 7). Using the additional information from CHANGEDISTLLER slightly improved the overall performance of our rule-based technique, increasing the average F-measure from 0.58 to 0.63 and AUC from 0.73 to 0.75.

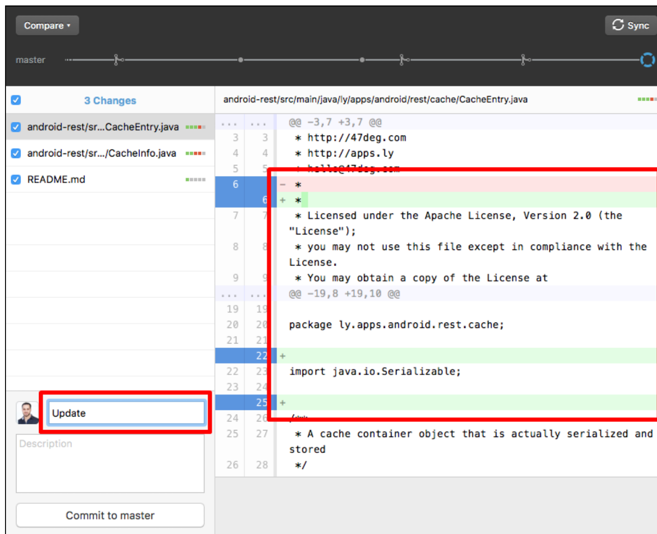
To understand the cases where source code analysis helps in detecting CI skip commits, we examined the cases that improved our performance. We found 65 cases from the three projects, where source code analysis helped in flagging CI skip commits, which were missed by the rule-based technique. The first two authors manually examined each of the 65 cases. We find that of the 65 commits, 73.9% are related to simple renaming and restructuring, 10.8% are related to annotations and another 15.4% are related to changes in import statements. In addition, to examine whether the difference in performance between using only the our rule-based technique and integrating source code analysis is statistically significant, we performed a Mann-Whitney test. We also use Cliff's Delta (d), which is a non-parametric effect size measure to interpret the effect size. As recommended in prior work [16], we interpret the effect size value to be small for $d < 0.33$ (positive as well as negative values), medium for $0.33 \leq d < 0.474$ and large for $d \geq 0.474$.

Table 8 shows the p-values and effect size values. It shows that for all performance measures the differences are not statistically significant, having p-values > 0.05. Also, the effect size values are small or negligible. The result show that although we see that code analysis does help, its improvement in performance is not statistically significant. However, we believe that if certain projects have many changes that are related to source code, such source code analysis may be worth the extra effort.

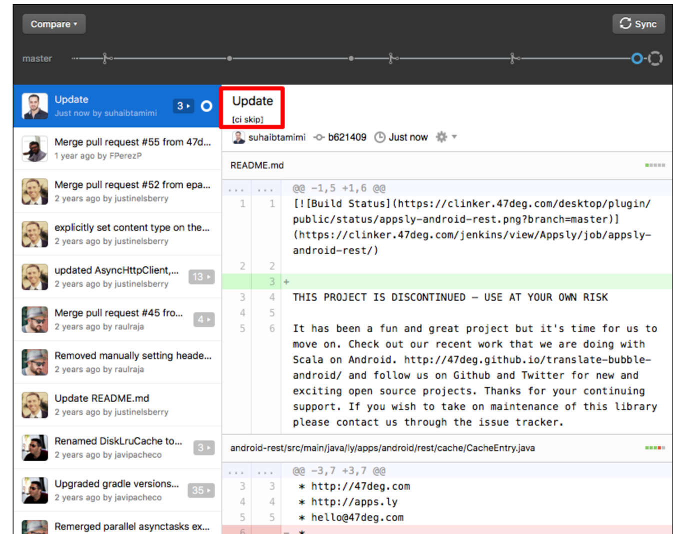
Given that 1) the rule-based technique is lightweight, 2) it can be applied without the additional installation of a source code analysis tool, and 3) that the performance improvements of source code analysis are not statistically significant, in the next section, we detail a prototype tool that can automatically detect CI skip commits using our devised rules.

8 TOOL PROTOTYPE: CI-SKIPPER

One of the main reasons that we preferred to use a rule-based technique is that it can be easily implemented. As we showed in RQ1, our rule-based technique is very effective in some projects, hence applying this technique can yield large



(a) Sample commit that does not modify any source code.



(b) The commit is tagged with `[ci skip]`.

Fig. 4. Screen shots of CI-Skipper. The commit is automatically detected to be CI skipped and the 'ci skip' tag is added to the commit message.

resource savings. However, as prior work has shown [19], in some cases, developers may not even know that CI skip is a feature of their CI framework. Therefore, we believe that devising a tool that can automatically detect and pre-label commits with CI skip would be very beneficial.

We built a tool, called CI-Skipper, that applies our rule-base technique. The tool is very lightweight and is easily integrated with any source code control versioning system. Our prototype was built to work with Git, since it is one of the most popular source code versioning systems today [5].

Figure 3 shows the workflow of CI-Skipper. Every time a developer commits a change to the repository (step 1), CI-Skipper is triggered through a `git` hook. Once triggered, CI-Skipper analyzes the change made by the commit's files, applying the defined rules to determine whether the commit should be CI skipped (step 2). If the commit should be CI skipped, CI-Skipper modifies the commit message by adding the tag `[ci skip]` in the commit message (step 3). Finally, when the commit is pushed to the remote repository, the CI system, which will get triggered examines the commit message and upon seeing the tag CI skip will skip kicking off the CI process for that commit (step 4).

CI-Skipper was developed to be easily installed and enabled/disabled. Figure 4 shows screen shots of how CI-Skipper works with an existing Git repository. CI-Skipper is free and publicly available. It can be easily installed from the node package manager `npm` by running the following command in the console.

```
npm install -g ci-skipper
```

Once installed, CI-Skipper can be enable or disabled with the following commands:

```
ci-skipper on //enable CI-Skipper.
ci-skipper off //disable CI-Skipper.
```

Through our use and testing of CI-Skipper, it performed well (without any noticeable overhead) and applied the rules correctly, by marking commits that fit our rules with `[ci skip]`.

9 Related Work

In this section, we present the work most related to our study. We divide the prior work into two main areas; work related to the improvement of CI technology and work related to the usage of CI.

9.1 Improvement of CI Technology.

There is a limited number of studies that investigate the possibility to improve CI tools. Brandtner *et al.* [4] introduced a tool called SQA-Mashup that integrates data from different CI tools to provide a comprehensive view of the status of a project. Campos *et al.* [6] propose an approach to automatically generate unit tests as part of the CI process. Other researchers investigated the improvement of communication between developers who use CI in their projects. They find that CI provides a mechanism to send notifications of build failures [12], [25]. Downs *et al.* [10] conducted an empirical study by interviewing developers and found that the use of CI substantially affect the team's work-flow. Based on their findings, a number of guidelines were suggested to improve the CI monitoring and communication when using CI.

Other work has focused on detecting the status of builds and investigated the reasons for build failures. Hassan and Zhang [17] used classifiers to predict whether a build would pass a certification process. Rausch *et al.* [28] collected a number of build metrics (e.,g. file type and number of commits) for 14 open-source Java project that use Travis CI in order to better understand build failures. Among other findings, their study showed that harmless changes sometimes break builds but this often indicates unwanted flakiness of tests or the build environment. Seo *et al.* [31] studied the characteristics of more than 26 million builds done in C++ and Java at Google. They found that the most common reason for builds failures is the dependencies between components. Ziftic and Reardon [41] propose a technique to automatically detect fail regression tests of CI build.

The technique is based on heuristics that filter and rank changes that might introduce the regression. Zampetti *et al.* [40] studied the use of automated static code analysis tools in Travis CI. Their findings show that static code analysis tools checks are responsible for only 6% of broken builds. Miller [27] reported the use of continuous integration in an industrial setting, and showed that compilation errors, failing tests, static analysis tool issues, and server issues are the most common reasons for build failures.

In the same line with this existing research, our work investigates the reasons software developers skip build commit. However, we focus on the detection of build commits that do not change the status of the build, or commits that need not be built.

9.2 Usage of CI.

A number of recent papers examined the usage of CI in the wild. Most of these studies performed surveys to gather feedback from CI users in order to better understand why it is so commonly used, as well as, what could be improved in the process.

Hilton *et al.* [19] investigated the cost, benefits, and usage of CI in open source projects. They found that CI improves the release cycle, however, developers tend not to be familiar with the many CI features. In another study, Hilton *et al.* [18] studied the usage of CI in the proprietary projects. Their findings showed that similar to open source projects, developers of proprietary projects agreed that CI is efficient to catch errors earlier and allows developers to worry less about their builds while also providing a common build environment for every contributor. However, CI can induce long build times (which is specifically a problem that our technique aims to solve) while also requiring a lot of set up to use and automate the build process. Developers have also complained about the lack of integration of new tools and debugging assistance when a build fails. Leppnen *et al.* [22] investigate the benefits of CI by conducting a semistructured interview with developers from 15 companies. Their study showed that faster feedback and more frequent releases are the most mentioned benefits of using CI.

Beller *et al.* [1] analyzed CI builds of open source projects written in Java and Ruby on GitHub. Their results showed that the main reasons for failed builds are failing test. They also found that getting results from CI builds requires, on median, 10 minutes. Our technique helps reduce this time by suggesting commits that can be CI skipped. Vailescu *et al.* [38] studied the quality outcomes for open-source projects that use CI services. Their findings showed that using CI has some positive outcomes on the open-source projects (e.g,. the productivity of project teams). Yu *et al.* [39] studied the impact of using CI on software quality. CI helps detecting bugs that are in a few files. Santos and Hindle [29] use build statuses reported by Travis CI as a measure of source code commit quality.

As shown in the aforementioned work, CI can improve the quality and the productivities of software development. However, getting results from CI can take considerable time for some projects. Hence, our work addresses this issue by detecting which commits can be CI skipped and automatically labels them for the developers.

10 Threats to Validity

In this section, we discuss the threats to internal, construct and external validity of our study.

10.1 Internal validity

Internal validity concerns factors that could have influenced our results. Our analysis heavily depends on the TravisTorrent dataset, which links commits from GitHub and Travis CI. There may be missing or incorrect links in the dataset, which would impact our analysis. To examine the correctness of these links, we manually checked the accuracy of a subset of these links and found the links in all of our cases to be correct. To identify the type of file changes in a commit, we use a list of extensions of the most common file types (e.g., readme files, etc.) provided in previous work [37]. In some cases, the list of file types we use may not be comprehensive. We also provide a list of all the file extensions that are used in our study [9].

To gain insight on the importance and use of the CI skip feature from developers, we conducted an online survey. We contacted 512 developers, and received 40 (7.81%) responses. While this response rate may seem to be a small number, it is within the acceptable range for questionnaire-based software engineering surveys [33]. Also, since the respondents to our survey voluntarily chose to respond, we may suffer from self-selection bias. To mitigate this bias, we tried to select our initial set of 512 developers randomly from different projects. Moreover, social desirability bias might have affected the response from our respondents, causing their responses to support the idea of CI skipping commits. To reduce this social desirability bias, we randomly contacted developers who may or may not use the CI skip feature provide by CI services.

10.2 Construct validity

Construct validity considers the relationship between theory and observation, in case the measured variables do not measure the actual factors. The rules we extracted are based on the projects we examined. Hence, an examination of a different set of projects may lead to different rules. However, we examined more than 1,800 skip commits, which gives us confidence in our extracted rules. In addition, in our manual examination of commits in Section 4.2, the first author performed the classification task since most of the rules were very straightforward (e.g., a commit only changes code comments or a help file). However, to ensure the validity of our classification, we got the second author to classify a statistically significant sample of 317 changes commits and found their agreement to be excellent (Cohen's Kappa value of +0.96).

The CI skip commits we examined are commits that are explicitly marked as so by developers. In some cases, developers may forget to label commits that should be skipped with `[ci skip]` or `[skip ci]`. To evaluate the devised rule-base technique we selected extra ten open source Java projects where developers explicitly mark at least 10% of the commits as skip commits. Also, the performance of the

[9]http://das.encs.concordia.ca/publications/which-commits-can-be-ci-skipped/

devised rule-based techniques shows that, on average, it achieves an AUC of 0.73, which is modest performance. To examine why our performance is not higher, we examine all the cases that the rule-based flagged as CI skip commits and the developers did not, and vice versa (i.e., false positives/negatives), we found that 99% of the cases that we flagged as CI skip commits are cases that should be CI skipped, however, developers tended to miss them. For the cases that we did not flag and developers skip them, we found that developers do indeed change source code or update dependences in these commits, but opt to skip the CI process. We believe that it is very difficult to detect such commits without the developers knowledge.

To answer our second research question, we measure the time required for the project to finish the CI processes. However, build-time heavily depends on the different configurations of Travis CI (e.g., using commands in the wrong phase, etc.), which may affect our results [15]. However, since our findings are based on a large number of projects, we expect the effect of the Travis CI configurations to be minimal.

10.3 External validity

Threats to external validity concern the generalization of our findings. Our study is based solely on Java projects, hence our findings may not hold for projects written in other programming languages. However, our approach of defining the rule-based techniques can be easily generalized to other programming languages by analyzing the skip commits of the other projects written in different programming languages. Second, the two datasets used in our study present only open source project hosted on GitHub that do not reflect proprietary projects. Furthermore, we examine projects that use Travis CI for their continuous integration services, and different CI platforms could have more advance features for controlling skip commits. That said, Travis CI is the most popular CI services on GitHub[10] that have a basic feature of skipping unrequited build commits.

According to a recent study, Travis CI is the most popular CI service on Github [19]. In this study we focus on implementing out technique using Travis CI. However, our technique is applicable to other CI services (e,.g. Circle CI[11], AppVeyor[12], and CodeShip[13]) that allow developers to skip commit using the CI skip feature or other CI services that provide a plugin to add this CI skip feature, such as Jenkins[14], Hudson[15], and Bamboo[16].

11 Conclusion

In this paper, we study CI skip commits that developers tend not to build a project on. We analyze the commit history of 392 open source Java projects provided by TravisTorren dataset [2]. We first investigate the reasons why developers CI skip commits and found that developers skip Travis CI build for eight main reasons for which five can be automated; changes that touch only documentation files, changes that touch only source code comments, changes that formatting source code, changes that touch meta files, prepare for releases. We then propose a rule-based technique to automatically detect the CI skip commits. We evaluate the accuracy of the defined rule-based technique using a testing dataset of ten Java projects that their developers use Travis CI skip feature. We found that the technique achieves F1-measure of 0.58 (AUC of 0.73) on average. We further applied our technique on all the 392 studied commits, and found that our technique is able to save up to 18.16% of a project's commits and amounting to a saving of 917 minutes on average.

In addition, through an online survey of 40 developers, we found that 75% of the developers believe it would be nice, important or very important to have a technique that automatically indicated CI skip commits. Developers also indicated that they CI skip commits to save development time and computational resources. Finally, we developed a tool based on the proposed technique that is available for public.

The results in this paper outline some directions for future work. First, as we discuss in the section 7, examining the impact of using of source code analysis on the improvement of the rule-base technique that we will examine in the future in more detail. Another interesting cases that require more in-depth investigation are the cases where the build commits are predicted to be CI skip commits but they result in fail builds, we plan to conduct a study to understand such cases. Finally, we want to investigate the potential of using machine learning techniques to improve the performance of detecting CI skip commits as potential future work as well.

[10]https://blog.github.com/2017-11-07-github-welcomes-all-ci-tools
[11]https://circleci.com/
[12]https://www.appveyor.com/
[13]https://codeship.com/
[14]https://jenkins.io/
[15]http://hudson-ci.org/
[16]https://www.atlassian.com/software/bamboo

References

[1] 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 14th International Conference on Mining Software Repositories*, MSR '17, pages 356–367. IEEE Press, 2017.

[2] M. Beller, G. Gousios, and A. Zaidman. Travistorrent: Synthesizing travis ci and github for full-stack research on continuous integration. In *Proceedings of the 14th working conference on mining software repositories*, MSR '17, 2017.

[3] T. F. Bissyand, F. Thung, D. Lo, L. Jiang, and L. Rveillre. Popularity, interoperability, and impact of programming languages in 100,000 open source projects. In *2013 IEEE 37th Annual Computer Software and Applications Conference*, pages 303–312, July 2013.

[4] M. Brandtner, E. Giger, and H. Gall. Supporting continuous integration by mashing-up software quality information. In *Proceedings of the Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering*, (CSMR-WCRE) '14, pages 184–193. IEEE, 2014.

[5] C. Brindescu, M. Codoban, S. Shmarkatiuk, and D. Dig. How do centralized and distributed version control systems impact software changes? In *Proceedings of the 36th International Conference on Software Engineering*, ICSE '14, pages 322–333. ACM, 2014.

[6] J. Campos, A. Arcuri, G. Fraser, and R. Abreu. Continuous test generation: Enhancing continuous integration with automated test generation. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 55–66. ACM, 2014.

[7] J. Cohen. A coefficient of agreement for nominal scale. *Educational and Psychological Measurement*, 20:37–46, 1960.

[8] M. A. Cusumano and R. W. Selby. *Microsoft Secrets: How the World's Most Powerful Software Company Creates Technology, Shapes Markets, and Manages People*. The Free Press, 1995.

[9] E. d. S. Maldonado, E. Shihab, and N. Tsantalis. Using natural language processing to automatically detect self-admitted technical debt. *IEEE Transactions on Software Engineering*, 43(11):1044–1062, 2017.

[10] J. Downs, J. Hosking, and B. Plimmer. Status communication in agile software teams: A case study. In *Proceedings of 2010 Fifth International Conference on Software Engineering Advances*, ICSEA '10, pages 82–87. IEEE, 2010.

[11] P. M. Duvall, S. Matyas, and A. Glover. *Continuous Integration: Improving Software Quality and Reducing Risk (The Addison-Wesley Signature Series)*. Addison-Wesley Professional, 2007.

[12] S. Dsinger, R. Mordinyi, and S. Biffl. Communicating continuous integration servers for increasing effectiveness of automated testing. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 374–377. IEEE, Sept ASE '12.

[13] J. L. Fleiss and J. Cohen. The equivalence of weighted kappa and the intraclass correlation coefficient as measures of reliability. *Educational and Psychological Measurement*, 33:613–619, 1973.

[14] B. Fluri, M. Wuersch, M. PInzger, and H. Gall. Change distilling:tree differencing for fine-grained source code change extraction. *IEEE Transactions on Software Engineering*, 33(11):725–743, Nov 2007.

[15] K. Gallaba and S. McIntosh. Use and misuse of continuous integration features: An empirical study of projects that (mis)use travis ci. *IEEE Transactions on Software Engineering*, pages 1–1, 2018.

[16] R. J. Grissom and J. J. Kim. *Effect sizes for research: A broad practical approach.* Lawrence Erlbaum Associates Publishers, 2005.

[17] A. E. Hassan and K. Zhang. Using decision trees to predict the certification result of a build. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, ASE '06, pages 189–198. IEEE Computer Society, 2006.

[18] M. Hilton, N. Nelson, T. Tunnell, D. Marinov, and D. Dig. Trade-offs in continuous integration: Assurance, security, and flexibility. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE '17, pages 197–207. ACM, 2017.

[19] 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 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE '16, pages 426–437. ACM, 2016.

[20] A. Hindle, M. W. Godfrey, and R. C. Holt. Release pattern discovery via partitioning: Methodology and case study. In *Proceedings of the Fourth International Workshop on Mining Software Repositories*, MSR '07, pages 19–. IEEE Computer Society, 2007.

[21] 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 11th Working Conference on Mining Software Repositories*, MSR '14, pages 92–101. ACM, 2014.

[22] M. Leppnen, S. Mkinen, M. Pagels, V. P. Eloranta, J. Itkonen, M. V. Mntyl, and T. Mnnist. The highways and country roads to continuous deployment. *IEEE Software*, 32(2):64–72, Mar 2015.

[23] J. LinÂker, S. M. Sulaman, R. M. de Mello, M. Hst, and P. Runeson. Guidelines for conducting surveys in software engineering. *Technical report*, 2015.

[24] I. Manotas, C. Bird, R. Zhang, D. Shepherd, C. Jaspan, C. Sadowski, L. Pollock, and J. Clause. An empirical study of practitioners' perspectives on green software engineering. In *Proceedings of 2016 IEEE/ACM 38th International Conference on the Software Engineering*, ICSE '16, pages 237–248. IEEE, 2016.

[25] K. Matsumoto, S. Kibe, M. Uehara, and H. Mori. Design of development as a service in the cloud. In *Proceedings of 15th International Conference on the Network-Based Information Systems*, NBiS '12, pages 815–819. IEEE, 2012.

[26] J. Micco. Tools for continuous integration at google scale - youtube, August 2012.

[27] A. Miller. A hundred days of continuous integration. In *Agile 2008 Conference*, pages 289–293. IEEE, Aug 2008.

[28] T. Rausch, W. Hummer, P. Leitner, and S. Schulte. An empirical analysis of build failures in the continuous integration workflows of java-based open-source software. In *Proceedings of the 14th International Conference on Mining Software Repositories*, MSR '17. ACM, 2017.

[29] E. A. Santos and A. Hindle. Judging a commit by its cover: Correlating commit message entropy with build status on travis-ci. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, pages 504–507. ACM, 2016.

[30] C. B. Seaman. Qualitative methods in empirical studies of software engineering. *IEEE Transaction Software Engineering*, 25(4):557–572, 1999.

[31] H. Seo, C. Sadowski, S. Elbaum, E. Aftandilian, and R. Bowdidge. Programmers' build errors: A case study (at google). In *Proceedings of the 36th International Conference on Software Engineering*, ICSE' 14, pages 724–734. ACM, 2014.

[32] E. Shihab, A. E. Hassan, B. Adams, and Z. M. Jiang. An industrial study on the risk of software changes. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 62:1–62:11. ACM, 2012.

[33] J. Singer, S. E. Sim, and T. C. Lethbridge. Software engineering data collection for field studies. In *Guide to Advanced Empirical Software Engineering*, pages 9–34. Springer London, 2008.

[34] E. Smith, R. Loftin, E. Murphy-Hill, C. Bird, and T. Zimmermann. Improving developer participation rates in surveys. In *Proceedings of the 6th International Workshop on Cooperative and Human Aspects of Software Engineering*, CHASE '13, pages 89–92. IEEE, May 2013.

[35] C. Thompson and D. Wagner. A large-scale study of modern code review and security in open source projects. In *Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering*, PROMISE '17, pages 83–92, New York, NY, USA, 2017. ACM.

[36] B. Vasilescu, D. Posnett, B. Ray, M. G. van den Brand, A. Serebrenik, P. Devanbu, and V. Filkov. Gender and tenure diversity in github teams. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, CHI '15, pages 3789–3798, New York, NY, USA, 2015. ACM.

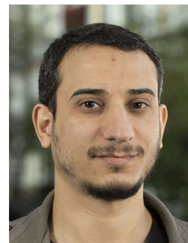
[37] B. Vasilescu, A. Serebrenik, M. Goeminne, and T. Mens. On the variation and specialisation of workload–a case study of the gnome ecosystem community. *Empirical Softw. Engg.*, 19(4):955–1008, 2014.

[38] 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 2015 10th Joint Meeting on Foundations of Software Engineering*, FSE '15, pages 805–816. ACM, 2015.

[39] Y. Yu, B. Vasilescu, H. Wang, V. Filkov, and P. Devanbu. Initial and eventual software quality relating to continuous integration in github. *arXiv preprint arXiv:1606.00521*, 2016.

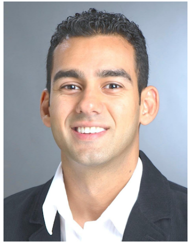
[40] F. Zampetti, S. Scalabrino, R. Oliveto, G. Canfora, and M. Di Penta. How open source projects use static code analysis tools in continuous integration pipelines. In *Proceedings of the 14th working conference on mining software repositories*, MSR '17. ACM, 2017.

[41] C. Ziftci and J. Reardon. Who broke the build? automatically identifying changes that induce test failures in continuous integration at google scale. In *In preceding of 39th International Conference on Software Engineering*, ICSE' 17, Buenos Aires, Argentina, 2017.



Rabe Abdalkareem is a PhD candidate in the Department of Computer Science and Software Engineering at Concordia University, Montreal. His research investigates how the adoption of crowdsourced knowledge affects software development and maintenance. Abdalkareem received his masters in applied computer science from Concordia University. His work has been published at premier venues such as FSE, ICSME and MobileSoft, as well as in major journals such as IEEE Software, EMSE and IST. Contact him at rab_abdu@encs.concordia.ca; http://users.encs.concordia.ca/rababdu.

Suhaib Mujahid is a Ph.D. student in the Department of Computer Science and Software Engineering at Concordia University. He received his masters in Software Engineering from Concordia University (Canada) in 2017, where his work focused on detection and mitigation of permission-related issues facing wearable app developers. He did his Bachelors in Information Systems at Palestine Polytechnic University. His research interests include wearable applications, software quality assurance, mining software repositories and empirical software engineering. You can find more about him at http://users.encs.concordia.ca/smujahi.



Emad Shihab is an associate professor in the Department of Computer Science and Software Engineering at Concordia University. He received his PhD from Queens University. Dr. Shihabs research interests are in Software Quality Assurance, Mining Software Repositories, Technical Debt, Mobile Applications and Software Architecture. He worked as a software research intern at Research In Motion in Waterloo, Ontario and Microsoft Research in Redmond, Washington. Dr. Shihab is a member of the IEEE and ACM. More information can be found at http://das.encs.concordia.ca.

Juergen Rilling is a professor in the Department of Computer Science and Software Engineering at Concordia University, Montreal, Canada. He obtained a Diploma degree in computer science from the University of Reutlingen, a M.Sc. in Computer Science from the University of East Anglia and his Ph.D. from the Illinois Institute of Technology, Chicago, US. The general theme of his research has been on providing software maintainers with techniques and methodologies to support the evolution of global software ecosystems. He also serves on the program committees of numerous international conferences and workshops in the area of software maintenance and program comprehension and as a reviewer for many of the major journals in his research area.