Beyond Traditional Software Development: Studying and Supporting the Role of Reusing Crowdsourced Knowledge in Software Development

Rabe Muftah B. Abdalkareem

A Thesis
in
The Department
of
Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements
For the Degree of
Doctor of Philosophy (Computer Science) at
Concordia University
Montréal, Québec, Canada

February 2019

Concordia University

School of Graduate Studies

This is to certify that the thesis prepared

By: **Mr. Rabe Muftah B. Abdalkareem**

Entitled: **Beyond Traditional Software Development: Studying and Supporting the Role of Reusing Crowdsourced Knowledge in Software Development**

and submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy (Computer Science)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

_____ Chair
Dr. Jun Cai

_____ External Examiner
Dr. Foutse Khomh

_____ Examiner
Dr. Roch Glitho

_____ Examiner
Dr. Yann-Gaël Guéhéneuc

_____ Examiner
Dr. Todd Eavis

_____ Supervisor
Dr. Juergen Rilling

_____ Co-supervisor
Dr. Emad Shihab

Approved by _____
Dr. Volker Haarslev, Graduate Program Director

1 April 2019 _____
Dr. Amir Asif, Dean
Gina Cody School of Engineering and Computer Science

Abstract

Beyond Traditional Software Development: Studying and Supporting the Role of Reusing Crowdsourced Knowledge in Software Development

Rabe Muftah B. Abdalkareem, Ph.D.

Concordia University, 2019

As software development is becoming increasingly complex, developers often need to reuse others' code or knowledge made available online to tackle problems encountered during software development and maintenance. This phenomenon of using others' code or knowledge, often found on online forums, is referred to as crowdsourcing. A good example of crowdsourcing is posting a coding question on the Stack Overflow website and having others contribute code that solves that question. Recently, the phenomenon of crowdsourcing has attracted much attention from researchers and practitioners and recent studies show that crowdsourcing improves productivity and reduces time-to-market. However, like any solution, crowdsourcing brings with it challenges such as quality, maintenance, and even legal issues.

The research presented in this thesis presents the result of a series of large-scale empirical studies involving some of the most popular crowdsourcing platforms such as Stack Overflow, Node Package Manager (*npm*), and Python Package Index (*PyPI*). The focus of these empirical studies is to investigate the role of reusing crowdsourcing knowledge and more particularly crowd code in the software development process.

We first present two empirical studies on the reuse of knowledge from crowdsourcing platforms namely Stack Overflow. We found that reusing knowledge from this crowdsourcing platform has the potential to assist software development practices, specifically through source code reuse. However, relying on such crowdsourced knowledge might also negatively affect the quality of the software projects. Second, we empirically examine the type of development knowledge constructed on crowdsourcing platforms. We examine the use of trivial packages on *npm* and *PyPI* platforms. We found that trivial packages are common and developers tend to use them because they provide them with well tested and implemented code. However, developers are concerned about the maintenance overhead of these trivial packages due to the extra dependencies that trivial packages introduce. Finally, we used the gained knowledge to propose a pragmatic solution to improve the efficiency of relying on the crowd in software development. We proposed a rule-based technique that automatically detects commits that can skip the continuous integration process. We evaluate the performance of the

proposed technique on a dataset of open-source Java projects. Our results show that continuous integration can be used to improve the efficiency of the reused code from crowdsourcing platforms.

Among the findings of this thesis are that the way software is developed has changed dramatically. Developers rely on crowdsourcing to address problems encountered during software development and maintenance. The results presented in this thesis provides new insights on how knowledge from these crowdsourced platforms is reused in software systems and how some of this knowledge can be better integrated into current software development processes and best practices.

Acknowledgments

First and foremost, I would like to thank Allah Almighty for giving me the strength and the ability to finish My Ph.D. thesis.

My sincere gratitude goes to my co-advisors Dr. Emad Shihab and Dr. Juergen Rilling, as without their guidance, instruction, support, enlightenment, and patience, this research would not have been completed. Their unique personalities, as supervisors and friends have allowed me to become a successful researcher.

I want to thank my committee members, Drs. Foutse Khomh, Roch Glitho, Yann-Gaël Guéhéneuc, and Todd Eavis for taking the time to read and critique my work and for their valuable suggestions and comments. I would also like to extend my thanks to Drs. Weiyi Shang and Nikolaos Tsantalis for providing me with insightful feedback.

I have also had the privilege to collaborate and discuss my research with great researchers. I want to thank Drs. Ahmed Hassan, Xin Xia, Shane McIntosh, and Alexander Serebrenik for sharing their insights and advice.

My appreciation extends to my colleagues and friends at the DAS lab and at Concordia: Suhaib Mujahid, Sultan Alqahtani, Mohamed Elshafei, Ahmad Abdellatif, Mahmoud Alfadel, Ellis Eghan, Hosein Nourani, Atique Reza, Behzad Dehghani, Xiaowei Chen, Mouafak Mkhallalati, Everton Maldonado, Sultan Wehaibi, Moiz Arif, Olivier Noury, and Giancarlo Sierra Monge.

Finally, this thesis would not be possible without the continuous care from my parents and family, and the support of my friends. My mother and father, without your constant prayers, encouragement, and love, I would not have been able to continue my Ph.D. journey. And, to my beloved wife, who has stood by my side throughout this journey.

Dedication

To my wife and children, Mohamed, Alaa, and Aya.

Contents

List of Figures

List of Tables

Chapter 1

Introduction and Problem Statement

While software engineering has changed tremendously over the years through the adoption of new technologies and practices, its fundamental premise to deliver high-quality software in a timely and cost-efficient manner has remained the same. At the same time, the emergence of Web 2.0 has provided software developers with the ability to collectively share their software development and programming experience in online communities. The software engineering community has not only adapted but also taken advantage of these newly available technologies in order to achieve the objectives of software engineering by introducing new software development paradigm such as open source [199] and collaborative software development [24].

Crowdsourcing, the process of outsourcing software development tasks to developers in online community platforms, is one example of open source development [103, 181]. It has become a common approach in software development, with development tasks being decomposed in smaller, well-defined tasks that can be completed by the crowd [122, 108]. Crowdsourcing in software engineering has led to the success of many forms, including open source systems' development (Linux and Apache) [108]. There are also multiple crowdsourcing platforms that cover coding (e.g,. TopCoder [2]), testing (e.g,.uTest [6]), and code reuse (e.g,. npm [4]), as well as some general software development tasks (e.g,. Stack Overflow [3]).

Although recent, the phenomenon of crowdsourcing has attracted much attention from both researchers and practitioners. And for a good reason. Recent studies have shown that crowdsourcing improves productivity, reducing time-to-market [107, 180]. However, like any solution, crowdsourcing brings with it many challenges such as quality issues, scalability and performance issues, maintenance issues and even legal issues. For example, one common concern with crowdsourcing is related to the fact that it is hard to ensure developers participate enough and submit good quality work and, therefore, potentially can impact the quality of the final product [107, 180].

In this thesis, we aim to build a solid understanding of how crowdsourcing can help developers build high-quality software. To achieve this goal, we believe that we first need to address the following problems: 1) understand how developers use crowdsourcing platforms, 2) once developers use the crowd knowledge constructed on these platforms, how this knowledge can impact the software quality, and 3) understand what the type of knowledge is constructed on these crowdsourcing platforms. Finally, use the gained knowledge to improve the efficiency of the reused crowdsourcing knowledge in the software development process.

1.1 Motivating Examples

Crowdsourcing is increasingly revolutionizing the ways how software is engineered [181]. Both, the open source community and private software companies have adopted the reuse of crowdsourced knowledge (i.e., source code) by outsourcing development tasks to the crowd. A good example of crowdsourcing is posting a coding question on Stack Overflow and having others contribute code to solve your posted code problem. Such form of knowledge sharing and reuse are the main focus of this thesis. To shed light on issues that can arise when reusing crowdsourced knowledge, we illustrate these problems involving two well-known crowdsourcing platforms and how developers use them.

One example of a commonly used crowdsourcing platform that developers rely on is Stack Overflow. Stack Overflow is a general crowdsourcing platform that programmers use to benefit from the crowd's programming knowledge. Developers share on Stack Overflow their knowledge, provide development support, learn new technologies, and search for solutions to both common and specific programming problems [179, 187]. Also, research reported that Stack Overflow has become the largest non-traditional source code repository that developers resort to for code reuse [12, 19]. In fact, the reuse of source code snippets posted on Stack Overflow is reusing crowdsourced knowledge. Thus, the quality of reusing crowdsourced code snippets posted on Stack Overflow is an important concern. Yet little is known about these code snippets (e.g,. the level of experience of the developer who wrote the code, the quality of these code fragments, and if these code fragments are original or cloned from another software systems). In addition, source code published on Stack Overflow is often only for illustrative purpose and may exclude important implementation aspects associated with complete or well-written code (e.g,. no error handling, non-secure programming style, or outdated coding styles/libraries).

A key challenge associated with this type of code reuse is that programmers often resort to these code snippets in an ad-hoc manner, simply copying-and-pasting these fragments into their own software system without thoroughly testing the copied code [12]. Additionally, developers often adopt code from crowdsourcing platforms without completely understanding and fully testing the copied

code [36]. As a result, such type of reuse can lead to quality issues in software systems which can cause additional future maintenance effort. Prior to assessing the impact of code/knowledge reused from Stack Overflow on current software projects, one has first to detect reused code. Detecting reused source code from Stack Overflow in a software system is a challenging task since: 1) it is a new problem with no previous work on how to identify reused knowledge from Stack Overflow; and 2) programmers who reuse source code from Stack Overflow occasionally perform obfuscation operations (e.g,. renaming variables, adding and removing code) on the reused code. These modifications are sometimes done unintentionally to meet the system's new context and quality (e.g., programming style) or deliberately by some programmers to hide the reused code, making it difficult to detect the origin of the code to avoid source code licensing or ownership issues.

The second example is to illustrate code reuse from other crowdsourcing resources such as the Node Package Management (*npm*) [4]. *npm* is a community based online registry that provides tools for managing packages of JavaScript code and their revisions. The goal of the Node.js/*npm* community is to make it easy and fast for developers to publish and use packages. In fact, the act of developing and publishing a package on *npm* is a form of crowdsourcing task that the package provide [181]. However, the Node.js/*npm* platform has a somewhat unusual characteristic. Node.js is a new JavaScript platform which inherited the relative poorness of JavaScript's standard library [204], that developers provide, such as a missing JavaScript functionalities in a crowdsourcing call. Packages published on *npm* cover different development tasks, ranging from simple tasks (e.g., identifying a character as a number) to more complicated tasks (e.g., building a testing framework).

Recent studies show that using *npm* as a package management system introduces maintenance overhead [32]. For example, multiple revisions of a package can coexist within the same project. A developer may use two packages that each require a different revision of a third package. In that case, *npm* will install both revisions in distinct places and each package will use a different implementation. Another example, a simple spelling error in a new update of a *npm* package called debug from version 2.3.3 to 2.4.0 led to the crash of approximately 24 thousand direct dependent packages that tried to download the new version [127]. Even though the bug was fixed within an hour, this bug impacted the installations of thousands of other packages in the *npm* platform. While the reuse of such packages tends to reduce development effort and cost [22, 114], this type of reuse might increase both system complexity and future maintenance efforts [10]

These examples show how strongly and drastically crowdsourcing platforms are affecting the software development process. Therefore, we believe that as these platforms continue to play an increasingly important role in our software systems, understanding and effectively using them is of critical importance. More specifically, in this thesis, we uncover insights about how developers use the crowdsourcing knowledge and the type of knowledge constructed from some of the most known

crowdsourcing platforms. We also examine the use of continuous integration process to improve the efficiency of such reuse.

1.2 Problem Statement

Recent studies have shown that crowdsourcing improves productivity, reducing time-to-market. However, like any solution, crowdsourcing also introduces new challenges such as quality issues, potential scalability and performance problems, maintenance and even legal issues. These concerns and motivations led to the formulation of this thesis problem statement, which is stated as follows:

With the popularity of crowdsourcing platforms and given the fact that software developers use crowdsourced knowledge (e.g., source code and development knowledge), we hypothesize that the adoption of crowdsourcing in the development and maintenance process has an impact on the quality and maintainability of software systems. We conduct empirical studies to gain insights on the actual use of such crowdsourced knowledge and use the findings from these studies to improve the use of crowdsourcing knowledge in software engineering.

1.3 Thesis Overview

In this section, we provide a brief overview of the thesis. The thesis consists of six chapters, which can be classified into three main parts. In the first part (Chapter 2), we provide background and related work to the thesis. In the second part (Chapters 3, 4, & 5), we present three empirical studies related to the understanding of the use of crowdsourcing knowledge in software engineering. The last part (Chapter 6), presents our contribution to improving the efficiency of reusing crowdsourcing knowledge. Finally, Chapter 7 concludes the thesis and discusses avenues for future work.

Chapter 2: Background and Related Work.

This chapter contains two main sections. The first section presents a broad background of the concept of crowdsourcing and its definition used in software engineering. The second section presents existing work that is related to this thesis. In particular, we review research on integrating crowdsourced knowledge in the software development processes and how it is used to provide recommendation systems. We also discussed work related to existing studies, which have been focusing on Stack Overflow as a crowdsourced platform. In section 2.3, we review research on traditional code reuse

that is also applicable for code reuse from crowdsourcing platform such as Stack Overflow. This includes works related to code search from the web, source code recommendation, and studies related to cloning of source code.

Chapter 3: Understanding the Usage of Crowdsourced Knowledge.

General crowdsourcing platforms such as Stack Overflow, heavily rely on contributions of the crowd to provide accumulated, quality knowledge to the software development community. Typically, users post questions on Stack Overflow, which are answered by one or (often) more participants. In essence, the job of answering the questions is outsourced to the crowd [108]. However, the role of Stack Overflow has evolved to include much more than just answering questions. For example, Treude *et al.* [187] qualitatively analyzed a sample of Stack Overflow questions and found that developers use Stack Overflow to share knowledge, provide development support, learn new technologies, and search for solutions to common and specific programming problems. However, aside from this small qualitative study, almost no prior work exists that studied the reasons why developers use the crowdsourced knowledge on Stack Overflow for. Thus, we investigate to understand how developers actually use crowdsourced knowledge. More specifically, this research aims to answer questions such as "What crowdsourced knowledge do developers actually use?" and "What types of this knowledge are most helpful to developers?"

In Chapter 3, we performed one of the first in-depth studies to gain insights on how developers actually use crowdsourced knowledge. We introduce a novel approach in contrast to existing studies and explored in which situations developers use unstructured crowdsourcing knowledge available on Stack Overflow. For our study, rather than mining Stack Overflow content, we mined GitHub commit histories looking for explicit mentions of Stack Overflow in their commit changes. We found that developers most often use Stack Overflow to gain knowledge related to topics such as development tools, APIs usage, and operating systems. More importantly, we found that developers use Stack Overflow to provide a rationale for feature updates or additions. In fact, we are the first to discover that developers even use Stack Overflow as a communication channel to receive user feedback about their software. This work was published in a highly competitive IEEE Software special issue on crowdsourcing in Software Engineering and was featured in an IEEE Software blog [13, 93].

Chapter 4: The Impact of Reused Source Code from Crowdsourcing Platforms.

Most popular crowdsourcing platforms such as Stack Overflow provide developers with source code that they can reuse [108, 13]. However, very little work exists on the actual code reuse from these

crowdsourcing platforms. Furthermore, detecting reused source code from these platforms is a challenging task, as these crowdsourcing platforms are not designed to track the reused source code. Moreover, being able to trace such code reuse from crowdsourcing platform such as Stack Overflow can provide new insights on: 1) How much source code from Stack Overflow is reused in software systems; 2) Why do developers reuse code from Stack Overflow; 3) When in a project's lifetime do developers reuse code from Stack Overflow; and 4) Who reuses code from Stack Overflow? In addition, having such insights will let us further examine the impact of reusing source code from crowdsourcing platforms on software quality and maintenance of systems, which reuse such crowdsourced code.

In Chapter 4, we propose a systematic detection technique to automatically identify reused crowdsourced code in a software system. Specifically, we focus on detecting reused code from Stack Overflow. Then, we perform a study focusing on code reuse from Stack Overflow in the context of mobile apps to investigate how much, why, when, and who reuses code. Moreover, to understand the potential implications of code reuse, we examine the percentage of bugs in files that reuse Stack Overflow code. We perform a study on 22 open source Android apps. For each app, we mine their source code and use clone detection techniques to identify code that is reused from Stack Overflow. We then apply different quantitative and qualitative methods to answer our research questions. Our findings indicate that 1) the amount of reused Stack Overflow code varies for different mobile apps, 2) feature additions and enhancements in apps are the main reasons for code reuse from Stack Overflow, 3) mid-age and older apps reuse Stack Overflow code mostly later on in their project lifetime and 4) that in smaller teams/apps, more experienced developers reuse code, whereas, in larger teams/apps, the less experienced developers reuse code the most. Additionally, we observed that the percentage of bugs is higher in files after reusing code from Stack Overflow. The results of this research appear in the Journal of Information and Software Technology [12].

Chapter 5: Examining the Type of Constructed Knowledge on Crowdsourcing Platforms.

Crowdsourcing has evolved to become an important practice of software development, shaped by the ability to collectively share software development and programming experiences in online code-sharing repositories such as GitHub. Crowdsourcing in software engineering has led to success in many forms, including the Node Package Management (*npm*). However, it is not always good news, in a recent incident code reuse of a very simple Node.js package called left-pad, which was used by another well-known package called Babel, caused interruptions to some of the largest Internet sites, e.g., Facebook, Netflix, and Airbnb [118, 200]. Many referred to the incident as the case that almost broke the Internet. The dependency that caused Babel to break was on a trivial 11 line package

that implements a left padding string function. Although the issue of using trivial packages has been discussed, most of these discussions have been informal. Basic research questions, such as "What is considered to be a trivial package?" and more important questions about why developers resort to such trivial packages and whether they perceive such reuse to have any drawbacks remain as open questions. To answer these research questions, we conducted one of the largest empirical software engineering studies in this domain to gain a more solid understanding as to why developers may take on additional dependencies to implement trivial tasks.

We first define a trivial package as a package that contains code that a developer can easily code him/herself and hence is not worth taking on an extra dependency for. Then, we employed qualitative and quantitative research methods to examine the prevalence, reasons, and drawbacks of using trivial packages. We performed a large survey involving 125 JavaScript and Python developers who use trivial packages. As part of this study, we mined more than 230,000 *npm* and 63,000 *PyPI* packages and 38,000 JavaScript and 14,000 Python projects. We found that trivial packages are commonly and widely used in JavaScript and Python projects. We also observed that, contrary to JavaScript developers beliefs that trivial packages are well-tested, only 45.2% of trivial packages that we mined actually have tests code. Additionally, we found that developers believe that using trivial packages tends to increase the dependency overhead of their software. More detailed analysis also showed that some of the trivial packages have their own dependencies. For example, 11.5% of the trivial packages have more than 20 dependencies. Hence, this work was the first to show that: 1) these trivial packages are commonly used and the left-pad incident was not isolated, 2) one cannot assume these trivial packages are well-tested, and 3) some of these trivial packages can bring more pain than they are worth, since some have as many as 20 dependencies. This work was published in the proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering [10].

Chapter 6: Improving the Efficiency of Reused Crowdsourced Knowledge.

So far, the previous chapters of this thesis focus on understanding how crowdsourcing knowledge is reused by developers and its impact on software systems. Our findings show that software quality is one of the main concerns of adopting crowdsourcing development in software engineering [10, 12].

While for traditional software development, a number of techniques have been used to ensure the quality of a software system, including testing [62], code reviews [160], and continuous integrations [195], little work exists that specifically investigates the applicability of these techniques for such crowdsourced code. In fact, there are two main ways to help developers ensure the quality of the reused crowdsourcing knowledge. First, through the use of tradition quality assurance techniques e.i., code reviews and testing. Second, helping developers to focus on the main task through removing extra noise that developers are required to handle during the integration of reused crowdsourcing

knowledge. In Chapter 6, we propose a technique to help developers eliminate the distraction cause by the noise such as reviewing cosmetic changes.Our objective is to gain an understanding to what extent existing quality assurance techniques can be improve to help developers remove the extra noise and focus on the main development task. Being able to reduce these challenges will help to promote further and encourage such type of code and knowledge reuse.

To do so, we developed a technique to improve the efficiency of the continuous integration (CI) process. Our main argument is that not every commit to the repository needs to trigger the CI process. For instance, developers may modify a project's documentation and cause 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. We proposed and evaluated a rule-based technique based on manual analysis of more than 1,800 explicitly CI skip commits. Our evaluation showed that the devised rule-based technique is able to detect and label CI skip commits with an average Areas Under the Curve (AUC) of 0.73. Our results provide insights into the potential importance of modifying CI techniques to ensure the quality of code reuse from crowdsourcing platforms. A manuscript based on this work has been accepted to IEEE Transactions on Software Engineering (TSE) [9].

1.4 Thesis Contributions

The main contributions of the thesis are:

1. We show that developers most often use knowledge from crowdsourcing platforms such as Stack Overflow to gain knowledge related to topics such as development tools, APIs usage, and operating systems.

2. We discover that developers use Stack Overflow as a communication channel to receive user feedback about their software in addition to the usual uses.

3. We propose an approach to trace reused source code form crowdsourcing platforms, namely Stack Overflow.

4. We show that there is a relationship between reused source code from Stack Overflow and software defects.

5. We provide a systematic approach to define and identify trivial packages in software engineering. Also, we created the term Trivial Packages, which is now commonly used within the research community.

6. We show that using trivial packages could increase the maintenance effort of software projects since we show that some of the trivial packages have their own dependencies. For example, 11.5% of the trivial packages have more than 20 dependencies.

7. We developed a technique to improve the efficiency of the Continuous Integration (CI) process through skipping commits that do not affect the source code.

1.5 Related Publications

Earlier versions of the work presented in this thesis has been previously published or submitted to different venues include the following ones:

- **R. Abdalkareem**, E. Shihab, and J. Rilling, "What Do Developers Use the Crowd For? A Study Using Stack Overflow", *Special Issue on Crowdsourcing for Software Engineering, IEEE Software*, 34 (2), pages 53-60 (2017).

- **R. Abdalkareem**, E. Shihab, and J. Rilling, "On code Reuse from Stack Overflow: An Exploratory Study on Android apps", *Information and Software Technology (IST)*, 88, pages 148-158 (2017).

- **R. Abdalkareem**, O. Nourry, S.Wehaibi, S. Mujahid, and E. Shihab, "Why Do Developers Use Trivial Packages? An Empirical Case Study on npm", *In Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE, pages 385-395 (2017), ACM, Acceptance rate: 24%

- **R. Abdalkareem**, "Reasons and Drawbacks of Using Trivial npm Packages: The Developers' Perspective", *In Proceedings of the 11th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering - Student Research Competition Track*, ESEC/FSE, pages 1062-1064 (2017), ACM.

- **R. Abdalkareem**, V. Oda, S. Mujahid, and E. Shihab, "On the Impact of Using Trivial Packages: An Empirical Case Study on npm and PyPI", *Springer's Journal of Empirical Software Engineering (EMSE)*, Major Revision Recommended, 37 pages (2018).

- **R. Abdalkareem**, S. Mujahid, E. Shihab, and J. Rilling, "Which Commits Can Be CI Skipped?", *Accepted in IEEE Transactions on Software Engineering (TSE)*, 17 pages, (2019).

In addition, the following work has been published during my Ph.D. program. These papers are relevant to my research [119, 139, 140, 138]. However they do not form part of the thesis content.

- S. Mujahid, G. Sierra, **R. Abdalkareem**, E. Shihab, and W. Shang, "An Empirical Study of Android Wear User Complaints", *Accepted in Empirical Software Engineering (EMSE)*, 24 pages, March, (2018).

- S. Mujahid, **R. Abdalkareem**, E. Shihab, "Studying Permission Related Issues in Android Wearable Apps", *Proceedings of 34th IEEE International Conference on Software Maintenance and Evolution*, ICSME 2018, 10 pages, Acceptance rate: 25%.

- E. Maldonado, **R. Abdalkareem**, E. Shihab, A. Serebrenik, "An Empirical Study On the Removal of Self-Admitted Technical Debt", *Proceedings of the 33rd IEEE International Conference on Software Maintenance and Evolution*, ICSME 2017, pages 238-248, Acceptance rate: 28%.

- S. Mujahid, G. Sierra, **R. Abdalkareem**, E. Shihab, W. Shang, "Examining User Complaints of Wearable Apps: A Case Study on Android Wear", *Proceedings of 4th IEEE/ACM International Conference on Mobile Software Engineering and Systems*, MOBILESoft 2017, pages 96-99 (2017).

1.6 Thesis Organization

The thesis consists of six chapters that are organized as follows: Chapter 2 provides necessary background related to the crowdsourcing in software engineering and research related to our analysis of crowdsourcing platforms. Chapters 3, 4, and 5 present the results of three empirical studies related to the usage and the impact of using crowdsourcing knowledge in software development. In Chapter 6, we present our results of improving the continuous integration service as away to ensure the quality of reused crowdsourcing knowledge. Finally, Chapter 7 summarized the thesis and discussed some direction of future work.

Chapter 2

Background and Related Work

In this chapter, we provide an overview of relevant background knowledge to our research that includes a brief description of crowdsourcing in software engineering including research on integrating crowd-based knowledge in the software development processes and how this knowledge is used to provide recommendation systems. In Section 2.2 we introduce existing studies on crowdsourcing platforms, namely Stack Overflow. In section 2.3, we review research on traditional code reuse that is also applicable for code reuse from crowdsourcing platforms such as Stack Overflow. This includes work related to code search from the web, source code recommendation, and studies related to source code cloning.

2.1 Crowdsourcing Software Development

The term crowdsourcing refers to the process of taking simple tasks or problems that once were performed internally by a company's employees and outsourcing them to an undefined group of people in the form of an open call [90]. In general, crowdsourcing platforms leverage the collective intelligence of users who contribute, evaluate or rank items in communities online such as Facebook, Wikipedia, Stack Overflow and others. The benefits of crowdsourcing include easy (and sometimes free) access to a wide range of workers, various solutions, lower labor rates, and reduced time-to-market [174, 100, 106]. Crowdsourcing has been used in a wide variety of domains, such as weather forecasting [41], information retrieval [14, 109] and software engineering [122, 180, 169].

In software engineering, the value of crowdsourcing has been recognized through the success of open source software development [103]. Mao *et al.* [122] defined crowdsourcing for software engineering as *"Crowdsourced Software Engineering is the act of undertaking any external software engineering tasks by an undefined, potentially large group of online workers in an open call format."* Based on this definition, Figure 1 illustrates the process of publishing a software engineering task

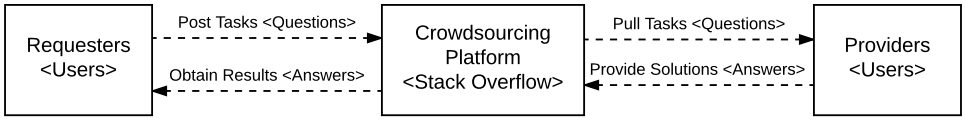


Figure 1: Illustration of interaction with a crowdsourcing platform, Stack Overflow as an example.

to a crowdsourcing platform (e.g., Stack Overflow). The crowd has increasingly used such platforms to answer developers' questions through Q&A sites such as Stack Overflow. Furthermore, crowdsourcing platforms almost cover all aspects of software development such as implementation, requirements, and testing. In their work, Latoza and Hoek [108] defined three models of crowdsourcing platforms based on a number of dimensions. Table 1 describes these three crowdsourcing models: peer production, competition, and micro-tasking model. The peer production model is suited where the idea of gaining experience mainly motivates contributors, such platforms are Stack Overflow and *npm*. The main characteristic of this model is that contributors are in an inactive mode in that they earn no direct benefit. The competitions model is defined when customers publish a task in a competitive way. Its main feature is that the task is complete. Finally, the micro-tasking model is where the crowd contributes to perform and to perfect small tasks.

The emerging number of crowdsourcing platforms that are used in the software development process leads researchers to explored approaches to crowdsourcing software development tasks [122]. Most of this work focuses on examining the use of crowd knowledge in software development tasks rather than understanding its use and impact such as questions & answering programming problems, software verification, testing, or UI design. For example, studies have shown that developers often rely on the crowd knowledge offered on Stack Overflow to find solutions to their to their programming problems (see section 2.2 for an extensive literature review). Crowdsourcing has also been proposed to improve different aspects of testing, such as that include better software tests and mutants [161], performance testing [142], mobile apps testing [123], and GUI testing [197]. Several approaches have been proposed to explore the possibility of using the crowd to recommend bug fixing [141] or finding bug through a crowd-based debugging [82]. Other researchers have examined approaches for crowd-based requirements engineering that aim to increase and improve the involvement of end users in defining the software they use [76]. In this section, we only discuss the most relevant studies; however, we refer the reader to a recent survey by Mao *et al.* [122] for a more comprehensive list of studies on crowdsourcing in software engineering.

Table 1: The Dimensions of Crowdsourcing and Concrete Examples of Three Crowdsourcing Models [108].

Dimension	**Explanation**	**Crowdsourcing Model and Example**		
		Peer Production (Stack Overflow & *npm*)	Competitions (TopCoder)	Microtasking (UserTesting)
Crowd size	The size of the crowd necessary to effectively tackle the problem	Large	Small	Medium
Task length	The amount of time a worker spends completing a task	Hours to days	Week	Minutes
Expertise demands	The level of domain familiarity required for a worker to contribute	Moderate	Extensive	Minimal
Locus of control	Ownership of the creation of tasks or subtasks	Worker	Client	Client
Incentives	The factors motivating workers to engage with the task	Intrinsic	Extrinsic	Extrinsic
Task interdependence	The degree to which tasks in the overall workflow build on each other	Medium	Low	Low
Task context	The amount of system information a worker must know to contribute	Extensive	Minimal	None
Replication	The number of times the same task might be redundantly completed	None	Several	Many

2.2 Work Related to Stack Overflow as a Crowdsourcing Platform

The wide-spread use of Stack Overflow among software developers has increased the interest of the software community in this domain, which is reflected by the number of publications using Stack Overflow data in their research [191]. This Stack Overflow related research includes investigating the possibility of integrating Stack Overflow in the software development process and the characteristics of Stack Overflow as Q&A website. In this section, we divided this related work into work on harnessing Stack Overflow data and work that focuses on analyzing Stack Overflow data.

Cordeiro *et al.* [44] indexed the Stack Overflow data dump to retrieve associated discussions in the context of exception stack traces. The main propose of this research is to assist developers to understand failure that occurred in their code. Their Eclipse plugin extracts keywords from exception traces and automatically generates a request in order to retrieve the most likely related posts from Stack Overflow. The results of their preliminary evaluation showed that their exception

stack trace approach outperforms a simple keyword-based approach.

Ponzanelli *et al.* [149, 150] devised an Eclipse plugin called SeaHawk that uses knowledge extracted from Stack Overflow to support programming and documentation. The plug-in automatically formulates queries by extracting keywords from code entities from a user's current work context. The generated queries run against an indexed Stack Overflow data dump to retrieve related discussions. Additionally, SeaHawk attempts to provide support for exploiting Stack Overflow data in a team-setting environment [15]. SeaHawk allows developers to create links between source code and Stack Overflow discussions via annotations in the source code. They conducted three experiments to evaluate 1) the relevance of the retrieved Stack Overflow discussions to the programming task, 2) the impact of SeaHawk when dealing with programming tasks for the first time, and 3) the help provided by SeaHawk to understand an existing and fully implemented method. Despite the fact that their tool might not suggest useful discussions, the results show that SeaHawk can assist developers during program comprehension and software development [149, 148].

Example Overflow is a code search that helps programmers in using Q&A websites. Zagalsky *et al.* [206, 20] extracted code snippets from the accepted answers, including some metadata (e.g., title, tag, answer) in Stack Overflow via its public API [1]. The search function is based on indexed data (Stack Overflow) using Apache Lucene, and the output search is ranked based on the metadata of the original posts. They empirically evaluate their proposed code search by choosing ten programming tasks, and compared the results with other search engines (e.g., Google and Koders). Their results indicated that Example Overflow suggested better results than other existing search engines covered in their experiments.

Rahman *et al.* [157], proposed SurfClipse a tool that provides immediate assistance to developers when they encounter runtime errors or exceptions. It exploits three search engines and Stack Overflow, considering the exception trace and the associated source code during the search. In their evaluation, they found that the inclusion of different types of contextual information associated with an exception can enhance the accuracy of their recommendations.

Ponzanelli *et al.* [151, 152, 153] developed a tool called Prompter, which searches Stack Overflow to recommend related discussions for the developers. Prompter generates a search query from the developers' programming context in the IDE using Stack Overflow's public API [1] to recommend code snippets from Stack Overflow discussions. The recommended discussions are not only based on their text similarities as in [149], also consider other information resources related to Stack Overflow discussions, such as users reputation, questions/answers score and tags. The results of their evaluation showed that Prompter is able to effectively suggest the right Stack Overflow discussions based on a given a code snippet. Additionally, in a control experiment, the results showed that Prompter can help developers during software development and maintenance.

AutoComment presented by Wong *et al.* [202] is an automatic comment generation approach based on natural language text available on Stack Overflow. AutoComment extracts code snippets together with their descriptions, from Stack Overflow, and leverages these descriptions to automatically generate comments for similar code segments in open source projects. The authors conducted a user study to evaluate their proposed approach by applying AutoComment to Java and Android projects in which they automatically generated 102 comments for 23 projects. It was found that the generated comments are accurate, adequate, concise, and helpful in helping developers to understand the code.

Rahman *et al.* [156] proposed an approach to mine users' comments about source code snippets published on Stack Overflow in order to document deficiencies and quality of the code. Their approach uses information retrieval techniques to find informative users' comments about the code snippets. This method was applied on 292 Stack Overflow code snippets with their discussions. The results showed that the proposed approach can extract insightful comments with 85.42% recall. In addition, a user study showed that about 80% of the suggested comments about the code were informative and useful.

De Souza *et al.* [51] developed a search engine that focuses on "how to do it" discussions from Stack Overflow. The ranking criteria used by their approach consists of 1) the textual similarity of the question-and-answer pairs to the query and 2) the quality of these pairs. Their evaluation showed that the approach was able to recommend at least one useful question-and-answer pair which included a reproducible code snippet.

Wang et al. [198], presented an approach to build a bidirectional link between the Android issue tracker and Stack Overflow discussions to facilitate the knowledge sharing between two separated communities. They exploit three techniques to measure similarity (Vector Space Model(VSM), Latent Semantic Indexing(LSI) and Latent Dirichlet Allocation(LDA)) and use temporal-locality method to effectively establish the links. Their evaluation showed that Vector Space Model with temporal-locality outperform the other techniques.

All the aforementioned research has the goal to incorporate knowledge and source code found on Stack Overflow in the software development and maintenance process. Common to this reviewed work is that they provide evidence that crowdsourcing platforms can help developers during learning and improving their programming skills, so as to reuse source code snippets from crowdsourcing platforms [175, 166]. However, none of this work has studied the implication of using crowd knowledge on the software quality. For example, examine the impact of code reuse from crowdsourcing platforms (e.g., Stack Overflow).

One of the main goals of this thesis is to provide empirical evidences on the actual impact of reusing source code snippets from crowdsourcing platforms on software quality.

2.3 Work Related to Source Code Reuse

Research on source code reuse has become an important part of program comprehension and software maintenance research. In fact, reusing source code from the crowd is in its objective similar to traditional source code reuse. Thus, in this section, we briefly review both traditional code reuse within software systems and code reuse from the Web. We discuss first empirical studies covering code search on the Web, then using the Web to recommend source code, finally, review some work related to copy and paste source code.

2.3.1 Empirical Studies of Code Search on the Web

A number of studies exist that focus on understanding the role of source code search in the software development process. An early study conducted by Sim *et al.* [175], reports on a user survey conducted with programmers to classify reasons why developers resort to online source code search during programming and maintenance tasks. Their study reports that the most common search queries were related to: function definitions, different uses of a function, variable definitions, and different uses of a variable. Furthermore, they observed that most search queries were performed to support defect repair, code reuse, program comprehension, feature addition, and impact analysis. Brandt *et al.* [36] empirically examined how developers use the Web during programming tasks. In their study they observed the behavior of 20 programmers and analyzed Web query logs from 24,293 programmers. Their findings showed that programmers use the Web to quickly learn technologies, looking for clarification, and to recall a certain programming languages syntax. In addition, they reported that developers do not immediately test copied code from the Web. In [70], an experimental study conducted by Gallardo-Valencia and Sim to investigate the types of problems which motivate developers to do searches on the Web. In their user study, they observed 24 developers in an industrial setting. They reported that around 82% of Web searches are performed in an ad-hock manner also known as opportunistic search. This type of Web search is done to 1) remember syntax details, 2) clarify implementation details including fixing a bug, 3) learning new concepts. Additionally, 18% percent of the Web search conducted to find open source projects or code snippets. Sadowski *et al.* [166] studied programmers' behavior when searching for code at Google Inc®. They analyzed survey results. They found that developers at Google search for code very frequently with an average of five session everyday. Furthermore, developers preform code search for different development tasks, including code review, finding a code snippet, and retrieving programming knowledge.

These empirical studies show that software developers frequently resort to code search during programming tasks. However, there are several common threats to validity in these studies, including that the results are based on observing or surveying a limited number of human subjects and that

analysis was conducted whether the code from the crowdsourcing platforms was actually reused in software systems.

2.3.2 Source Code Recommendation from the Web

Empirical studies in software engineering [175, 36] reported that developers often search the Web for example or code snippets. Numerous approaches have been proposed in the literature to support developers during programming tasks by providing code samples they can reuse [88, 186, 89].

To help programmers effectively benefit from the Web search engine, Hoffmann *et al.* [88] developed Assieme, a Web search interface that combine information from Web-accessible Java Archive files, API documentation and Web pages that include sample code snippets. It uses the Google search engine to find relevant Web pages containing source code. It then parsed the Web pages to create source code databased, and uses text on Web pages to rank code snippets and the libraries that are referenced in that code. Mica is a tool that augments the Web search to find code examples and API documentations [182]. It eliminates irrelevant results from Google Web APIs, and highlights the source code elements. PARSEWeb [186] is a tool that assists programmers in using APIs. It uses online search to recommend relevant method call sequences by applying static analysis on code snippets which return from a search engine. Other approaches aim to recommend code elements or code examples by mining source code in software repositories. For example, Holmes *et al.* [89] used structure matching from source code in Eclipse IDE to retrieve the most similar code snippets. The approach combines three heuristics to measure similarity between code snippets; 1) inheritance similarity that finds classes with the same parent. 2) method call similarity that retrieve methods with similar call-graph. 3) a use heuristic that find methods using similar data types. Keivanloo *et al.* [102] proposed a reusable code recommendation by ranking high quality code snippets. Their approach measured the code snippet's similarity to a query by combining textual similarity and clone detection techniques. The recommended code snippets are ranked according to completeness and popularity of their usage patterns. More recently, Moreno *et al.* [137] presented MUSE, a technique for mining and ranking actual code examples that show how to use a specific method. Their approach parses existing applications to collect method usages pattern by static code analysis. A clone detection tool is used to group similar code example and MUSE ranks them according to their popularity. MUSE provides users with one code snippets from each group of clone that it selected based on its reusability, understandability, and popularity.

The aforementioned examples focus on recommending source code snippets to developers for learning API usage or source code reuse. Even though these approaches are appropriate for assisting developers in programming tasks, they lack information about the quality of the recommended source code. In addition, developers may reuse the suggested code snippets in a new context without

completely understanding the functionality of the code. One of the main aims of this thesis is to study the impact of integrating reused code from crowdsourcing platforms on the quality of the target software system.

2.3.3 Empirical Studies of Copy and Paste Source Code

Using source code from crowdsourcing platforms sometimes involves code cloning (copy and paste) and detecting code reuse. Research done in this area may also, in some cases, be relevant for studying code reuse form crowdsourcing platforms.

One of the most common code smells in software systems is cloning [104, 164] , copying code fragments and then reuse by pasting with or without minor modifications. Cloning can occur within the same project, or across projects as part of source code reuse. However, research in the area of code cloning lacks the consensus about its impact of clones on the quality of source code. For example, Kapser and Godfrey [99] described several clone patterns, and found that clones have often a positive effect on open source systems. Rahman *et al.* [155] empirically studied the relationship between code clones and defect proneness, and found that the majority of defects are not significantly associated with code clones. On the other hand, there are studies that discuss the negative effect of code clone on source code quality. Juergens *et al.* [94] investigated the occurrence of clones in open and proprietary systems. They found that inconsistent clones introduce defects that are difficult to detect. Lozano and Wermelinger [117] reported that code clones increase the maintenance efforts when compared to non-cloned code. Mondal *et al.* [136] linked the type of clones (Type-1, Typ2 and Type3) to defect proneness in open source systems, and found Type-3 is the most related to bug fixing change.

Most of the studies focus on the clone existence within the same software. However, developers frequently copy and paste code form outside resource such as Q&A websites [70], and little effort has been devoted to study the impact of such copy and paste action. For example, source code posted on Stack Overflow may not be complete or can contain vulnerabilities.

Other research works focus on the detection and study of code cross software systems boundaries. Inoue *et al.* [92] developed a prototype called Ichi Tracker to explore the evolution of a code fragments utilizing online code search engines and code clone detection techniques. The tool accepts a code fragment as an input and returns related files containing. Using the Ichi Tracker, developers can identify the origin of a source code fragment or a modified version of the code fragment, including potential license violations. German et al. [72] examined source code migration across three different systems (Linux, FreeBSD and OpenBSD) and its legal implications. They tracked reused code fragments using clone detection methods. Their result showed that code migration occurred between these systems. Additionally, the copying tends to be preformed without violating the license terms.

Davies et al. [48] proposed a signature-based matching technique to determine the origin of code entities. They found that their technique can be utilized to identify security bugs in reused libraries. Kawamitsu et al. [101] proposed a technique to automatically detect source code reuse between two software repositories at the file level. Their approach is based on measuring the similarity between pair of source files and using the commit time to identify the original source file revision. They found that in some instances developers did not record the version of the reused file correctly.

Common to these studies is that they propose techniques to detect and investigate the reuse of software components. One of the main objectives of this thesis is to conduct empirical studies to assess qualitative and quantitative the reuse of code form crowdsourcing platforms such as (e., *npm* and Stack Overflow).

2.4 Chapter Summary

This chapter first reviews the definition of crowdsourcing in software engineering. Second, it surveys prior research on crowdsourcing in the software engineering domain. Specifically, it discussed work related to the integration of crowdsourcing in software development and how it is related to traditional source code reuse. Overall, prior studies on crowdsourcing software development have tended to focus on improving the software development process through the integration of crowdsourcing knowledge in the software development, but these studies do not consider how developers use the crowd knowledge in the software development process and what type of knowledge is provided by the crowd. Chapters 3 and 4 describe two empirical studies on the use of crowdsourcing knowledge from Stack Overflow. In Chapter 5, we examine the type of knowledge constructed on crowdsourcing platforms namely, *npm* and *PyPI*. Finally, in Chapter 6, we present our prosed rule-based technique to improve the efficiency of using crowdsourcing knowledge through the detection of commits that can be CI skipped.

Chapter 3

Understanding the Usage of Crowdsourced Knowledge

Stack Overflow relies on the crowd to construct quality developer-related knowledge. What developers use the crowd constructed knowledge on Stack Overflow for has not yet been fully addressed by existing research. As part of this research, we answer this question, by analyzing 1,414 Stack Overflow related commits and observe that developers use this crowd based knowledge to support development tasks and to collect user feedback. We also studied both the helpfulness and delay of Stack Overflow posts by identifying the type of questions that are more likely to be answered by the crowd. We find that development tools and programming language issues are areas where the crowd is most helpful and that web framework related questions take the longest to receive an accepted answer for. Our findings can help developers to better understand how to effectively use Stack Overflow as a development support tool, help Stack Overflow designers to improve their platform, and the research community to provide important insights on the strengths and weaknesses of Stack Overflow as a development tool.

3.1 Introduction

Question and Answer sites (Q&A), such as Stack Overflow, are extremely popular amongst software developers. Such Q&A sites heavily rely on the contributions of crowds to provide accumulated, quality knowledge to the software development community. Typically, users post questions related to software development topics on these Q&A sites that are answered by one or more participants. In essence, the job of answering the questions is outsourced to the crowd [108].

Over the years, the role of Q&A sites has evolved to more than just answering questions. However,

what role Q&A sites plays in today's development lifecycle is still an open question. Therefore, the goal of this chapter is to answer the questions "What reasons do developers resort to the crowd on Stack Overflow for?" and "What areas is the crowd most helpful in? and what areas take longer to obtain answers for?" Answering these questions will benefit developers to better understand what knowledge they can obtain from the crowd, what knowledge tends to be most helpful and what knowledge may take longer to attain.

Existing work, for example, Treude *et al.* [187] qualitatively analyzed a sample of Stack Overflow questions, and found that developers use Stack Overflow to share knowledge, provide development support, learn new technologies, and search for solutions to both common and specific programming problems. However, a key difference of our study is that we specifically examine cases where a developer who is making a code commit on GitHub has explicitly referenced a Stack Overflow post as the knowledge resource. The explicit mentioning of such Stack Overflow posts 1) provides us with confidence that a Stack Overflow is indeed related to a particular code commit, 2) allows us to build a richer dataset (since we have a link between the commit and the associated Stack Overflow post), and 3) indicates cases where developers see a need for traceability/justification/documentation for their committed changes. Moreover, our study does not focus on a specific type of task like existing studies (e.g., bug triage [16]), domain (e.g., mobile [163]), or programming language, rather we examine all type of commits that explicitly mention Stack Overflow.

Our findings corroborate some of the earlier observations [187], i.e., that developers use Stack Overflow to gain knowledge, and provide additional insights in terms of the actual type of Stack Overflow knowledge that is directly applied by programmers (e.g., we find that knowledge about specific programming language and API usage are the most common types of knowledge use from Stack Overflow). At the same time, some of our findings are novel and have not been reported in any of the earlier work. For example, we find that some developers use Stack Overflow to document known bugs and even implement features based on Stack Overflow posts. Moreover, we find that the crowd is most helpful in resolving questions related to development tools, programming languages, and implementation issues and that the most time consuming posts to answer are posts related to web frameworks and the documentation of bugs. The findings shed light on what developers use Stack Overflow for so that we can gain a better understanding of areas where crowd knowledge is most resorted to. Moreover, our study of the most helpful areas and the areas that take longest to acquire can be used to emphasize knowledge that requires more attention from practitioners and researchers that contribute to Stack Overflow.

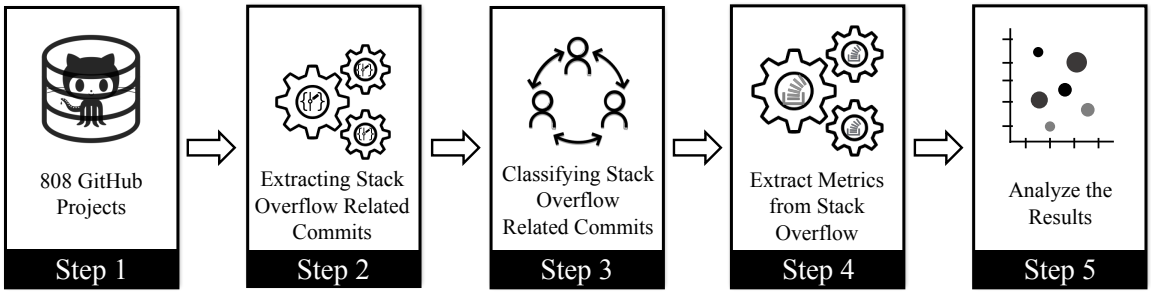


Figure 2: Overview of our data extraction and analysis approach

3.1.1 Organization of the Chapter

This chapter is organized as following: Section 3.2 describes the study design and approach. Section 3.3 presents our case study results. Section 3.4 discusses the implication of our results. In section 3.5, we highlights the work most related to our study. Section 3.6 discusses the threats to validity of our study. We conclude the Chapter in Section 3.7.

3.2 Study Design and Approach

The goal of our study in this chapter is to determine the reasons that developers use the crowd for in their own projects, what areas they find the crowd to be the most helpful in and the areas that are most time consuming to attain answers for. In the following sections, we describe how we collect our *Stack Overflow related commits*, how we classify them, and how we measure their helpfulness and delay. Figure 2 shows the overall methodology.

3.2.1 Selection of Studied Projects

To conduct our study, we first need to identify software projects that contain Stack Overflow related commits. At the same time, it is important to study a large sample of software projects in order to improve generalizability and confidence in our analysis results. To select the projects that we want to study, we used the GHTorrent dataset [73] to obtain a list of non-forked projects (mainline) written in the most popular programming languages [196]: Ruby, Python, JavaScript, PHP, Java, Scala, C and C++. Based on our selection criteria, we are able to identify 4,163,814 projects. However, since it is a well known fact that GitHub contains a large number of software projects that are inactive or immature, we set a few other constraints to ensure that we only consider active and mature projects. Thus, we only considered projects that: (1) have at least 100 pull request, (2) have at least three developers, and (3) have more than 100 commits in the last year. Similar constraints were recommended in [96]. Applying this filtering further reduced the number of projects to 4,026,

Table 2: Statistics of Projects, Languages, and Stack Overflow Commits Analyzed in Our Study.

Language	**# Projects**	**# Stack Overflow Related Commits**
JavaScript	189	307
Python	179	348
Ruby	132	227
Java	123	193
PHP	99	154
C/C++	65	131
Scala	21	54
Total	**808**	**1,414**

which we tried to clone for analysis. Since some projects were no longer available (e.g., they were deleted or made private), we were able to clone and study a total of 3,974 projects.

3.2.2 Extracting Stack Overflow Related Commits

After obtaining the list of software projects, the next step is to identify Stack Overflow related commits. For the identification process, we rely on string pattern matching techniques to detect these commits. For each project, we search all of their commit logs for the term 'stackoverflow' and its variants (i.e., capitalized first letter, all capitalized, with spaces). After applying the pattern matching technique, we obtained 1,780 *Stack Overflow related commits* that originated from 929 projects. As a final step, we performed a manual inspection of these commits and their associated projects to filter out duplicates and irrelevant commits (false positives). In the end, we were left with 1,414 commits from 808 projects, which we further analyzed. Of all the Stack Overflow related commits, approximately 97% of them contained a link to the Stack Overflow related post. Table 2 shows the descriptive statistics of our dataset. As mentioned earlier, the projects in the dataset cover several programming languages and each programming language has a number of related projects and commits.

3.2.3 Classifying Stack Overflow Related Commits

Once we determined the number of *Stack Overflow related commits* to be further analyzed, we preformed an iterative coding process to identify and categorize the different reasons that developers use Stack Overflow [170]. We first inspected every commit message, the source code associated with the commit, and the Stack Overflow post referenced in the commit. We read the main issues

discussed in the commit message and the Stack Overflow post, which helps us determine how to classify the commit. The aforementioned process was performed iteratively in that every time a new category is added, we re-examine all the previously classified commits to determine if the categorization changed. As a result of this classification process, we ended up with 14 different reasons why developers mention Stack Overflow in source code commits.

Like any human activity, our classification is prone to human bias. To examine the validity of the classification, we got another Ph.D. student to independently classify a statistically significant sample of commits to reach a 95% confidence level using a 5% confidence interval. The statistically significant sample of 302 Stack Overflow related commits was classified into different areas and Cohen's Kappa coefficient was used to evaluate the level of agreement between the two annotators. Cohen's Kappa coefficient is a well-known statistical method that is used to evaluate the inter-rater agreement level for categorical scales. The resulting coefficient is scaled to range 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. In our work, we found the level of agreement between the annotators to be +0.78, which is considered to be excellent agreement [64].

3.2.4 Measuring the Helpfulness and Delay of Stack Overflow Posts

The second goal of our study is to better understand which areas developers find the crowd to be most helpful in and which areas take longer to attain an answer to, i.e., which posts take longer to receive an accepted answer for.

As a proxy for helpfulness, we use the number of votes (sum of upvotes - downvotes) that a Stack Overflow question receives. Our intuition here is that if a question is helpful to a developer, then they will give it an upvote, which indicates that this question/post is helpful. The more votes a question has, the more helpful it is considered to be. Once we measure the votes for the individual posts, we group them into their respective areas. To provide a **helpfulness** measure for a specific area, we present the median of the votes for all of its posts.

We further argue that if a post takes longer time to obtain an accepted answer, then the developer will be delayed more, which is negatively perceived. Therefore, we use the time to obtain an accepted answer for posts in a specific area as a proxy for **delay**. For each area, we measure the time difference between the initial post (question) and its first accepted answer. Then, we aggregate all of the values and present the median time per area.

3.3 Results

Table 3: Identified reasons for using Stack Overflow with a description, example and the percentage of commits in each area.

Reasons for Using Stack Overflow	Description	Example	%Comit.
Programming Languages	Any knowledge related to programming languages and their features. For instance, using format in Python, using sequential for loop in JavaScript, how to do casting, regular expression, or a programming language limitation.	"Changed all boolean casts that were using Boolean() function to use double negation(!!), which is faster: *[StackOverflow Link]*"	22.07%
API Usage	Knowlege related to how to use an API including argument, deprecation, specify method to perform a task in an API.	"mailutils: send_email() with attachments* Extends mailutils. send_email() API to support attachments, following the recipe:*[StackOverflow Link]*"	21%
Configuration Management	Knowledge related to configuration management. For example, knowledge on how to configure Maven tool in the development environment.	"Fix maven assembly warning about using root dir. It's a bad practice in Maven to define '/' as the output dir. It's better to leave it empty. See also *[StackOverflow Link]*."	7.21%
Web Frameworks	Stack Overflow posts related to the usage of web frameworks and their configurations.	"Fix client names with dot do not work this is Spring Framework MVC behavior as described in *[StackOverflow Link]*"	6.51%
Web Browsers	A developer uses knowledge regarding web browsers. For example, the presence or absence of features in specific browsers.	"Fix grid context menu position for Firefox.Firefox does not have offsetX. pageX is absolute and lets the menu jump all over the place. Solution based on *[StackOverflow Link]*"	4.31%
Development Tools	Knowledge related to configuring development tools (e.g., IDE, Git, SVN) versions, settings, etc.	"Update Git to delete a remote branch with '-delete' more memorable syntax. Use Git1.7 syntax based on this answer: *[StackOverflow Link]*"	4.17%
Implementation Issues	Developers used suggestions or tutorials from Stack Overflow posts to implement an algorithm or a feature in their projects without copying and pasting source code.	"Introduce the non-daemon process pool as an alternative to the original multiprocessing pool. This adds support for hierarchical multiprocessing (child classes can use multiprocessing again). The code is based on the following StackOverflow answer:*[StackOverflow Link]*."	3.89%
Database Technologies	Knowledge related to database and their supporting technologies (e.g., database configurations, maintenance, modeling, queries, etc.).	"Postgres column renaming. Switched "name" column name to "shoreline_name" so we don't collide with possible Postgreskeywords/types *[StackOverflow Link]*."	2.83%
Operating Systems	Knowledge related to operating systems features or issues.	"Explicitly set empty extension name for backup files on Mac, this parameter is needed, otherwise an error is shown. See this SO post for more information: *[StackOverflow Link]*"	2.40%
Documenting Bugs	The developer fixed a bug in the project and provided the link to the Stack Overflow post where the bug has been described.	"Fix AttributeError when IssueEvent has assignee. This was discovered by a user on StackOverflow *[StackOverflow Link]* and fixed as soon as I realized it was a bug."	13.08%
Promoting Stack Overflow	A developer introduced a tag on Stack Overflow related to his/her project to facilitate its documentation or to promote the usage of their tag on Stack Overflow.	"Promote stackoverflow for questions", "Drop google groups in favor of stackoverflow tag." , "Link to *[StackOverflow Link]* for Q and A Thanks to Vincent Scheib for arranging and Paul Kinlan for donating his Stack Overflow karma to create the tag"	3.18%
Feature/System Improvements	A developer implements a new feature or improves the project based on Stack Overflow users request.	"Extend key bindings for prompt commands to support predefined searches This adds support for binding keys to ':/' and ':?',for example: bind stage 2: Based on this request by Joelpet on stackoverflow:*[StackOverflow Link]*"	1.77%
Code Reuse	A developer copies and pasts a source code snippet from a Stack Overflow post.	"Close tip popup on click outside the tip box. Credit: *StackOverflow Link*"	1.70%
Other	Developers use knowledge from Stack Overflow, but we cannot identify the type of usage exactly or some rare cases that it is not worth of having a separate category for them.		5.87%

To understand what developers use the crowd constructed knowledge on Stack Overflow for, we break down our study into two main parts and associate a research question with each part:

RQ1: What are the main reasons developers resort to Stack Overflow?

RQ2: What areas is the crowd most helpful to developers in? What areas takes longest to attain answers for from the crowd?

Answering RQ1 helps us to determine what developers use Stack Overflow for so that we can gain a better understanding of areas where crowd knowledge is most resorted to. We can use our findings to further facilitate the integration of crowdsourcing into software development. Answering RQ2 helps us to better understand what types of knowledge from Stack Overflow is considered most helpful by the developers and takes longest to acquire. Identifying these types of knowledge can be used to emphasize knowledge that requires more attention from practitioners and researchers that contribute to Stack Overflow.

3.3.1 RQ1: What are the Main Reasons Developers Resort to Stack Overflow?

As mentioned earlier, we manually examined each commit message, the code changes associated with a commit, and the Stack Overflow post mentioned in the commit to determine the reason the commit mentions the Stack Overflow post. Thus, we use our classification to identify the reasons why developers use Stack Overflow.

Table 3 shows the 14 different reasons developers use Stack Overflow, that are grouped into five high-level categories, namely 'Using Knowledge', 'Documenting Bugs', 'Promoting Stack Overflow', 'Feature/System Improvements' and 'Code Reuse'; another category, 'Other', was added to categorize commits that rarely appeared and/or did not fit into any of the major categories. For each reason, we provide a description, an example and the frequency they occurred (as a percentage of commits). As shown in Table 3, we found that developers resort to the crowd on Stack Overflow mainly to gain knowledge. The most frequent knowledge is related to programming languages (in 22.07% of the commits), to ask about API use (in 21% of the commits), configuration management (in 7.21%), gain knowledge about web frameworks (in 6.51% of the commits), and web browsers (in 4.31% of the commits). From our analysis of the commits and posts related to the aforementioned categories, we observed that the developers mainly take advantage of the technical knowledge provided by the crowd on Stack Overflow. Our findings show the key role of the crowd is to support and complement traditional documentation.

We also found that developers use Stack Overflow to document bugs (in 13.08% of the commits) and even for feature/system improvements that they implement (in 1.77%). These findings show

that the role of the crowd on Stack Overflow is more than just providing knowledge or finding relevant code. The crowd on Stack Overflow can also provide insight on known issues and features that users would like to see. These results suggest that developers pay attention to such issues raised by the crowd and that Stack Overflow serves as medium for identifying and tracking feature requests and issues.

The other two interesting categories were less popular and related to the promotion of Stack Overflow in a project (in 3.18% of the commits), where developers would introduce a tag on Stack Overflow to facilitate documentation. Lastly, we found that the direct reuse of code from Stack Overflow is very minimal in terms of the number of explicit mentioning of the Stack Overflow posts in commit messages (in 1.70% of the commits). However, we believe that developers reuse more code than they admit due to various reasons such as potential copyright violations or plagiarism.

> Developers most often use the Stack Overflow to gain knowledge related to topics such as development tools, APIs usage, and operating systems. More importantly, Developers use Stack Overflow to provide rationale for feature updates or additions. In fact, developers even use Stack Overflow as a communication channel to receive user feedback about their software.

3.3.2 RQ2: What Areas is the Crowd Most Helpful to Developers in? What Areas Takes Longest to Attain Answers for from the Crowd?

Now that we have identified the different reasons developers use Stack Overflow, we want to better understand what knowledge the crowd is most helpful for and what knowledge takes longest to acquire.

Table 4 shows the number of questions, the median number of votes, the number of accepted answers, and the median time (hours) to obtain an accepted answer for each reason. The ascending ranking for helpfulness is based on the median votes for the different reasons of knowledge reuse from Stack Overflow and the descending ranking for delay is based on the median hours it takes to obtain an accepted answer for a given category.

From Table 4, we observe that posts related to development tools, programming languages, implementation issues, and configuration management are areas where the crowd can be considered to be the most helpful. On the other hand, the posts related to web frameworks, documenting bugs, and development tools are areas that take the longest to answer.

However, the most interesting analysis comes from combining these two views, i.e., helpfulness and delay, to determine areas that developers can expect to receive helpful answers in a timely

Table 4: The Helpfulness & Delay classified by the different reasons of using knowledge from Stack Overflow.

Reason	**Helpfulness**			**Delay**		
	#Questions	**Median of Votes§**	**Rank**	**#Accepted Answers**	**Median of Time†**	**Rank**
Development Tools	61	39	1	52	6.7	9
Programming Languages	310	29	2	279	0.3	1
Implementation Issues	55	26	3	48	2.6	4
Configuration Management	102	19	4	87	5.6	7
Database Technologies	41	17	5	37	1.1	3
Web Browsers	61	16	6	50	0.5	2
Web Frameworks	93	15	7	77	13	11
Operating Systems	37	13	8	33	0.5	2
API Usage	301	10	9	254	3.1	5
Code Reuse	25	7	10	19	6.1	8
Feature/System Improvements	27	3	11	18	3.6	6
Documenting Bugs	178	2	12	139	9.4	10
Promoting Stack Overflow	0	0	NA	0	0	NA
Other	76	15	-	59	2.6	-

§The median of number of votes a question receive on Stack Overflow.

†The median time taken for a question to receive an accepted answer in hours.

manner and vice versa. We use a bubble plot, shown in Figure 3, that plots the ranks of helpfulness vs. delay for each area of using Stack Overflow. The size of the bubble represents the number of commits for a particular area. From Figure 3, we observe that areas such as 'implementation issues', 'programming languages', 'database technologies', and 'web browsers' provide the highest utility for developers, i.e., developers receive very helpful and quick answers. On the other hand, areas such as 'API usage' and 'operating systems' tend to be answered quickly, but the answers given are not perceived to be very helpful. Similarly, answers to questions in areas such as 'web frameworks' and 'documenting bugs' are perceived to be less helpful and take even longer to answer. In such cases, the crowd on Stack Overflow may not be the right resource for developers who are looking for answers.

The crowd was the most helpful on topics such as development tools and programming languages. The questions that took the longest to resolve were related to Web frameworks. In addition, developers reuse source code snippets from Stack Overflow. However, code snippets from Stack Overflow seem to be not so helpful.

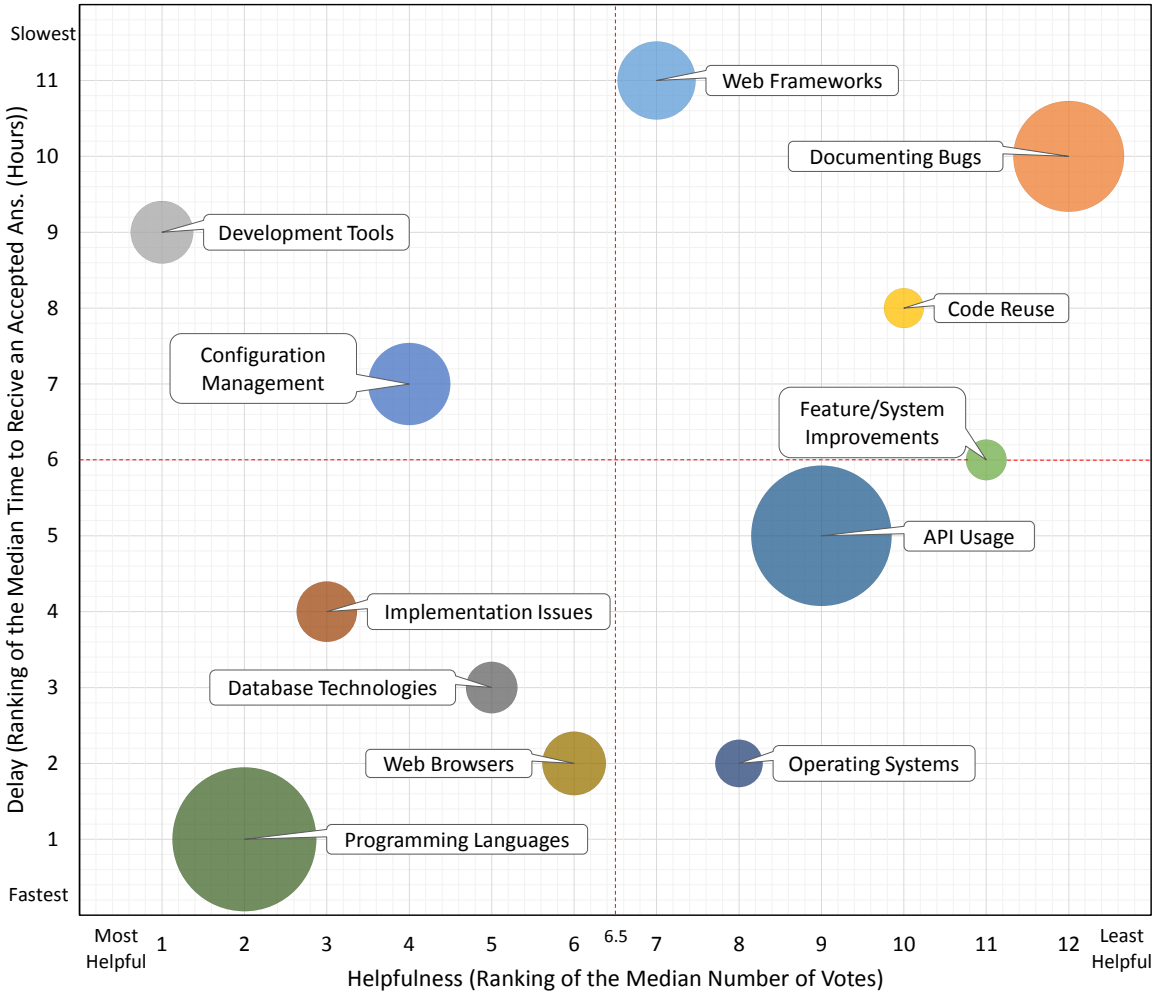


Figure 3: The x-axis of the plot shows the average ranking of the median of number of votes for each area (1 is the most helpful). The y-axis shows the median time (hour) for a question to receive an accepted answer for each area (1 as the fastest), while the size of the bubble present the number of commits.

3.4 Discussion and Implications

There are two key observations that we believe can impact future development of Q&A sites and how developers use the crowd. The first observation can be helpful for Stack Overflow designers to enhance current Stack Overflow features to meet the increasing demand from developers. The second is an observation that can help developers improve traceability and documentation of their changes.

3.4.1 Using the Crowd for More Than Just Knowledge

Clearly, our results show that the majority of developers use Stack Overflow to gain knowledge. However, we also observed that developers use the crowd on Stack Overflow for more than acquiring knowledge. In fact, developers seem to be using these crowd based resources to document bugs and determine features that they want to implement. Hence, we believe that future versions of Stack Overflow need to incorporate a mechanism where developers can obtain direct feedback from the crowd. Presently, we see that role of the crowd to mainly focus on reporting bugs and requesting requirements. However, we envision that in the future the crowd on Stack Overflow will play an increasing role, as a source for refining requirements, providing testing, and even helping refine software design. Another interesting finding is that developers reuse code snippet from Stack Overflow. How to ensure the quality or integrity of these shared code snippets is an area where crowd based platforms can do better (now all someone can do is give an upvote). Also, providing some sort of scoring system that indicates the 'adaptability' or 'ease of integration' of a code snippet would be beneficial.

One important suggestion based on this specific finding for Stack Overflow designers is to provide techniques to assess the quality of source code snippets posted on Stack Overflow, in order to investigate and find ways one can automatically generate test cases for such source code snippets posted on Stack Overflow, for example.

3.4.2 Linking Changes to Crowd Discussion

Our dataset is based on the fact that developers explicitly mentioned the Stack Overflow posts in their commit messages. With the increasing use of the crowd in software development, we believe that developers should link to discussions that they used to help them reach their final coding solutions. These discussions can help document and provide rationale for certain programming decisions, hence developers should provide links to them in their commits. Much like how commits contain bug IDs, we believe that in the future, every commit should also provide a link to any crowd-based discussions that are related. One unique feature of crowd-based discussions is that they continue to evolve, and

so even if a bug is found in posted code, others may help provide an update or a fix in the future. If links to these discussions are provided, this evolved code can help address future issues with the code.

3.5 Related Work

We reviewed prior research on the use of Stack Overflow as a crowdsourcing platforms in Chapter 2. In this section, we discuss the work that is most closely related to this Chapter. In addition to the study by Treude *et al.* [187], other works also studied how developers use Q&A sites. Barua *et al.* [19] proposed a semi-automatic approach to study general topics discussed on Stack Overflow and their trends and found that web and mobile development are the most popular topics. More specifically, Bajaj *et al.* [17] used Stack Overflow data to analyze common challenges and misconceptions among web developers. Rosen and Shihab [163] used Stack Overflow to determine what mobile developers on Stack Overflow ask about. Other researchers have performed studies that examine how Stack Overflow affects developers' activities during software development. For example, Vasilescu et al. [192] analyzed the effect of Stack Overflow activities on the software development process. They established associations between GitHub and Stack Overflow users, and found a correlation between participants' activities in the two platforms. Zagalsky *et al.* [207] also investigated the use of Stack Overflow and mailing lists as communication channels for the R project. They found that both resources provide active communication channels where participants are willing to help others. They also observed that Stack Overflow resorts to a crowd-based knowledge construction approach, where participants contribute knowledge independently, whereas for mailing list the focus is on improving specific answers.

In many ways, our work shares similar goals with these prior studies, i.e., to determine what developers use the crowd for during software development. However, our study differs in that we only consider explicit links between Stack Overflow posts and source code commits, to ensure that we only consider actual knowledge reuse from Stack Overflow in our analysis. Moreover, we use characteristics derived from these posts and commits to understand what knowledge is most helpful and what knowledge is most time consuming to attain.

3.6 Threats to Validity

There are a number of limitations to our study. First, the commits were manually classified by the first author. Like any human activity, this process is susceptible to human error. To ensure the validity of the classification, we got another Ph.D. student to classify a statistically significant sample

of 308 commits and found their agreement to be excellent (Cohen's Kappa value of +0.78) [64]. Also, our findings are based on 1,414 commits, where developers explicitly mention Stack Overflow. There may be other cases where developers use Stack Overflow, but do not mention it in the commit message. Lastly, our study is based on open source projects that are hosted on GitHub, therefore, our study may not generalize to other open source or commercial projects.

3.7 Chapter Summary

In this chapter, we investigate the reasons developers use Stack Overflow for and what areas the crowd is most helpful and what areas are the most time consuming to attain answers for. We find that the crowd mostly provide technical knowledge to developers, however, the role of crowd-based sites, such as Stack Overflow is evolving. Our results revealed that using crowd knowledge through Q&A platforms, such as Stack Overflow, can be used for various purposes of the software development process including collecting users' feedback and code reuse. We draw from our findings to suggest that crowd-based sites such as Stack Overflow provide tools to support feedback from the crowd to developers and provide mechanisms to evaluate the quality of code posted on such sites. For developers, the ability to provide direct links to crowd-based resources will become essential since such links can serve as living documentation of their code and/or design decisions. In addition to its direct findings, our study highlights areas where crowd based knowledge may not be the best fit, e.g., for questions related to web frameworks.

In the next chapter, we focus on the impact of reusing crowdsourcing knowledge on software quality. We first describe our proposed approach to detect reused code from Stack Overflow. Then, we examine how much code is reused form Stack Overflow and what is the impact of the reused code on the software quality.

Chapter 4

The Impact of Reused Source Code from Crowdsourcing Platforms

Source code reuse has been widely accepted as a fundamental activity in software development. Recent studies showed that Stack Overflow has emerged as one of the most popular resources for code reuse. Therefore, a plethora of work proposed ways to ask questions, search for answers and find relevant code on Stack Overflow. However, little work exists on the impact of code reuse from Stack Overflow. To better understand the impact of code reuse from Stack Overflow, we perform an exploratory study focusing on code reuse from Stack Overflow and more specifically in the context of mobile apps. As part of our study, we investigate how much, why, when, and who reuses code from Stack Overflow. Moreover, to understand the potential implications of such crowd-based code reuse, we examine the percentage of bugs in files that reuse Stack Overflow code. As part of our research, we perform our study on 22 open source Android apps. For each project, we mine their source code and use clone detection techniques to identify code that is reused from Stack Overflow. We then apply different quantitative and qualitative methods to answer our research questions. Our findings indicate that 1) the amount of reused Stack Overflow code varies among mobile apps, 2) feature additions and enhancements in apps are the main reasons for code reuse from Stack Overflow, 3) mid-age and older apps reuse Stack Overflow code mostly later on in their project lifetime and 4) that in smaller teams/apps, more experienced developers reuse code, whereas in larger teams/apps, the less experienced developers reuse code the most. Additionally, we found that the percentage of bugs is higher in files after reusing code from Stack Overflow. Our results provide insights on the potential impact of code reuse from Stack Overflow on mobile apps. Furthermore, these results can benefit the research community in developing new techniques and tools to facilitate and improve code reuse from Stack Overflow.

4.1 Introduction

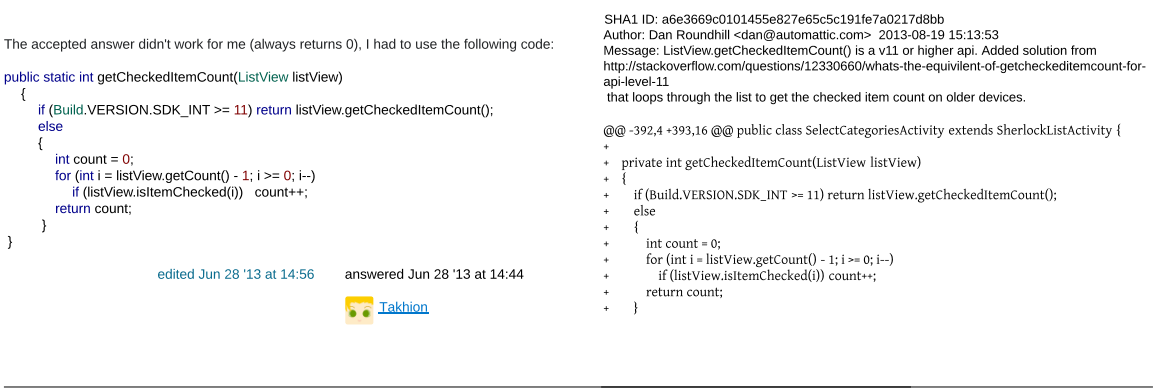
A key premise of software development is to deliver high-quality software in a timely and cost-efficient manner. Code reuse has been widely accepted to be an essential approach to achieve this premise [113]. The reused code can come from many different sources and in different forms, e.g., third-party libraries [113], source code of open source software [92], and Question and Answer (Q&A) websites such as Stack Overflow [162, 166].

In recent years, the development of mobile apps has emerged to be one of the fastest growing areas of software development [77]. Some of the most common characteristics of mobile apps are: (1) they are developed by small teams [184], (2) they are developed by less experienced programmers and (3) they are constrained by limited resources [135]. As a result of these constraints, mobile developers tend to resort frequently to Q&A websites such as Stack Overflow for solutions to their coding problems [162, 19]. A primary challenge with this type of code reuse is that programmers often resort to these code snippets in an ad-hoc manner, by copying-and-pasting these fragments in their own source code. However, the impact of this ad-hoc reuse on software processes, product quality, and mobile app development at large remains an open question.

There has been a plethora of work focusing on Stack Overflow (e.g., [151, 143]), code reuse (e.g., [113, 72]) and mobile development (e.g., [116, 168]), which studied all of these topics in isolation. However, vital questions about how mobile developers reuse code from Stack Overflow remain unanswered. This is particularly important since 1) prior work showed that Stack Overflow is very popular among mobile developers [162, 19] and 2) as we report later in this chapter, once code is reused from Stack Overflow, it can potentially have a negative impact on the target mobile app.

Therefore, in an effort to better understand code reuse from Stack Overflow amongst mobile developers, we perform an exploratory study using quantitative and qualitative methods on 22 open source mobile apps. In particular, we answer three fundamental questions about code reuse from Stack Overflow which are: RQ1) Why is Stack Overflow code reused in mobile apps? It is important to study why Stack Overflow code is reused since it helps us understand the key reasons and tasks where mobile app developers resort to Stack Overflow. RQ2) At what point of time during the development process of mobile apps does code reuse from Stack Overflow occur? Knowing when in the project's lifetime code is mostly reused from Stack Overflow helps us better understand in what stage of the development mobile developers need the most help and resources. RQ3) Who reuses code from Stack Overflow? Knowing who reuses code from Stack Overflow can provide us with insights on the type of reuse (e.g., is it reused due to lack of experience or is it well thought-out reuse by experienced developers).

We find that the amount of reused Stack Overflow code varies among mobile apps. Also, the main reasons for code reuse from Stack Overflow are to implement new features, enhance existing



The accepted answer didn't work for me (always returns 0), I had to use the following code:

```
public static int getCheckedItemCount(ListView listView)
   {
      if (Build.VERSION.SDK_INT >= 11) return listView.getCheckedItemCount();
      else
      {
         int count = 0;
         for (int i = listView.getCount() - 1; i >= 0; i--)
            if (listView.isItemChecked(i))   count++;
         return count;
      }
}
```

edited Jun 28 '13 at 14:56 answered Jun 28 '13 at 14:44

Takhion

(a) Code snippet from Stack Overflow

```
SHA1 ID: a6e3669c0101455e827e65c5c191fe7a0217d8bb
Author: Dan Roundhill <dan@automattic.com>  2013-08-19 15:13:53
Message: ListView.getCheckedItemCount() is a v11 or higher api. Added solution from
http://stackoverflow.com/questions/12330660/whats-the-equivilent-of-getcheckeditemcount-for-
api-level-11
 that loops through the list to get the checked item count on older devices.

@@ -392,4 +393,16 @@ public class SelectCategoriesActivity extends SherlockListActivity {
+
+   private int getCheckedItemCount(ListView listView)
+   {
+      if (Build.VERSION.SDK_INT >= 11) return listView.getCheckedItemCount();
+      else
+      {
+         int count = 0;
+         for (int i = listView.getCount() - 1; i >= 0; i--)
+            if (listView.isItemChecked(i)) count++;
+         return count;
+      }
```

(b) A Commit that reuse the code snippet

Figure 4: (a) Source code snippet posted on Stack Overflow, (b) Description of a commit that reuses the code snippet in WordPress Android

functionality, refactor code, use APIs, and test source code (RQ1). Moreover, we observe that mid-age and older apps reuse Stack Overflow code later on in their lifetime (RQ2). With regards to the experience of the developers who reuse code depends on the team/app size; more experienced developers reuse code in smaller teams/apps, while less experienced developers reuse code in larger teams/apps (RQ3). Finally, to shed light on the potential impact of reused Stack Overflow code on mobile apps, we examine bug fixing commits of files that contain code reused from Stack Overflow. We find that these files have a higher percentage of bug fixing commits after the introduction of reused Stack Overflow code, indicating that reusing such code may negatively impact the quality of mobile apps.

4.1.1 Organization of the Chapter

The rest of this chapter is organized as follows: Section 4.2 introduces a motivated example of our research. Section 4.3 sets up our case study. Section 4.4 discusses our preliminary analysis on how much reuse occurs in mobile apps. In Section 4.5, we report our case study results. We discuss the implications of code reuse on mobile app quality in Section 4.6. Related work is presented in Section 4.7. Section 4.8 presents the threats to validity and Section 4.9 concludes our study.

4.2 Motivating Example

Existing research (e.g., [19, 162]) has shown that developers of mobile apps resort to Stack Overflow for help (e.g., to resolve implementation problems or examine best coding practices). In contrast to this existing work, the objective of our research is to study code reuse from Stack Overflow during

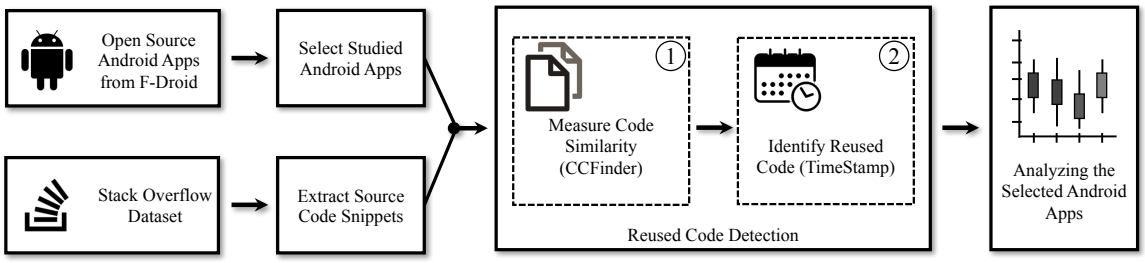


Figure 5: Approach overview of our study

the implementation or maintenance of mobile apps.

In what follows, we show the motivation for our research, by describing a real world example, where code has been reused from Stack Overflow in the WordPress[1] Android app (we also elaborate more on how much reuse is done later on in this chapter). Figure 4a shows an answer post with a code snippet which has been posted on Stack Overflow[2] to provide a solution for the use of the getCheckedItemCount() method in the ListView class of the Android API, with the getCheckedItem-Count() method returning the number of items currently selected in the list.

The post describes a compatibility issue that arises when the method is used in older Android API versions (prior to version 11). In the WordPress example, a developer encountered the same problem using the ListView class and resorted to the code solution posted on Stack Overflow. Figure 4b shows details of the actual commit that reuses the code from Stack Overflow, which in this case even includes the link to the original Stack Overflow post. In addition, using the timestamps of the commit and the Stack Overflow post, we are able to determine that the code was committed in the WordPress for Android project after it was posted on Stack Overflow.

Being able to identify that Stack Overflow code exists in a mobile app is important for several reasons: first, the reused code can be considered third-party code that may negatively impact the project. In particular, given that the origin of these code snippets is unknown, the reused code needs to be carefully reviewed since it may be incomplete and/or may have been developed for a different context. Second, although we do not address it in this chapter, using Stack Overflow code can lead to potential license violations in the mobile app reusing the code snippet.

4.3 Case Study Setup

As mentioned before, the goal of this chapter is to perform an exploratory study on the reuse of Stack Overflow code in mobile apps. To perform our study, we extract code snippets from Stack

[1] `https://github.com/wordpress-mobile/WordPress-Android`

[2] `http://stackoverflow.com/q/12330660/whats-the-equivilent-of-getcheckeditemcount-for-api-level-11`

Overflow in order to examine the code reuse. We then select 22 open source mobile apps and examine their source code for potential code reuse from Stack Overflow. Figure 5 provides an overview of our approach. Once we identify the reused code, we analyze our dataset and answer our research questions.

4.3.1 Building Stack Overflow Code Snippets Corpus

To perform our study, we first downloaded and extracted code snippets from Stack Overflow. We obtained the Stack Overflow data dump (published March 16, 2015) in XML format. The data dump contained 24,120,523 posts, including 8,978,719 questions and 15,141,804 answers. Each discussion includes a question and zero or more answer posts and their meta data *(e.g., body, creation date and number of votes)*. Questions are typically tagged with terms describing the categories under which these Q&A discussions are grouped.

During the next processing step, we extract the relevant discussions for a study context. Since we focus on Android open source apps, we extracted all discussions related to the "Java" tag (810,071 discussions). We then further filtered these discussions to ensure that they also contain the "Android" tag, which left us with 106,861 discussions related to Java and Android. Since we are only interested in code reuse from Stack Overflow, we further limited our dataset to only consider discussions that contain source code. For our analysis, we focused on the source code in the answers of the Stack Overflow discussions, since code in questions is not likely to be reused. This further reduced the set of discussion posts to 53,683. Since prior work suggested the use of highly voted code snippets, we only focused on code snippets in answers that have a high number of votes [143]. We therefore examined the average votes for posts with their code snippets size (P_{size}) within our 53,683 selected Stack Overflow posts where, $P_{size} \leq 5$ *lines*, $P_{size} > 5 \cap P_{size} < 30$ *lines* and $P_{size} \geq 30$ *lines*. We found that snippets with 30 or more lines had the highest average votes with 2.02 while snippets with ≤ 5 *lines*, $> 5 \cap < 30$ *lines* have lower average numbers of votes (1.85 and 1.69, respectively). Therefore we decided to focus only on the posts which contained snippets with at least 30 lines of code because 1) they have the highest average votes, i.e., Stack Overflow users find them the most helpful, 2) smaller code snippets would result in too many false positives (e.g., it may flag common and simply code such as `if` and `for` loop statements), 3) choosing 30 lines of code will decrease the size of the corpus and increase the scalability of the clone detection tool. In the end, our dataset contains 7,458 posts. Table 5 summarizes how we arrived at this final number of posts.

4.3.2 Extracting Code Snippets from Stack Overflow

Once we identified all of the Stack Overflow posts that contained source code, we needed to extract the code snippet from these posts. We applied a lightweight heuristic to identify the code in the

Table 5: Selection process of Stack Overflow posts

Step	**# Posts**
All posts in the Stack Overflow dataset	24,120,523
Posts tagged with Java	810,071
Posts tagged with Java and Android	106,861
Answer posts contain source code snippets	53,683
Code snippets with >= 30 lines	7,458

Stack Overflow posts. In Stack Overflow, the body of a post is provided in HTML. Code elements in these posts are surrounded by the <code> tag. Therefore, we identify the code elements by searching for the <code> tags. Once we extracted the code elements, we manually checked the contents of the text between these <code> tags. We found that many posts do not only contain Java code. For example they may contain stack traces, XML code, plain text or URLs. To eliminate this non-Java code, we read the <code> tag contents and removed such code by applying regular expressions to filter out lines that start with special characters such as <,#,?,$,.,-, or a blank line. Moreover, even for the posts with Java code we eliminated all lines that begin with import statements. Performing the aforementioned preprocessing steps are necessary to further improve the matching between the Stack Overflow code snippets and code from the mobile app.

4.3.3 Selecting Mobile App Projects

In addition to obtaining the Stack Overflow code snippets, we require the source code from mobile apps to study the code reuse. For our study, we resorted to the well-known F-Droid repository [5]. At the time of writing this work, F-Droid included 1,591 open source mobile apps. In addition to providing the APKs of each app, F-Droid provides a link to the source code of each app, which we used to extract the source code of the mobile apps.

Table 6 presents a summary of the steps, which we performed to arrive at our dataset of 22 mobile apps. For each of the 1,591 apps, we examined the amount of reuse in the app exploiting an existing clone detection tool (step ❶ in Figure 5, which is described later in this section). We were able to run the clone detection tool successfully on 1,496 of the 1,591 downloaded apps. Of the 1,496 apps only 377 had one or more clones from Stack Overflow code. Next, we filtered apps that had at least one instance of code reuse from Stack Overflow. We make this check by comparing the date that the code segment was inserted in the project, using the commit date of the code with the date that the Stack Overflow code was posted. If the commit date is *after* the date of the Stack Overflow post, then we can conclude that the code is reused from Stack Overflow (step ❷ in Figure 5). This additional filter step further reduced our original dataset to 22 mobile apps.

Table 6: Selection process of the studied mobile apps

Step	# **Apps**
Downloaded from F-Droid	1,591
The ones we were able to run CCFinder on	1,496
That have at least one clone from Stack Overflow	377
That reused at least one case of code from Stack Overflow	22

To illustrate the diversity of our mobile app dataset, we present various app statistics in Table 7. As presented in Table 7, the apps belong to a number of different categories, that vary by the number of commits and contributors. Finally, the studied apps range in size from 745 to 151,264 lines of code (LOC).

4.3.4 Detection of Reused Code from Stack Overflow in the Mobile Apps Case-Study

Once we created our Stack Overflow code snippets corpus and the code from the mobile apps, our next step is to detect the reused code from Stack Overflow within the mobile apps. We use the CCFinder[98] clone detection tool. CCfinder is a token based clone detection tool developed to detect Type-1 and Type-2 clones [98]. Three types of clones can be distinguished: Type-1 clones, which means that the two detected code snippets are identical. Type-2 clones, which means that the two source code snippets have the same structure except for variation in identifiers and literals, while Type-3 consider two code snippets to be similar even with further modification than Type-2 clones [104, 164]. In our study, we detect similar code snippets utilizing Type-1 and Type-2 clones. Restricting our analysis to type-1 and type-2 clones a) reduces the number of potential false positives compared to type-3 clones, especially given that Stack Overflow contains a large amount of code snippets, while b) providing an acceptable recall (compared to just using type-1 clones).

We execute CCFinder using its default configuration, i.e, the minimum length of the detected code clones is 50 tokens. It is important to note that although the code snippets extracted from Stack Overflow needed to be a minimum of 30 lines, the overlap between any code from a mobile app and a Stack Overflow code snippet has to be only 50 tokens, for the code fragment to be flagged as a clone. For example, we may have a case where only 5 lines (which have more than 50 tokens) from a mobile app appear in a Stack Overflow snippet that is 30 lines long.

We selected CCFinder as our clone detection tool for several reasons. First, it detects type-1 and type-2 clones. Second, it is very efficient with respect to CPU and memory usage. Finally, it is freely available for research purposes. CCfinder returns clone groups with different sizes based on the number of matched tokens. In some cases, CCFinder may indicate multiple clones (of different

Table 7: Descriptive statistics of the 22 mobile apps and the percentage of reused code from Stack Overflow

ID	App's Name	Category	#Comit.	#Contr.	#LOC	%Reused
1	OsmAnd	Travel & Local	23,124	419	151,264	0.20
2	Open Explorer	Productivity	1,669	10	130,565	0.07
3	WordPress	Social	11,467	54	69,760	0.19
4	AnkiDroid	Education	7,463	110	45,088	0.08
5	Barcode Scanner	Tools	3,152	77	42,514	0.38
6	ForPDA	Social	170	2	42,254	1.30
7	APG Encrypt	Communication	4,366	68	41,564	0.70
8	Xabber Classic	Communication	1,005	15	37,091	0.30
9	Smart Receipts Pro	Finance	301	2	28,136	0.80
10	F-Droid	Tools	2,550	55	21,039	0.48
11	FrostWire	Media & Video	3,763	28	19,713	0.10
12	Andlytics Track	Shopping	1,388	24	16,694	0.29
13	Open Training	Health & Fitness	501	6	10,264	1.36
14	BeTrains NMBS/SNCB	Transportation	209	7	9,421	3.52
15	YASFA	Social	21	2	9,128	1.57
16	Secrecy Secure file storage	Tools	155	1	6,549	1.95
17	Tram Hunter	Travel & Local	260	5	5,516	0.42
18	OpenLaw	Book & Reference	333	1	4,037	0.37
19	OCR Test	Productivity	101	1	3,910	5.70
20	OpenDocument Reader	Business	233	1	2,996	0.87
21	Blippex	Tools	18	3	2,050	1.61
22	AnagramSolver	Word	46	2	745	0.94
-	**Average**	-	2,831.6	40.59	31,832	1.06
-	**Median**	-	417.0	6.50	18,204	0.59

sizes) in one code segment. In such cases, we take the largest clone as the match, since we need to detect the most similar code to the Stack Overflow code snippet.

Once we obtain the code snippets that are reused from Stack Overflow, we analyze the amount of reuse in the studied mobile apps, which we discuss next.

4.4 How Much Source Code from Stack Overflow is Reused in Software Systems?

Prior to delving into our research questions, we performed a preliminary analysis to quantify how much code reuse occurs from Stack Overflow. Since recent work has shown that mobile developers

often resort to Stack Overflow [162, 19], this investigation helps in quantifying how much (in terms of code reuse) Stack Overflow is being used as a resource by mobile app developers.

To perform our analysis, we measure reuse in two complementary ways. First, we measure the percentage of Stack Overflow posts that are reused in mobile apps. Second, we measure the percentage of code within mobile apps, which is reused from Stack Overflow. We chose to use percentages instead of raw numbers since they allow us to easily compare the values.

Our analysis shows that from the 7,458 posts in our Stack Overflow code snippet corpus, 99 posts contain code snippets that were reused in the 22 studied mobile apps. This shows that approximately 1.33% of the Stack Overflow posts are reused in the 22 studied mobile apps. It is important to note however, that in some cases, some posts in our corpus are reused more than once. As for the amount of a mobile app's code that is reused, Table 7 (column 7) shows the percentage of the app that is reused from Stack Overflow. Table 7 shows that the OCR Test app (5.70%) has the highest amount of source code originating from Stack Overflow, while the Open Explorer app (0.07%) has the smallest amount of reused code. The average amount of code reused from Stack Overflow for an app is 1.06% and the median is 0.59%.

> Approximately 1.33% of the Stack Overflow posts in our dataset are reused in the examined mobile apps. Moreover, on average 1.06% (0.59% median) of the examined mobile apps' source code is reused from Stack Overflow.

4.5 Case Study Results

This section presents and discusses the results related to our research questions. For each research question, we present the motivation behind the question, the approach, our findings and their implications.

4.5.1 RQ1: Why do Mobile Developers Reuse Code from Stack Overflow?

Motivation: We saw earlier that code reuse does occur from Stack Overflow. One of the first questions that comes to mind is *why* do mobile developers reuse code from Stack Overflow? Answering this question will help us and the research community to better understand the types of activities and tasks which mobile app developers resort to Stack Overflow for. In addition, our analysis will provide some insights into the type of reuse benefits mobile app developers received from Stack Overflow. It is important to note here that our analysis is in the context of code reuse from Stack Overflow for mobile apps and not the general use of Stack Overflow as studied in prior work [162].

Table 8: Reuse categories based on coding of commit messages

Category	**% of Commits**
Enhancing existing code	33.33%
Adding new features	23.70%
Refactoring	12.59%
API usage	11.11%
Fixing bugs	10.37%
Test	0.74%
Other	8.14%

Approach: To answer this research question, we performed a qualitative analysis, where we manually examined the commit messages of the commits that introduced the Stack Overflow code in each of the 22 mobile apps. We observed that the reused code was related to 140 different commits. We printed the message associated with each commit and examined all of them in turn. Five of the commits had commit messages that were unreadable (the message was corrupted), hence, we discarded them from our dataset. Our final dataset contained a total of 135 commit messages that introduced the Stack Overflow code in the 22 selected mobile apps.

To determine the reasons why mobile developers reuse code from Stack Overflow, we applied ground theory [40] to discover and categorize the commits. Two graduate students (1 Ph.D. student and 1 master student) separately open coded each commit message. Each student came up with their own categories by reading and analyzing the commit messages. After the participants completed their classifications, they met and discussed the commit messages that were not consistently classified (i.e., each member grouped these messages in different categories) to reach an agreement. The 135 commits were grouped into seven different groups, which were derived from the examined commit messages. Finally, we used Cohen's Kappa coefficient [43] to evaluate the level of agreement between the two coders. The Cohen's Kappa coefficient is a well-known statistical method that is used to evaluate the inter-rater agreement level for categorical scales. The resulting coefficient is scaled to range 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.

Findings: As a result of our manual classification process, we ended up with seven different categories that the commit messages were grouped into. We observe that mobile developers tend to reuse code from Stack Overflow for different reasons such as: using API, fixing bugs, testing, refactoring, adding new features and enhancing existing code. In some cases, there were commit messages that were not descriptive enough to allow for a clear classification, e.g., the message would simply contain a sequence of digits or only contain one word such as "initial", however, this was a small percentage of the examined commits. Table 8 presents the percentage of commits for each category. We observe

that enhancing existing features and adding new features are the two most common reasons for code reuse from Stack Overflow in mobile apps, accounting for approximately 57% of the examined commits. Additionally, we found that Cohen's Kappa coefficient shows the level of agreement between the two coders to be +0.82, which is considered to be an excellent agreement [64].

Implications: Based on our findings, there are a number of implications for our RQ1. First, we believe that *the Stack Overflow community* will know what mobile developers are using their posts for, so they can provide better support for such activities (e.g., in addition to sharing code, one can point to possible documentation since developers are mostly using the code to implement new features). Other more drastic measures can involve asking users or contributors of the code to provide quality assurance mechanisms (e.g., tests or code reviews) for snippets that are deemed to have a higher probability of reuse. Our findings are also useful for *the mobile developer community* since they will know what tasks other developers are reusing code from Stack Overflow for, e.g., they will know that Stack Overflow may contain relevant resources to help with refactoring. Finally, *the research community* will know in what context mobile app developers reuse code from Stack Overflow, so they can develop techniques and tools to assist with such tasks, e.g., use Stack Overflow to help with testing, though such cases are rare in our dataset.

> Mobile app developers reuse code from Stack Overflow to use APIs, fix bugs, conduct testing, refactor existing code, add new features and enhance existing code. The most common reason for reuse from Stack Overflow is the enhancement of existing code.

4.5.2 RQ2: When in a Mobile App's Lifetime do Developers Reuse Code from Stack Overflow?

Motivation: After examining the various reasons for source code reuse from Stack Overflow, we would like to know when in the project's lifetime mobile app developers tend to reuse code the most. Answering this question helps us better understand at what stage of development mobile developers need the most help and resources. It also tells whether the impact of reuse can only be expected late in the mobile app (in case we find most reuse happens later on) or throughout the project's lifetime.

Approach: To address this research question, we first compute the age of each app in number of days. We then classify the apps in terms of their maturity in terms of days since their first commit. This classification enables us to perform a fair comparison since different projects will have different ages (i.e., lifetime). The distribution of ages of the 22 apps is shown in Figure 6. We use the first and third quantiles to divide the apps into three different maturity groups. As shown in Figure 6, the

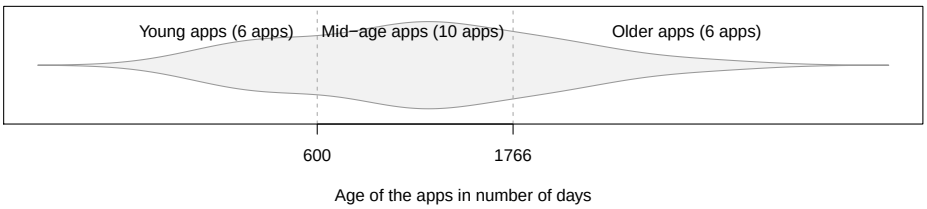
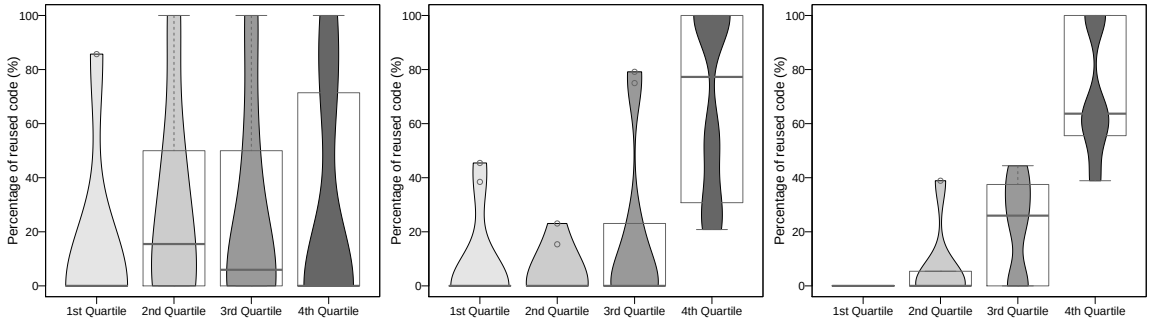


Figure 6: Distribution of the app's age of the 22 examined mobile apps



(a) Young mobile apps (b) Mid-age mobile apps (c) Older mobile apps

Figure 7: Percentage of reused Stack Overflow code per quartile in apps divided based on the age

first quantile is at 600 days and the third quantile is at 1,767 days. Based on this distribution, we divide the apps into 1) young apps ($app's\ age < 600\ days$), 2) mid-age apps ($600\ days =< app's\ age =< 1,767\ days$), and 3) older apps ($app's\ age > 1,767\ days$). Based on this classification, we end up having 6 young apps, 10 mid-age apps and 6 older apps.

After classifying the apps into three groups, i.e., young, mid-age and older apps, we counted the percentage of commits that reuse code in each app. To gain a finer grained view of the reuse, we measured the percentage of reused commits for each app per quartile (i.e., in the first, second, third or fourth quartile). Such a fine grained view can tell us, for example, whether most of the reuse for mid-age apps occurs early (e.g., first quartile) or later on (e.g., fourth quartile) in their lifetime.

We measure the reuse as a percentage (rather than raw number) of commits since the raw number of commits in each app (and each quartile) can vary. To ensure that we only count relevant commits, we remove all merge commits from our calculation of the total commits.

Findings: Figure 7 shows bean plots (with superimposed boxplots) of the percentage of the code reuse from Stack Overflow for the three mobile app groups. Bean plots are useful in presenting the distribution of data, whereas the superimposed box plots highlight key statistics. Figure 7a shows that for younger apps, most code reuse from Stack Overflow occurs in the middle of the apps' lifetime, i.e., second and third quartiles, yet this difference is not statistically significant. To determine

whether there are statistically significant differences between the values in the different quartiles, we perform a one-way analysis of variance (ANOVA), and we could not observe a statistically significant difference. Figure 7b shows that for mid-age apps, most reuse occurred late in the mobile apps' lifetime (i.e., fourth quartile). ANOVA showed that there is a statistically significant difference (*p-value* is <0.05) between the median of the fourth quartile and the medians of the other quartiles. For the older apps, we observe that, once again, most reuse from Stack Overflow happened late in the mobile app lifetime, i.e., the third and fourth quartiles. Also, the use of ANOVA shows that the observations for the third and fourth quartiles are statistically significant, with $p\text{-}value < 0.05$. Our findings indicate that although developers tend to reuse code for feature addition and enhancement, this reuse (feature addition and enhancement) varies based on the age of the app.

Implications: There are a number of implications of our findings in RQ2. For *the Stack Overflow community*, our findings show that it is not just young or immature mobile apps that reuse code from Stack Overflow, rather even more mature or older mobile apps tend to reuse code from Stack Overflow. Hence, proper mechanisms can be developed to tailor results that are returned to users based on the maturity of their mobile app. Alternatively, Stack Overflow could add a mechanism for explicitly rating the source code snippets that have been reused in actual mobile apps. For *the mobile developer community*, our findings show that indeed, other developers resort to Stack Overflow even at later stages in the development of their projects. We believe that developers should link to Stack Overflow posts that they used to help reach their final coded solutions. For *the research community*, our findings can be used to motivate the development of techniques that support maintenance or late-stage development with resources from Stack Overflow.

> Mobile app developers tend to reuse code from Stack Overflow at later stages of the project for mid-age and older mobile apps.

4.5.3 RQ3: Who Reuses Code from Stack Overflow?

Motivation: Prior work has indicated that reusing code is often negatively perceived by developers and associated with less experienced developers [21]. Hence, we wanted to empirically examine who reuses Stack Overflow code amongst mobile developers, e.g., is such code reuse more common among less experienced developers that are looking for quick solutions or is it the experienced developers who resort to code reuse from Stack Overflow?

Approach: To answer this research question, we first measure the developer experience within the mobile app. Similar to prior work [173, 27], we use the developer activity, measured using the number

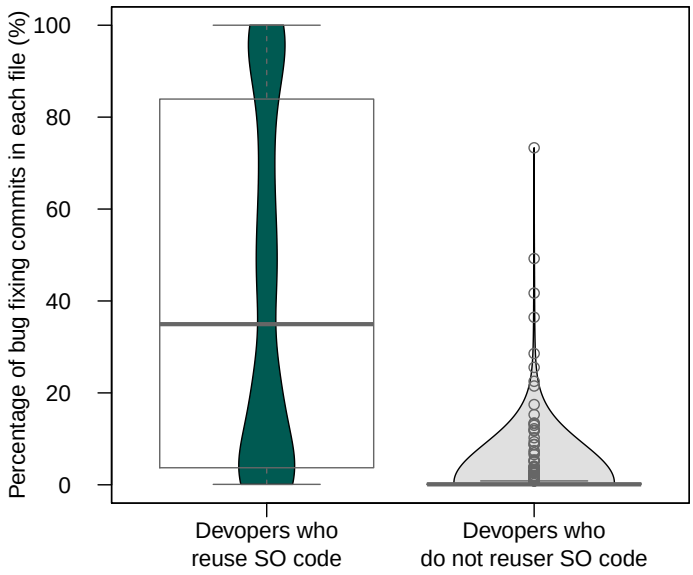
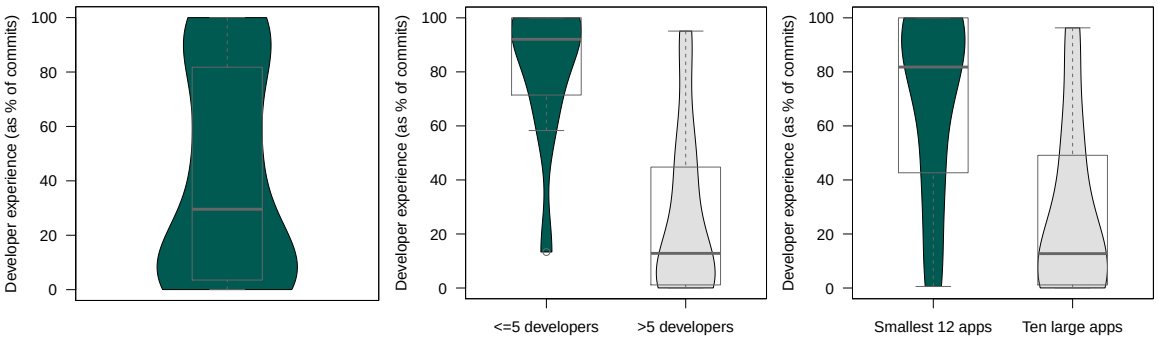


Figure 8: Distribution of the developers' experience for developers who reuse code from Stack Overflow against the rest of developers in the 22 studied mobile apps

of previous commits from the start of the project to the time of code reuse, to determine a developer's experience level. We observed that some of the developers commit from different emails and with a variety of different names. Therefore, we manually examined the names and email addresses of all developers in our dataset and merged similar identities. In total, we found 37 unique developers who reused code from Stack Overflow in the 22 examined mobile apps. Once again, we normalized the experience of the developers and present it as a percentage of the total number of commits in the project (i.e., a developer's experience in a project is measured as $\frac{\#\ commits\ by\ the\ developer}{total\ commits} \times 100$).

Findings: At first, we compare the experience of all mobile app developers who reuse code from Stack Overflow against developers who do not reuse code from Stack Overflow for the 22 studied mobile apps. Figure 8 compares the percentage of developers' experiences for both developers who reuse Stack Overflow code and developers who do not reuse Stack Overflow code. Our study shows that Stack Overflow code reuse is mainly performed by more experienced developers (median equal to 29.56%) than the rest of the developers (median 0.04%) in our 22 studied mobile apps. We also find that the difference between the percentage of developers' experiences in the two groups is statistically significant, with a $p\text{-}value < 0.05$.

Subsequently, we examined in more detail the experience of all mobile app developers (in terms of their commit activity) who reuse code from Stack Overflow. Figure 9a shows the results of our analysis when all apps are grouped together. The figure shows that on median, developers who commit 29.56% of the total commits are the ones who reuse code from Stack Overflow. The box in the box plot also shows wide variation, which in essence makes it difficult to observe any type of



(a) All mobile apps (b) Apps divided based on No. of developers (c) Apps divided based on size (LOC)

Figure 9: Distribution of the developers' experience for developers that reuse code from Stack Overflow

trend.

Thus, we decided to perform an in-depth analysis based on both team size and LOC of the apps since they are two clear factors that can impact the app, i.e., apps developed by smaller teams may follow different development processes compared to apps that are developed by larger teams and larger apps may do more than apps that are smaller in size, hence this was a clear and intuitive way to divide the apps.

We then repeated the same analysis for apps that are developed by small teams ($<= 5$ developers) and apps developed by larger teams (> 5 developers). Figure 9b shows a clear difference among these two different team sizes. From Figure 9b we observe that in apps developed by smaller teams, more experienced developers (median 92.05%) tend to reuse code from Stack Overflow, whereas in apps developed by larger teams, it is the less experienced developers (median 12.83%) who reuse code from Stack Overflow. We also perform the Mann-Whitney test to identify if the observation is statistically significant, which confirms that the difference is statistically significant with the *p-value* being < 0.05.

Next, we considered the size in terms of lines of code of the apps for our analysis. We divided the apps into the top 10 largest and the remaining 12 apps. Figure 9c shows that for the 10 largest apps, the less experienced (median 12.73%) mobile app developers tend to reuse code from Stack Overflow. For the smallest 12 apps, the results shows that more experienced (median 81.78%) developers reuse code from Stack Overflow. Finally, we used the Mann-Whitney test to identify if this difference is statistically significant. We found that the *p-value* is < 0.05, which confirms that the difference is statistically significant.

Implications: There are a number of implications from our findings in RQ3. For the *Stack Overflow community*, our findings can motivate the need to develop techniques that analyze and attach a "safety index" to code snippets since our results provide evidence that for larger teams/apps, less experienced developers tend to reuse code from Stack Overflow. For the *mobile developer community*, our findings show that in larger teams and apps, code developed by junior or less experienced developers needs to be more closely reviewed since it may be code that is reused. Such reuse can also have licensing implications, although this topic is beyond the scope of this chapter. For the *research community*, our results can serve as motivation for the need to perform more fine grained studies on reuse by different types of developers. Additionally, our findings can help shed light on code ownership (since we see who reuses code is also related to the size of the team or project the developer works with), a topic that has received increasing attention lately.

> More experienced mobile app developers reuse Stack Overflow code in smaller teams/apps, whereas less experienced developers reuse Stack Overflow code in larger teams/apps in our dataset.

4.6 Does Code Reuse from Stack Overflow Impact the Quality of Mobile Apps?

Thus far, we have investigated the *how much*, *why*, *when* and *who* questions related to reusing code from Stack Overflow in mobile apps. Our findings showed that the amount of reused Stack Overflow code varies for different mobile apps, that feature additions and enhancements are the main reasons for code reuse, that mid-age and older apps reuse Stack Overflow later in their lifetime, and that more experienced developers reuse code in smaller teams/apps, while less experienced developers reuse code in larger teams/apps.

In addition to these previous findings, we also examine the implications of code reuse from Stack Overflow on the quality of mobile apps. Hence, we examine the bug fixing commits of files that reuse code from Stack Overflow *before* and *after* the reuse occurred. For this analysis as shown in Figure 10, we marked all files that contain reused Stack Overflow code in all of the 22 studied mobile apps. Then, for each file we retrieve all the commits that ever touched the file. We identified the commit that introduces the reused code from Stack Overflow and divided the commits into two groups: commits that occurred *before* and commits that occurred *after* the introduction of the reused code. We used heuristics to classify each commit as a bug fixing or non-bug fixing commit.

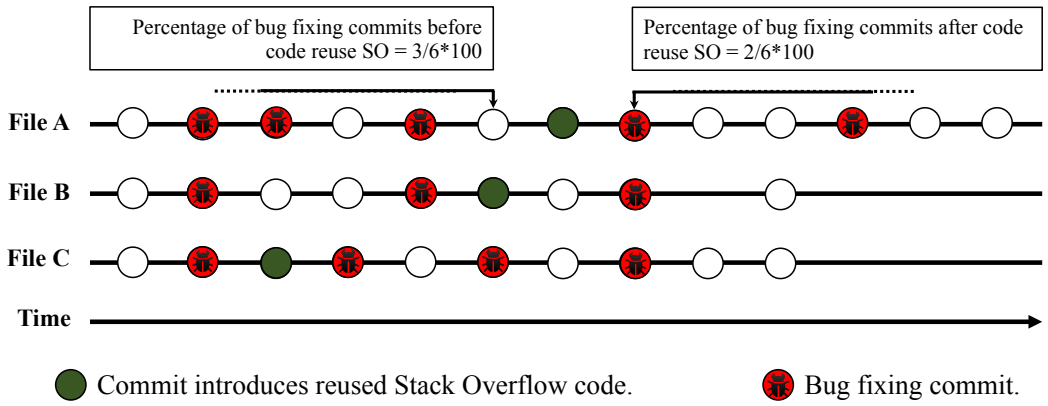


Figure 10: The approach to compute the percentage of bug fixing commits *Before* and *After* reuse per file

Similar to prior work [133, 63], we use a set of keywords to identify bug fixing commits. A commit is identified as a bug fixing commit if its commit message contains one of the following keywords "fix", "bug", "defect", "patch", "error". We then compute the ratio of bug fixing commits by dividing the number of fixing commits by the total number of commits for each file *before* and *after* the introduction of the reused code. Finally, we compare the percentage of bug fixing commits in the two periods, *before* and *after* reusing Stack Overflow code. We compare the percentage of bug fixing commits instead of the raw numbers since Stack Overflow code could be introduced at different times, i.e., we may not have the same total number of changes before and after the introducing Stack Overflow code. Doing this will show if a higher percentage of bugs exists after code is reused from Stack Overflow.

Figure 11 shows the distribution of percentage of bug fixing for all studied files. We observed that the median percentage of bug fixing *before* reusing the Stack Overflow code is 5.96% and the max is 50%. On the other hand, the right box plot represents the percentage of bug fixing *after* reusing Stack Overflow code. We observed that the percentage of bug fixing commits is higher after the code reuse from Stack Overflow (median equal to 19.09%). To determine if this difference is statistically significant, we perform a Mann-Whitney test, which confirmed that the difference is statistically significant with p-value < 0.05.

In addition, we computed the effect size of the difference using Cliff's Delta (d), which is a non-parametric effect size measure for ordinal data. Cliff's d ranges in the interval [-1, 1] and is considered small for $d < 0.33$ (positive as well as negative values), medium for $0.33 \leq d < 0.474$, and large for $d \geq 0.474$. The result shows that the difference has a small effect size (Cliff's $d = 0.225$) when comparing the percentage of bug fixing commit *before* reusing the Stack Overflow code and the percentage of bug fixing commit after the reuse of source code from Stack Overflow.

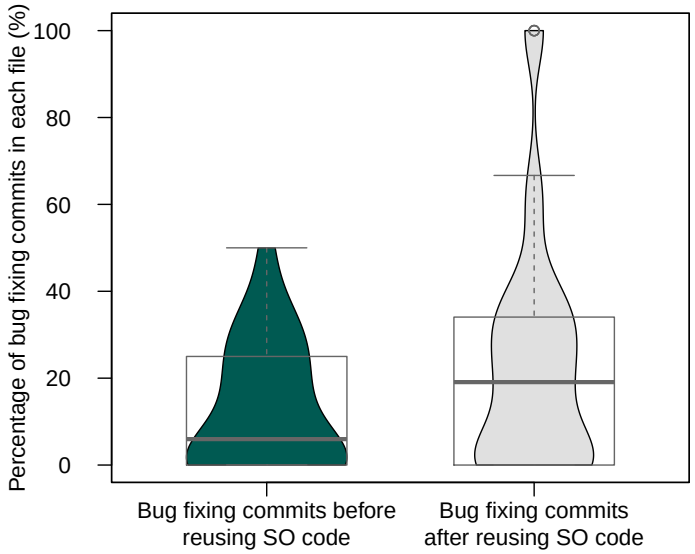


Figure 11: The Percentage of bug fixing commits *Before* and *After* reuse per file

Although this is not a comprehensive study on the impacts of code reuse in mobile apps, we see our aforementioned finding as preliminary evidence that reuse from Stack Overflow may have a negative impact on the quality of mobile apps.

> Files that contain code reused from Stack Overflow have a higher percentage of bug fixing commits after the introduction of reused Stack Overflow code, indicating that reusing code from Stack Overflow may negatively impact the quality of mobile apps.

4.7 Related Work

Chapter 2 discussed work that is related to the use of Stack Overflow as a crowdsourcing platforms. In this section, we presented work that is most related to this chapter. Work related to this chapter research can be divided into three categories: research on the origin of source code and code reuse, work related to the use of Stack Overflow, and work related to code reuse in mobile apps.

4.7.1 Origin of Source Code and Code Reuse

Inoue *et al.* [92] developed a prototype called Ichi Tracker to explore the evolution of a code fragment utilizing online code search engines and code clone detection techniques. The tool accepts a code fragment as input and returns related files containing query code. Using the Ichi Tracker, developers

can identify the origin of a source code fragment or a modified version of the code fragment, including potential license violations. German *et al.* [72] examined source code migration across three different systems (Linux, FreeBSD and OpenBSD) and its legal implications. They tracked reused code fragment using clone detection methods. Their result showed that code migration did occur between these systems. Additionally, the copying tended to be performed without violating the license terms. Davies *et al.* [48] proposed a signature-based matching technique to determine the origin of code entities. They found that their technique can be utilized to identify security bugs in the reused libraries. Kawamitsu *et al.* [101] proposed a technique to automatically detect source code reuse between two software repositories at the file level. It is based on measuring the similarity between two source files and using the commit time to identify the original source file revision. They found that in some instances developers did not record the version of the reused file.

Our work differs from this existing research in several aspects. First, the main focus of our work is on reuse of source code from Stack Overflow in the context of mobile apps. Secondly, while our work is similar to some existing work (e.g., [72]), in that we also detect source code reuse using a clone detection technique, our focus is analyzing why, when and who reuses code from Stack Overflow. This is in contrast to existing work, where the focus had been on detecting the origin of source code, license violations and code migration.

4.7.2 Work Related to the Use of Stack Overflow

Barua *et al.* [19] proposed a semi-automatic approach to study general topics discussed on Stack Overflow by developers. They found that web and mobile development are the most popular topics. Rosen and Shihab [162] used Stack Overflow to determine what mobile developers on Stack Overflow ask about. They found that questions posted on Stack Overflow cover almost all issues related to the development of mobile apps, and that app distribution and user interface questions are the most viewed. We consider the prior results of these studies as evidence that Stack Overflow is a very popular source of programming-related knowledge and source code, which served as a motivation for our work. In our study however, we focus on the reuse of source code from Stack Overflow in mobile apps. More specifically we propose an approach to identify reused code from Stack Overflow and answer a number of research questions such as, why mobile app developers reuse code from Stack Overflow, when in a mobile app's lifetime developers reuse code from Stack Overflow, what is the experience of these developers, and the potential impact of reusing code from Stack Overflow on software quality. To the best of our knowledge, our study is the first to empirically study code reuse from Stack Overflow in mobile apps.

Several other work has attempted to recommend code snippets from Stack Overflow. Cordeiro *et al.* [44] indexed the Stack Overflow data dump to retrieve associated discussion. They extract

keywords from exception traces in Eclipse to automatically suggest Q&A threads from Stack Overflow to developers. Ponzanelli *et al.* [151] developed a tool called Prompter that recommends related code from Stack Overflow. Prompter generates a search query from the developers' programming context in the IDE and searches Stack Overflow to recommend code snippets. Rahman *et al.* [157], proposed SurfClipse that provides immediate assistance to developers when they encounter runtime errors or exceptions. It exploits the code terms to search in three search engines and Stack Overflow, and shows the collected results in the Eclipse IDE. Wang *et al.* [198], devised an approach to build a bidirectional link between the Android issue tracker and Stack Overflow discussions to facilitate the knowledge sharing between two separated communities. They exploit the semantic similarity with temporal-locality to effectively establish the link. Once such a link is established, it allows contributors (developers and users) in the two communities to gain the benefit of knowledge sharing. In the previous chapter, we examined why developers use Stack Overflow and found, amongst other findings, that developers do reuse code from Stack Overflow [13].

In contrast to existing research that mainly focused on recommending code snippets form Stack Overflow to assist developers, our work's main goal is to perform an exploratory study on code that is reused from Stack Overflow in mobile apps.

4.7.3 Work Related to Mobile Apps

Several studies on mobile software development have been published in recent years. Syer *et al.* [183] compared the source code of BlackBerry and Android apps along three aspects, source code, code dependencies and code churn. They found that BlackBerry apps are larger and rely more on third party libraries, whereas Android apps have less number of files and rely mostly on the Android platform. Ruiz *et al.* [165] compared the extent of code reuse in the different categories of Android apps. They observed that around 23% of the classes inherit from one of the base classes in the Android API and 27% of the classes inherit from a domain specific base class. Furthermore, they found that 217 mobile apps are completely reused by another mobile app. Minelli and Lanza [130] proposed a software analytics platform called SAMOA that has been utilized to analyze 20 Android apps. Their analysis showed that mobile apps intensely depend on the usage of external APIs. Chen et al. [42] developed an approach, based on geometry characteristics, that can detect clones between mobile apps. Our work differs from this related work in that we focus on code reuse from Stack Overflow and not across mobile apps or from APIs.

4.8 Threats to Validity

In this section, we discuss the limitations of our study and the applicability of the results.

4.8.1 Threats to Internal Validity

In what follows, we discuss confounding factors that potential could have influenced our study results. First, to identify the reused code from Stack Overflow, we use the CCFinder clone detection tool. Hence, we are limited by the accuracy of CCFinder in detecting Type-1 and Type-2 clones. In some cases, we may have missed instances of code reuse if CCFinder did not flag the code as a clone. To help mitigate this issue, we manually investigated some of the clones and in all cases the flagged clones were reused code from Stack Overflow. In addition, we use CCfinder with the default configuration parameters. Changing these parameters may lead to different results. When we identify reused code, we ensure that the date of the Stack Overflow post is prior to the commit date. In certain cases, the code in the project may not have actually originated from Stack Overflow, if for example the developer takes a long time to commit. Furthermore, we found there are some cases where Stack Overflow posts are reused more than once that developers may copy the source code snippets between mobile apps.

To determine why developers reuse code form Stack Overflow, we manually analyzed the commits. Like any human activity, the categorization may be prone to human error or bias. To alleviate this threat, we had two students separately code the commits messages and come to an agreement on any discrepancies. We also computed Cohen's Kappa to evaluate the inter-rater agreements, which showed excellent inter-rater agreement (Cohen's Kappa value of +0.82). In addition, we use the commit message that introduced the code as an indicator of the reuse. However, a commit may contain the Stack Overflow code combined with other code. In this analysis, we consider a commit as a single unit of change with the reused code being part of the complete unit, hence, the reason should be the same for all the modified code. Similar to prior work, e.g., [173, 27], we use the number of previous commits to measure experience. In some cases, the number of previous commits may not be representative of the actual experience of a developer. Furthermore, given that we manually identified the unique developers by comparing names and email addresses of all developers, it is possible that we potentially wrongly identified a developer as unique. The analysis in RQ2 is based on the size of the apps and number of developers, which may not present all the development features of apps.

4.8.2 Threats to External Validity

These threats concern the possibility that our results may not be generalizable. In this study, we focus on Stack Overflow, which is one of many Q&A websites, hence, our results may not generalize to reuse from other Q&A websites. In addition, we examined 22 Android Apps in which code reuse occurred, hence, our findings may not generalize to other apps, especially apps that are not open

source. Finally, we studied source code reuse from Stack Overflow in the context of mobile apps, which may not be generalizable to other application domains.

4.9 Chapter Summary

In this chapter, we investigated the reuse of code from Stack Overflow in mobile apps. We conducted an exploratory study using 22 Android apps. We found that the amount of reused Stack Overflow code varies among mobile apps, that feature additions and enhancements are the main reasons for code reuse from Stack Overflow, that mid-age and older apps reuse Stack Overflow code later in the lifetime and that more experienced developers reuse code in smaller teams/apps, while less experienced developers reuse code in larger teams/apps. We also examined the potential implications of such Stack Overflow code reuse and found that this code reuse can be the cause of additional bugs, which are introduced in these apps. The results from our studies not only provide some valuable insights into the potential reuse of code from crowdsourcing in software systems but also provide insights into the challenges associated with code reuse from crowdsourcing platforms such as Stack Overflow when applied in current software development.

The focus of this chapter is to study the impact of reusing source code from crowdsourcing platforms, namely Stack Overflow. However, Stack Overflow is not the only crowdsourcing platform that can provide developers with such knowledge. In the next chapter, we, therefore, shift our focus to examine other crowdsourcing platforms, namely *npm* and *PyPI* to study the type of knowledge they provide.

Chapter 5

Examining the Type of Constructed Knowledge on Crowdsourcing Platforms

Code reuse has traditionally been encouraged since it enables one to avoid re-inventing the wheel. However, due to a recent incident where a trivial package led to the breakdown of some of the most popular web applications such as Facebook and Netflix, some questioned such reuse. Reuse of such trivial packages is particularly prevalent in emerging platforms such as Node.js, one of the most popular crowdsourced platforms today. However, to date, there is no study that examines the reason why developers reuse trivial packages. In this chapter, we study two large package management platforms *npm* and *PyPI*. For this study, we mine more than 230,000 *npm* packages and 38,000 JavaScript applications and more than 63,000 *PyPI* packages and 14,000 Python applications in order to study the prevalence of trivial packages. We found that trivial packages are common and are gaining in popularity, making up between 16.8% to 10.5% of the studied platforms. We conducted a survey with 125 developers who use trivial packages to understand the reasons and drawbacks of their use. Our survey revealed that trivial packages are perceived by developers to be well implemented and tested pieces of code. However, developers are concerned about maintaining and the risks of breakages due to the extra dependencies such trivial packages introduce. To objectively verify the survey results, we empirically validate the most cited reason for use of these trivial packages, as well as the drawbacks of their usage. We find that contrary to developers' beliefs, around 45% of *npm* and 51% *PyPI* trivial packages even have tests. However, trivial packages appear to be 'deployment tested' and to have similar test, usage and community interest as non-trivial packages. On the other

hand, we found that 11.5% and 2.9% of the studied trivial packages have more than 20 dependencies in *npm* and *PyPI*, respectively. We conclude that developers should be careful about which trivial packages they decide to use.

5.1 Introduction

Code reuse, in the form of combining related functionalities in packages, is often encouraged due to its multiple benefits. In fact, prior work showed that code reuse can reduce the time-to-market, improve software quality and boost overall productivity [22, 114, 134, 8]. Therefore, it is no surprise that emerging platforms such as Node.js encourage reuse and do everything possible to facilitate code sharing, often delivered as packages or modules that are available on package management platforms, such as the Node Package Manager (*npm*) [144, 33].

However, it is not all good news. There are many cases where code reuse has had negative effects, leading to an increase in maintenance costs and even legal actions [126, 146, 92, 12]. For example, in a recent incident through code reuse of a Node.js package called left-pad, which was used by Babel, caused interruptions to some of the largest Internet sites, e.g., Facebook, Netflix, and Airbnb. Many referred to the incident as the case that 'almost broke the Internet' [118, 200]. That incident lead to many heated discussions about code reuse, sparked by David Haney's blog post: *"Have We Forgotten How to Program?"* [80].

While the real reason for the left-pad incident was that *npm* allowed authors to unpublish packages (a problem which has in the meanwhile been resolved [145]), it still raised awareness of the broader issue of taking on dependencies for trivial tasks that can be easily implemented [80]. Since then, there have been many discussions about the use of trivial packages. Loosely defined, *a trivial package is a package that contains code that a developer can easily code him/herself and hence, is not worth taking on an extra dependency for.* Many developers agreed with Haney's position, which stated that every serious developer knows that 'small modules are only nice in theory' [34], suggesting that developers should implement such functions themselves rather than taking on dependencies for trivial tasks. Other work showed that *npm* packages tend to have a large number of dependencies [52, 53] and highlighted that developers need to use caution since some dependencies can grow exponentially [23]. In fact, in our dataset, we found that more than 11% of the *npm* trivial packages have more than 20 dependencies.

So, the million dollar question is "why do developers resort to using a package for trivial tasks, such as checking if a variable is an array?" At the same time, other questions regarding how prevalent trivial packages are and what the potential drawbacks of using these trivial packages have remained unanswered by existing research. To address these questions, we performed an empirical

study involving more than 230,000 *npm* packages and 38,000 JavaScript applications and 63,000 *PyPI* packages and 14,000 Python applications to better understand why developers resort to using trivial packages. Our empirical study is qualitative in nature and is based on survey results from 125 JavaScript and Python developers. We also quantitatively validate the most commonly developer-cited reason and drawback related to the use of trivial packages.

Since, to the best of our knowledge, this is the first study to examine why developers use trivial packages, we first propose a definition of what constitutes a trivial package, based on feedback from JavaScript developers. We also examine how prevalent trivial packages are in *npm* and *PyPI* and how widely they are used in JavaScript and Python applications. Our findings indicate that:

The definition of trivial packages is the same in JavaScript and Python. The developers from the two different ecosystems tended to have the same definition of trivial packages. Confirming our definition of *npm* trivial packages, we find that *PyPI* trivial packages are packages that have $\leq$ 35 LOC and a McCabe's cyclomatic complexity ≤ 10.

Trivial packages are common and popular in both, *npm* and *PyPI* management platforms. Of the 231,092 *npm* and 63,912 *PyPI* packages in our dataset, 16.8% and 10.6% of them are trivial packages. Moreover, of the 38,807 JavaScript and 14,717 Pythons applications on GitHub, 10.9% and 6.9% of them directly depend on one or more trivial packages.

JavaScript and Python developers differ in their perception of trivial packages. Only 23.9% of JavaScript developers considered the use of trivial packages as bad, whereas, 70.3% of Python developers consider the use of trivial package as a bad practice.

Trivial packages provide well implemented and tested code and increase productivity. Developers believe that trivial packages provide them with well implemented/tested code and increase productivity. At the same time, the increase in dependency overhead and the risk of breakage of their applications are the two most cited drawbacks.

Developers need to be careful which trivial packages they use. Our empirical findings show that many trivial packages have their own dependencies. In *npm*, 43.7% of *npm* of trivial packages have at least one dependency and 11.5% of *npm* trivial packages have more than 20 dependencies. In *PyPI*, 36.8% of *PyPI* trivial packages have more than one dependency, and only 2.9% have more than 20 dependencies. Our study also reveals that the problem of dependency drawback in *PyPI* is not as bad as in *npm* since more than 63% of the *PyPI* trivial packages does not have any dependencies.

To facilitate the replicability of our work, we make our dataset and the anonymized developer responses publicly available [11].

5.1.1 Organization of the Chapter

The remainder of this chapter is organized as follows. Section 5.2 provides the background and introduces our datasets. Section 5.3 presents how we determine what a trivial package is. Section 5.4 examines the prevalence of trivial packages and their use in JavaScript and Python applications. Section 5.5 presents the results of our developer surveys, presenting the reasons and perceived drawbacks for developers who use trivial packages. Section 5.6 presents our empirical validation of the most commonly cited reason for and drawback of using trivial packages. The implications of our findings are noted in section 5.7. We discuss the related works in section 5.8, the limitations of our study in section 5.9, and present our conclusions in section 5.10.

5.2 Background and Datasets

In this section, we provide background on the two studied package management platforms, *npm* and *PyPI*. We also provide an overview of the dataset collected and used in the rest of our study.

5.2.1 Node Package Manager (*npm*)

JavaScript is used to write client and server side applications. Its popularity has steadily grown, thanks to popular frameworks such as Node.js and an active developer community [33, 201]. JavaScript projects can be classified into two main categories: *JavaScript packages* that are used in other projects or *JavaScript applications* that are used as standalone software. The Node Package Manager (*npm*) provides tools to manage JavaScript packages. *npm* is the official package manager for JavaScript and its registry contains more than 250,000 packages at the time of writing this work [79].

To perform our study, we gather two datasets from two sources. We obtain JavaScript packages from the *npm* registry and applications that use *npm* packages from GitHub.

***npm* Packages:** Since we are interested in examining the impact of 'trivial packages', we mined the latest version of all the JavaScript packages from *npm* as of May 5, 2016. For each package we obtained its source code from GitHub. In some cases, the package publisher did not provide a GitHub link, in which case we obtained the source code directly from *npm*. In total, we mined 252,996 packages.

GitHub JavaScript Applications: We also want to examine the use of the packages in JavaScript applications. Therefore, we mined all of the JavaScript applications on GitHub. To ensure that we are indeed only obtaining the applications from GitHub, and not *npm* packages, we compare the URL of the GitHub repositories to all of the URLs we obtained from *npm* for the packages. If a URL from GitHub was also in *npm*, we flagged it as being an *npm* package and removed it from the application

Table 9: Background of Participants in the Two Surveys to Determine Trivial Packages.

npm				***PyPI***			
Experience in JavaScript	#	**Developers' Position**	#	**Experience in Python**	#	**Developers' Position**	#
<1	2	Undergrad Student	2	<1	1	Undergrad Student	0
2 - 3	3	Graduate Student	8	2 - 3	3	Graduate Student	9
3 - 5	1	Professional Developer	2	3 - 5	2	Professional Developer	4
>5	6	-	-	>5	7	-	-
Total	**12**	**Total**	**12**	**Total**	**13**	**Total**	**13**

list. To determine that an application uses *npm* packages, we looked for the 'package.json' file, which specifies (amongst others) the *npm* package dependencies used by the application.

To eliminate dummy applications that may exist in GitHub, we choose non-forked applications with more than 100 commits and more than 2 developers. We performed this filtering for both, the JavaScript and Python datasets. Similar filtering criteria were used in prior work by Kalliamvakou *et al.* [96]. In total, we obtained 115,621 JavaScript applications and after removing applications that did not use the *npm* platform, we were left with 38,807 applications.

5.2.2 Python Package Index (*PyPI*)

PyPI is the official package management platform for the Python programming language. Python is one of the most popular programming language today, mainly due to its storng community support and versatility, i.e., Python is used in many different domains from game development to server side applications [195, 159]. Once again, we distinguish between *Python packages*, which are used in Python applications and standalone *Python applications*, which typically use Python packages. Similar to the case of JavaScript, we gather two datasets from two sources to perform our study. We obtain Python packages from the *PyPI* registry and applications that use *PyPI* packages from GitHub.

***PyPI* Packages:** We also collected the latest versions of the Python packages from *PyPI* in order to determine which packages are 'trivial packages'. *PyPI* contains around 118,324 packages [112], as of September 3, 2017. n total, we were able to obtain 116,905 packages from the *PyPI* registry since some package did not exist anynmore.

GitHub Python Applications: To examine the usage of 'trivial packages' in Python applications, we mined all of the Python applications hosted on GitHub. We followed the same aforementioned

process used to gather JavaScript applications, to ensure that we are indeed only obtaining the Python applications from GitHub, and not *PyPI* package repositories. In a nutshell, we compare the URL of the GitHub repositories to the URLs we obtained from *PyPI* for the packages. If a URL from GitHub was also in *PyPI*, we flagged it as being an *PyPI* package and removed it from the application list. In total, we obtained 14,717 Python applications that are hosted on GitHub.

5.3 What are Trivial Packages Anyway?

Although what a trivial package is has been loosely defined in the past (e.g., in blogs [84, 81]), we want a more precise and objective way to determine trivial packages. To determine what constitutes a trivial package, we conducted two separate surveys one for each of the studied package management platforms (*npm* and *PyPI*). We mainly asked participants what they considered to be a trivial package and what indicators they used to determine if a package is trivial or not. We conducted two different surveys since: 1) the two studied packages management platforms serve different programming languages, 2) developers from the two package management platforms have different perspective of what consider to be 'trivial packages'.

For each package management platforms (*npm* and *PyPI*), we devised an online survey that presented the source code of 16 randomly selected packages that range in size between 4 - 250 JavaScript/Python lines of code (LOC). Participants were asked to 1) indicate if they thought the package was trivial or not and 2) specify what indicators they use to determine a trivial package. We opted to limit the size of the selected packages in the surveys to a maximum of 250 JavaScript/Python LOC since we did not want to overwhelm the participants with the review of excessive amounts of code.

We asked the surveys participants to indicate trivial packages from the list of packages provided. We provided the surveys participants with a loose definition of what a trivial package is, i.e., a package that contains code that they can easily code themselves and hence, is not worth taking on an extra dependency for. Figure 12 shows an example of a trivial JavaScript package, called *is-Positive*, which simply checks if a number is positive. The surveys questions were divided into three parts: 1) questions about the participant's development background, 2) questions about the classification of the provided packages, and 3) questions about what indicators the participant would use to determine a trivial package. For the *npm* survey, we sent the survey to 22 developers and colleagues that were familiar with JavaScript development and received a total of 12 responses. We also sent the *PyPI* survey to 18 developers and colleagues that were familiar with Python development and received a total of 13 responses. It is important to note that we sent the two surveys to different groups of developers, to make sure that the participants in one survey are not

baised through their experience of participating in the other (i.e., first) survey.

```
module.exports = function (n) {
return toString.call(n) === '[object Number]' && n > 0;
};
```

Figure 12: Package is-Positive on *npm*

Participants Background and Experience: The first four columns of Table 9 show the background of participants in the *npm* survey. Of the 12 respondents, 2 are undergraduate students, 8 are graduate students, and 2 are professional developers. Ten of the 12 respondents have at least 2 years of JavaScript experience and half of the participants have been developing with JavaScript for more than five years.

The last four columns of Table 9 show the background of participants in the *PyPI* survey. Of the 13 participants in this survey, 9 identified themselves as graduate students and 4 as professional developers working in industry; 7 participants had more than 5 years of Python development experience, 2 respondents had between 3 to 5 years, 3 others had 2 to 3 years of experience, and finally one person had less than 1 year of Python practice. We were happy to have the majority of our respondents be well-experienced with Python.

Result: We asked participants of the two surveys to list what indicators they use to determine if a package is trivial or not and to indicate all the packages that they considered to be trivial. Of the 12 participants in the *JavaScript* survey, 11 (92%) state that the complexity of the code and 9 (75%) state that size of the code are indicators they use to determine a trivial package. Another 3 (20%) mentioned that they used code comments and other indicators (e.g., functionality) to indicate if a package is trivial or not. The results of the *Python* survey reveal that 9 (69%) of the developers use size of the code and 9 (69%) of them use complexity of the code as the main indicators to determine trivial packages. Another 7 (54%) of the participants stated that they use source code comments to determine trivial Python packages and 3 (23%) of the participants mentioned some other indicators that they can use to identify a trivial package. For example one participant related a trivial Python package as *"If it's only one function"*.

Since it is clear that size and complexity are the most common indicators of trivial packages and they are a universial measure that can be measued for both, Javascript and Python, we use these two measures to determine trivial packages. It should be mentioned that participants could provide more than 1 indicator, hence the percentages above sum to more than 100%.

Next, we analyze all of the packages that were marked as trivial from the two surveys. Our main

goal of this analysis is to find which values of the size and complexity metrics are indicative of trivial packages.

***npm* Survey Responses:** In total, we received 69 votes for the 16 packages. We ranked the packages in ascending order, based on their size, and tallied the votes for the most voted packages. We find that 79% of the votes consider packages that are less than 35 lines of code to be trivial. We also examine the complexity of the packages using McCabe's cyclomatic complexity, and find that 84% of the votes marked packages that have a total complexity value of 10 or lower to be trivial. It is important to note that although we provide the source code of the packages to the participants, we do not explicitly provide the size or the complexity of the packages to the participants to not bias them towards any specific metrics.

***PyPI* Survey Responses:** we received 89 votes for the 16 packages. Similar to the case of *npm*, we ranked the packages in ascending order, based on their size, and tallied the votes for the most voted packages. We find that 76.4% of the votes consider packages that are equal or less than 35 lines of code to be trivial. We also examine the complexity of the packages using McCabe's cyclomatic complexity, and find that 79.8% of the votes marked packages that have a total complexity value of 10 or lower to be trivial Python package. Similar to *npm*, we also did not provide any metric values for the packages to avoid bias.

Based on the aforementioned findings, we used the two indicators JavaScript/Python LOC $\leq$ 35 and complexity $\leq$ 10 to determine trivial packages in our dataset. Hence, we define trivial JavaScript/Python packages as $\{X_{LOC} \leq 35 \cap X_{Complexity} \leq 10\}$, where X_{LOC} represents the JavaScript/Python LOC and $X_{Complexity}$ represents McCabe's cyclomatic complexity of package X. Although we use the aforementioned measures to determine trivial packages, we do not consider this to be the only possible way to determine trivial packages.

Our study shows that developers indicated that trivial packages for the two studied package management platforms (*npm* and *PyPI*) have similar definition. As our two surveys show, size and complexity are commonly used measures to determine if a package is trivial for both JavaScript and Python programming languages. Based on our analysis, JavaScrip and Python packages that have $\leq$ 35 LOC and a McCabe's cyclomatic complexity $\leq$ 10 are considered to be trivial packages.

5.4 How Prevalent are Trivial Packages?

In this section, we want to know how prevalent trivial packages are. We examine prevalence from two aspects: the first aspect is from package management platforms (*npm* and *PyPI*) perspective, where we are interested in knowing how many of the packages on these two package management platforms are trivial. The second aspect considers the use of trivial packages in JavaScript and Python applications.

5.4.1 How Many of *npm*'s & *PyPI*'s Packages are Trivial?

***npm*:** We use the two measures, LOC and complexity, to determine trivial packages, which we now use to quantify the number of trivial packages in our dataset. Our dataset contained a total of 252,996 *npm* packages. For each package, we calculated the number of JavaScript code lines and removed packages that had zero LOC, which removed 21,904 packages. This left us with a final number of 231,092 packages. Then, for each package, we removed test code since we are mostly interested in the actual source code of the packages. To identify and remove the test code, similar to prior work [74, 189, 210], we look for the term "test" (and its variants) in the file names and file paths.

Out of the 231,092 *npm* packages we mined, 38,845 (16.8%) packages are trivial packages. In addition, we examined the growth of trivial packages in *npm*. Figure 13 shows the percentage of trivial to all packages published on *npm* per month. We see an increasing trend in the number of trivial packages over time and approximately 15% of the packages added every month are trivial packages. We investigated the spike around March 2016 and found that this spike corresponds to the time when *npm* disallowed the un-publishing of packages [145].

***PyPI*:** For the *PyPI* dataset, we are also interested in discerning the trivial packages from the others in terms of LOC and complexity. For such, we mined the 116,905 available packages on the *PyPI* applications. We got all the 116,905 packages from *PyPI* register. However, a package on *PyPI* could be released/distributed in different formats and we were not able to process them. We found that 42,242 of *PyPI* packages are platform exclusive (e.g., windows .exe or mac .dmg) or are corrupted compressed .gz files that we could not analyzed. This process left us with 74,663 *PyPI* packages which we measure their LOC and complexity. We then remove packages that had zero or one LOC, which removed another 10,751 packages. We also discarded test files similar to previous work [74, 189, 210], we look for the term "test" (and its variants) in the file names and file paths.

Our analysis reveals that out of the 63,912 *PyPI* packages we analyzed, 6,759 (10.6%) packages are trivial packages in the *PyPI* package management platform. We again examined the growth of trivial packages in *PyPI*. Figure 14 shows the percentage of trivial to all packages published on

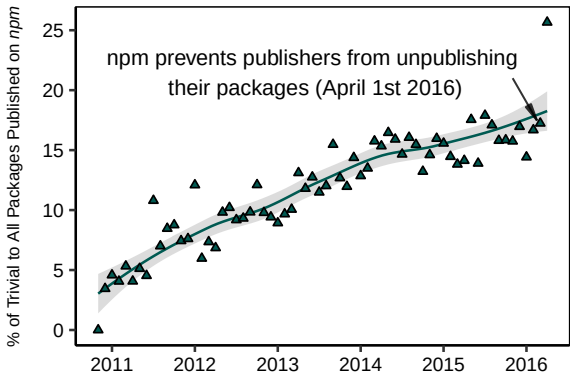


Figure 13: Percentage of Published Trivial Packages on *npm*.

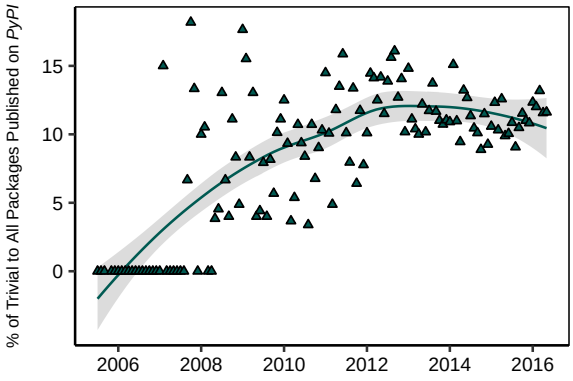


Figure 14: Percentage of Published Trivial Packages on *PyPI*.

PyPI per month. We see an increasing trend in the number of trivial packages over time and approximately 8.6% of the packages added every month are trivial packages.

> Trivial packages make up 16.8% of the studied *npm* packages and 10.6% of the *PyPI* packages. These trivial packages make a non-negligible portion of the packages in the two ecosystems.

5.4.2 How Many Applications Depend on Trivial Packages?

JavaScript Applications: Just because trivial packages exist on *npm*, it does not mean that they are actually being used. Therefore, we also examine the number of applications that use trivial packages. To do so, we examine the package.json file, which contains all the dependencies that an application installs from *npm*. However, in some cases, an application may install a package but not use it. To avoid counting such instances, we parse the JavaScript code of all the examined applications and use regular expressions to detect the require dependency statements, which indicates that the application actually uses the package in its code[1]. Finally, we measured the number of packages that are trivial in the set of packages used by the applications. Note that we only consider *npm* packages since it is the most popular package manager for JavaScript packages and other package managers only manage a subset of packages (e.g., Bower [35] only manages front-end/client-side frameworks, libraries and modules). We find that of the 38,807 applications in our dataset, 4,256 (10.9%) directly depend on at least one trivial package.

Python Applications: Similar to the case of Javascript, we also analyzed the Python applications that include depend on trivial packages. In contrast to JavaScript's availability of a 'packages.json'

[1]Note that if a package is required in the application, but does not exist, it will break the application.

Table 10: Position and development experience of survey respondents.

npm				***PyPI***			
Experience in JavaScript	#	**Developers' Position**	#	**Experience in Python**	#	**Developers' Position**	#
1 - 3 years	7	Industrials	68	1 - 3 years	0	Industrials	27
>3 - 5 years	14	Independent	15	>3 - 5 years	3	Independent	4
>5 years	67	Casual	2	>5 years	34	Casual	1
-	-	Other	3	-	-	Other	5
Total	**88**	**Total**	**88**	**Total**	**37**	**Total**	**37**

file, analyzing Python applications presents some challenges to fully identify a given script's dependency set for the reasons described previously on Section 5.4.1. We statically parse the source code after relevant "import" like clauses, along with other statements that allow for verifying that the packages are effectively being put in use (i.e., the package is both supposed to be installed and its functions/definitions are indeed being called, rather than merely being just imported and not used). To facilitate this analysis, we use the popular `snakefood` [29] tool, which generates dependency graphs from Python code. Our analysis showed that out of the 14,717 examined Python applications, 1,024 (6.9%) were found to depend on one or more trivial *PyPI* package.

> Of the 38,807 JavaScript applications in our dataset, 10.9% of them depend on at least one trivial package. Out of the 14,717 examined Python applications, 1,024 were found to depend on trivial *PyPI* packages (6.95% of the selected GitHub applications)

5.5 Survey Results

We surveyed developers to understand the reasons for and the drawbacks of using trivial packages. We use a survey because it allows us to obtain first-hand information from the developers who use these trivial packages. In order to select the most relevant participants, we sent out the survey to developers who use trivial packages. We used Git's `pickaxe` command on the lines that contain the required dependency statements in the JavaScript and Python applications. Doing so helped us identify the name and email of the developer who introduced the trivial package dependency.

Survey Participants: To mitigate the possibility of introducing misunderstood or misleading questions, we initially sent the survey to two developers and incorporated their minor suggestions

to improve the survey. For *npm* participants, we sent the survey to 1,055 JavaScript developers from 1,696 applications. To select the developers, we ranked them based on the number of trivial packages they use. We then took a sample of 600 developers that use trivial packages the most, and another 600 of those that indicated the least use of trivial packages. The survey was emailed to the 1,200 selected developers, however, since some of the emails were returned for various reasons (e.g., the email account does not exist anymore, etc.), we could only reach 1,055 developers. We also sent the survey to all Python developers after filtering out the invalid and duplicated developers' emails. We successfully sent the survey to 460 Python developers that introduce trivial Python packages from *PyPI* in 1,024 Python applications in our dataset.

The survey listed the trivial package and the application that we detected the trivial package in. In total, we received 125 developer responses. First, we received 88 responses to our survey from the *JavaScript* developers, which translates to a response rate of 8.3%. Our survey response rate is in line with, and even higher, than the typical 5% response rate reported in questionnaire-based software engineering surveys [176]. The left part of Table 10 show the JavaScript experience and the position of the developers. Of the 88 respondents, 83 of them identified as developers working either in industry (68) or as a full time independent developers (15). The remaining 5 identified as being a casual developers (2) or other (3), including one student and two developers working in executive positions at *npm*. As for the development experience of the survey respondents, the majority (67) of the respondents have more than 5 years of experience, 14 have between 3-5 years and 7 have 1-3 years of experience.

Second, we received 37 survey responses from the *Python* developers, yielding a response rate of 8.04%, which is again in accordance with what is supposedly been observed on other studies in the software engineering domain [176]. The right part of Table 10 shows the Python experience and position of the developers. 27 of the respondents refer themselves as a developers working in the industry and 4 developers identified themselves as a full time independent developers. The reset of the respondents are identified as being a casual developers (1) or other (5) including researchers and students. Regarding the development experience of the survey respondents, the vast majority of the respondents (92%) identified themselves to have more than five years of Python development experiences. Only 3 respondents identified themselves to have development experience in Pythons range between more than 3 to five years.

The fact that most of the respondents are experienced JavaScript and Python developers gives us confidence in our surveys responses.

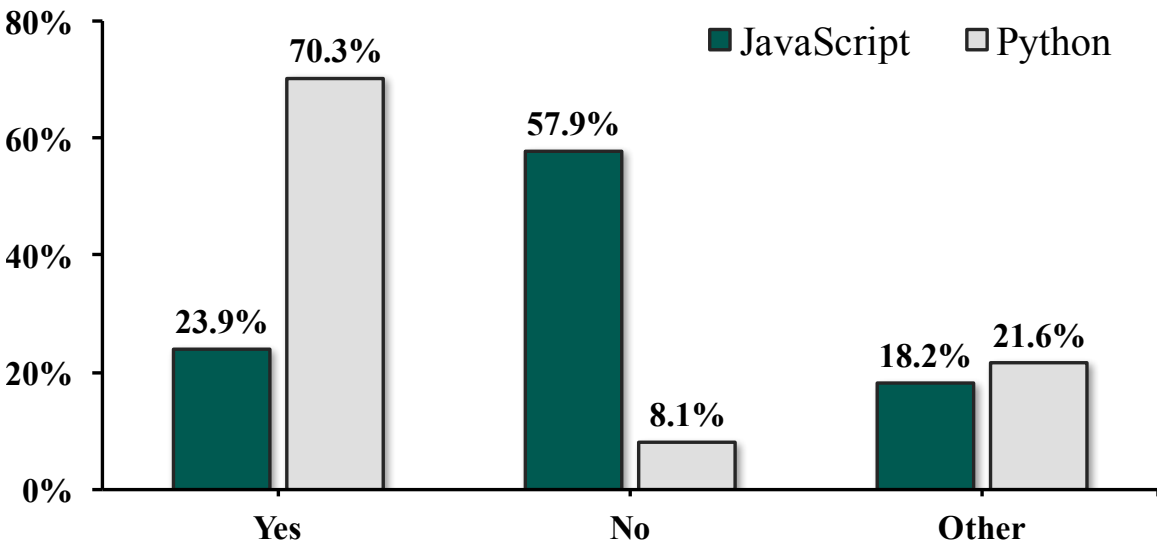


Figure 15: Developer responses to the question "is using a trivial package bad?". Most JavaScript developers answered no, whereas most Python developers answered yes.

5.5.1 Do Developers Consider Trivial Packages Harmful?

The first question of our survey to the participants is: "Do you consider the use of trivial packages as bad practice?" The reason to ask this question so bluntly is that it allows us to gauge, in a very deterministic way, how the developers felt about the issue of using trivial packages. We provided three possible replies, Yes, No or Other in which case they were provided with a text box to elaborate. Figure 15 shows the distribution of responses from both JavaScript and Python developers. Of the 88 JavaScript participants, 51 (57.9%) stated that they do NOT consider the use of trivial packages as bad practice. Another 21 (23.9%) stated that they indeed think that using trivial package is a bad practice. The remaining 16 (18.2%) stated that it really depends on the circumstances, such as the time available, how critical a piece of code is, and if the package used has been thoroughly tested.

Contrary to the case of JavaScript, 26 (70.3%) of the Python developers who responded to our survey generally consider the use of trivial packages as bad practice. Only 3 (8.1%) of survey participants stated that they do not think that using trivial package is a bad practice. The remaining 8 (21.6%) indicate that it really depends on the circumstances. For example, P-*PyPI* 3 states: *"If the language doesn't provide such common, inherently useful functionality then fixing this oversight by the use of a third-party library is only reasonable. Moreover, little functionality is actually 'trivial'. It may be short to implement but most likely a mistake in it will introduce a bug into the program as surely as a mistake in something 'non-trivial'."*

Table 11: Reasons and percentage for using trivial packages in both *npm* and *PyPI*.

Reason	**Description**	***npm***		***PyPI***	
		#Responses	**%**	**#Responses**	**%**
Well-implemented & tested	Participants state that trivial packages are effectively implemented and tested.	48	54.6%	20	54.1%
Increased productivity	Trivial packages reduce the time needed to implement existing source code.	42	47.7%	12	32.4%
Well-maintained code	It eases source code maintenance, since other developers maintain the trivial package.	8	9.1%	2	5.4%
Improved readability & reduced complexity	Using trivial packages improve the source code quality in terms of readability and reduce complexity.	8	9.1%	5	13.5%
Better performance	Trivial packages improve the performance of web applications compared to the use of large frameworks.	3	3.4%	0	0.0%
No reason	-	7	8.0%	7	18.9%

Survey participants from the two package management platforms have different perception about using trivial packages. Whereas most of the surveyed developers from *npm* (57.9%) do NOT believe that using trivial packages is a bad practice, the majority of the surveyed developers from *PyPI* (70.3%) consider using trivial packages as bad practice.

5.5.2 Why Do Developers Use Trivial Packages?

While we have answered the question as to whether developers think using trivial packages is a bad practice, what we are most interested in is why do developers resort to using trivial packages and

what do they view as the drawbacks of using trivial packages. Therefore, the second part of the survey asks participants to list the reasons why they resort to using trivial packages. To ensure that we do not bias the responses of the developers, the answer fields for these questions were in free-form text, i.e., no predetermined suggestions were provided. We then analyze separately the responses from the two surveys (JavaScrip and Python). After gathering all of the responses, we grouped and categorized the responses in a two-phase iterative process. In the first phase, two of the authors carefully read the participant's answers and came up with a number of categories that the responses fell under. Next, they discussed their groupings and agreed on the extracted categories. Whenever they failed to agree on a category, the third author was asked to help break the tie. Once all of the categories were decided, the same two authors went through all the answers again and classified them into their respective categories. For the majority of the cases, the two authors agreed on most categories and the classifications of the responses. To measure the agreement between the two authors, we used Cohen's Kappa coefficient [43]. The Cohen's Kappa coefficient has been used to evaluate inter-rater agreement levels for categorical scales, and provides the proportion of agreement corrected for chance. The resulting coefficient is scaled to range between -1 and +1, where a negative value means less than chance agreement, zero indicates exactly chance agreement, and a positive value indicates better than chance agreement [65]. In our categorization, the level of agreement measured between the authors was of +0.90 and +0.83 for the *npm* survey and *PyPI* survey, respectively, which is considered to be excellent inter-rater agreement.

Table 11 shows the reasons for using trivial packages, as reported by respondents from both JavaScript and Python surveys. As we can see from the table, two most cited reasons (i.e., well-implemented & tests and increased productivity) are the same for both *npm* and *PyPI*. However, when it comes to the 3 less common reasons, there is a slight difference between *npm* and *PyPI*, most notably, the reason of trivial packages provide better performance was not evident in our survey.

Next, we discuss each of the reasons presented in Table 11 in more detail:

R1. Well-implemented & tested:

The most cited reason for using trivial packages is that they provide well implemented and tested code. More than half of the responses mentioned this reason with 54.6% and 54.1% of the responses from JavaScript and Python, respectively. In particular, although it may be easy for developers to code these trivial packages themselves, it is more difficult to make sure that all the details are addressed, e.g., one needs to carefully consider all edge cases. Some example responses that mention these issues are stated by participants P-*npm* 68, P-*npm* 4, and P-*PyPI* 5, who cite their reasons for using trivial packages as follows: P-*npm* 68: *"Tests already written, a lot of edge cases captured [...]."*, P-*npm* 4: *"There may be a more elegant/efficient/correct/cross-environment-complatible solution to*

a trivial problem than yours", and P-*PyPI* 5: *"They have covered extra cases that I would not do or thought initially."*

R2. Increased productivity:

The second most cited reason is the improved productivity that using trivial packages enables with 47.7% and 32.4% for JavaScript and Python, respectively. Trivial tasks or not, writing code on your own requires time and effort, hence, many developers view the use of trivial packages as a way to boost their productivity. In particular, early on in a project, a developer does not want to worry about small details, they would rather focus their efforts on implementing the more difficult tasks. For example, participants P-*npm* 13 and P-*npm* 27 from the JavaScript survey state: P-*npm* 13: *"[...] and it does save time to not have to think about how best to implement even the simple things."* & P-*npm* 27: *"Don't reinvent the wheel! if the task has been done before."*. Another example from the Python survey, participant P-*PyPI* 17 states: *"Often I do write the code myself. And then package it into a re-usable module so that I don't have to write it again later. And again. And again... At this point, whether the module is authored by myself or someone else is mostly irrelevant. What's relevant is that I get to avoid repeatedly implementing the same functionality for each new project."*

The aforementioned are clear examples of how developers would rather not code something, even if it is trivial. Of course, this comes at a cost, which we discuss later.

R3. Well-maintained code:

A less common (9.1% and 5.4% of the responses from javaScript and Python), but cited reason for using trivial packages is the fact that the maintenance of the code need not to be performed by the developers themselves; in essence, it is outsourced to the community or the contributors of the trivial packages. For example, participants P-*npm* 45 and P-*PyPI* 1 states, P-*npm* 45: *"Also, a highly used trivial package is probable to be well maintained."* and P-*PyPI* 1: *"The simple advantages are that they may be trivial AND used by many people and therefore potentially maintained by developers."* Even tasks such as bug fixes are dealt with by the contributors of the trivial packages, which is very attractive to the users of the trivial packages, as reported by participant P-*npm* 80: *"[...], leveraging feedback from a larger community to fix bugs, etc."*

R4. Improved readability & reduced complexity:

Participants for also reported that using trivial packages improves the readability and reduces the complexity of their code with 9.1% and 13% responses for the two package management systems. For example, P-*npm* 34 states: *"immediate clarity of use and readability for other developers for commonly used packages[...]"* & P-*npm* 47 states: *"Simple abstract brings less complexity."* Python

developers report the same advantage of using trivial packages. For example, P-*PyPI* 5 states that *"Code clarity. When many two liners become one liners it saves space. Its the whole point of batteries included mentally..."*

R5. Better performance:

A few of the JavaScript participants (3.4%) stated that using trivial packages improves performance since it alleviates the need for their application to depend on large frameworks. For example, P-*npm* 35 states: *"[...] you do not depend on some huge utility library of which you do not need the most part."* While JavaScript developers reported that trivial packages improve the performance, non of the Python respondents report such a claim. One explanation for this is that JavaScript is used to develop front-end applications, which is often sensitive to performance, whereas the Python is used to implement applications in a wide variety of domains.

Overall the developer responses show that there a different perception of using trivial package among developers from the two package management platforms. Only a small percentage (8.0%) of the respondents from JavaScript stated that they do not see a reason to use trivial packages. However, for Python developers 18.9% of the respondents believe that there are no advantages of using trivial packages.

> JavaScript and Python developers believe that the two most cited reasons for using trivial packages are 1) they provide well implemented and tested code and 2) they increase productivity. However, only JavaScript developers cited that using trivial packages improves the performance of their applications. Additionally, only 8.0% of the JavaScript participants see no reason of using trivial packages while more than 18% of Python developers believe that there is no real reason to use trivial packages.

5.5.3 Drawbacks of Using Trivial Packages

In addition to knowing the reasons why developers resort to trivial packages, we wanted to understand the other side of the coin - what they perceive to be the drawbacks of their decision to use these packages. The drawbacks question was part of our survey and we followed the same aforementioned process to analyze the survey responses. In the case of the drawbacks the Cohen's Kappa agreement measure was +0.86 and +0.91 for *npm* and *PyPI*, respectively, which is considered to be an excellent agreement.

Table 12 lists the drawback mentioned by the survey respondents along with a brief description and the frequency of each drawback. As we can see from the table, the top two most cited drawbacks

(i.e., dependency overhead and breakage of applications) are the same for both, *npm* and *PyPI*. However, for the less cited drawbacks, *npm* developers cited performance, development slow down and missed learning opportunities as the next set of drawbacks, whereas in *PyPI*, the developers consider security, development slow down and decreased performance as the next set of drawbacks. It is worth noting however that there is very little difference between the individual drawbacks (e.g., security vs. development slow down) within the two ecosystems (i.e., *npm* and *PyPI*). Next, we discuss each of the drawbacks in more detail:

D1. Dependency overhead:

The most cited drawback of using trivial packages is the increased dependency overhead, e.g., keeping all dependencies up to date and dealing with complex dependency chains, that developers need to bear [33, 131]. This situation is often referred to as 'dependency hell', especially when the trivial packages themselves have additional dependencies. This drawback came through clearly in many comments, which account for 55.7% of the responses form JavaScript developers. For example, P-*npm* 41 states: *"[...] people who don't actively manage their dependency versions could [be] exposed to serious problems [...]"* & P-*npm* 40: *"Hard to maintain a lot of tiny packages"*. For Python developers, the percentage of responses related to dependency overhead is high (67.6%) as well. Some example responses from Python developers that mention these issues are stated by participants P-*PyPI* 2, P-*PyPI* 4 & P-*PyPI* 13 who state that: P-*PyPI* 2: *"...it's more difficult to distribute something with a dependency that doesn't come with Python."*, P-*PyPI* 4: *"Lots of brittle dependencies."* & P-*PyPI* 13: *"When your projects consist of a lot trivial modules, it becomes almost impossible to track their update and some time you might forget what even they do."* Hence, while trivial packages may provide well-implemented/tested code and improve productivity, developers are clearly aware that the management of the additional dependencies is something they need to deal with.

D2. Breakage of applications:

Developers also worry about the potential breakage of their application due to a specific package or version becoming unavailable. JavaScript developers stated this issue in 18.2% of the responses while the percentage is 32.4% for Python developers. For example, in the left-pad issue, the main reason for the breakage was the removal of left-pad, P-*npm* 4 states: *"Obviously the whole 'left-pad crash' exposed an issue"* & P-*PyPI* 22 states: *"potential for breaking (NPM leftpad situation)"*. However, since that incident, *npm* has disabled the possibility of a package to be removed [145]. Although disallowing the removal solves part of the problem, packages can still be updated, which may break an application. This is issue was clear from one of the responses, P-*PyPI* 7, who stated *"Potential*

Table 12: Drawback and percentage for using trivial packages in both *npm* and *PyPI*.

Drawback	**Description**	***npm***		***PyPI***	
		#Responses	**%**	**#Responses**	**%**
Dependency overhead	Using trivial packages results in a dependency mess that is hard to update and maintain.	49	55.7%	25	67.6%
Breakage of applications	Depending on a trivial package could cause the application to break if the package becomes unavailable or has a breaking update.	16	18.2%	12	32.4%
Decreased performance	Trivial packages decrease the performance of applications, which includes the time to install and build the application.	14	15.9%	3	8.1%
Slows development	Finding a relevant and high quality trivial package is a challenging and time consuming task.	11	12.5%	4	10.8%
Missed learning opportunities	The practice of using trivial packages leads to developers not learning and experiencing writing code for trivial tasks.	8	9.1%	0	0%
Security	Using trivial packages can open a door for security vulnerability.	7	8.0%	5	13.5%
Licensing issues	Using trivial packages could cause licensing conflicts.	3	3.4%	2	5.4%
No drawbacks	-	7	8.0%	3	8.1%

for breaking changes from version to version." For a non-trivial package, it may be worth it to take the risk, however, for trivial packages, it may not be worth taking such a risk.

D3. Decreased performance:

This issue is related to the dependency overhead drawback. Developers mentioned that incurring the additional dependencies slowed down the build time and increased application installation times (15.9% & 8.1%). For example, P-*npm* 64 states: *"Too many metadata to download and store than a real code."* & P-*npm* 34 states: *"[...], slow installs; can make project noisy and unintuitive by attempting to cobble together too many disparate pieces instead of more targeted code."* Another Python developer ,P-*PyPI* 1, states: *"If the modules are not so ubiquitous, then needing the dependency is a real drag as one will have to install it. Also, the same job done with your own may run much faster and be easier to understand.* As mentioned earlier, in some cases it is not just the fact that the trivial package adds a dependency, but in some cases the trivial package itself depends on additional packages, which negatively impacts performance even further.

D4. Slows development:

In some cases, the use of trivial packages may actually have a reverse effect and slow down development with 12.5% & 10.8% of responses from JavaScript and Python developers. For example, as P-*npm* 23 and P-*npm* 15 state: P-*npm* 23: *"Can actually slow the team down as, no matter how trivial a package, if a developer hasn't required it themselves they will have to read the docs in order to double check what it does, rather than just reading a few lines of your own source."* & P-*npm* 15: *"[...], we have the problem of locating packages that are both useful and "trustworthy" [...]"*. It can be difficult to find a relevant and trustworthy package. Even if others try to build on your code, it is much more difficult to go fetch a package and learn it, rather than read a few lines of your code. Python developers also agree on this issue, for example P-*PyPI* 15 states *"If finding, reading, and understanding the documentation of a module takes longer than reading it implementation, the hiding of functionality in third-part trivial modules obscures the source base."*

D5. Missed learning opportunities:

In certain cases reported by only JavaScript developers (9.1%), the use of these trivial packages is seen as a missed learning opportunity for developers. For example, P-*npm* 24 states: *"Sometimes people forget how to do things and that could lead to a lack of control and knowledge of the language/technology you are using"*. This is a clear example of where just using a package, rather than coding the solution yourself, will lead to less knowledge about the code base. In contrast to JavaScript developers, Python developers seem do not to be worried about this issue since the use of trivial packages is not as common within the Python developer community as JavaScript developers.

D6. Security:

In some cases the trivial packages may have security flaws that make the application more vulnerable. This is an issue pointed out by a few developers (8.0% & 13.5%), for example, as P-*npm* 15 mentioned earlier, it is difficult to find packages that are trustworthy. Also, P-*npm* 57 mentions: *"If you depend on public trivial packages then you should be very careful when selecting packages for security reasons"* & P-*PyPI* 3 states *"more dependencies, greater likelihood of not knowing of how code actually works at lower level, security issues."* As in the case of any dependency one takes on, there is always a chance that a security vulnerability could be exposed in one of these packages.

D7. Licensing issues (3.4%):

In some cases from both responses (3.4% and 5.4% for JavaScript and Python), developers are concerned about potential licensing conflicts that trivial packages may cause. For example, P-*npm* 73 states: *"[...], possibly license-issues"*, P-*npm* 62: *"[...], there is a risk that the 'trivial' package might be licensed under the GPL must be replaced anyway prior to shipping."* P-*PyPI* 23 also mentions *"Can be licensing hell"*.

In general, we observe similar concerns regarding the use of trivial packages in the two studied software managements platforms. There were also approximately 8% of the responses in both package management systems that stated they do not see any drawbacks with using trivial packages.

> The two most cited drawbacks of using trivial packages are 1) they increase dependency overhead and 2) they may break their applications due to a package or a specific version becoming unavailable or incompatible.

5.6 Putting Developer Perception Under the Microscope

The developer surveys provided us with great insights on why developers use trivial packages and what they perceive to be their drawbacks. However, whether there is empirical evidence to support their perceptions remains unexplored. Thus, we examine the most commonly cited reason for using trivial packages, i.e., the fact that trivial packages are well tested, and drawback, i.e., the impact of additional dependencies, based on our findings in Section 5.5.

5.6.1 Examining the 'Well Tested' Perception

As shown in Table 11, more than half of the responses from the studied package management platforms indicate that they use trivial packages since they are well implemented and tested. However, is this really the case - are trivial packages really well testing? In this section, we want to examine whether this believe has any grounds or not.

Node Package Manager (*npm*):

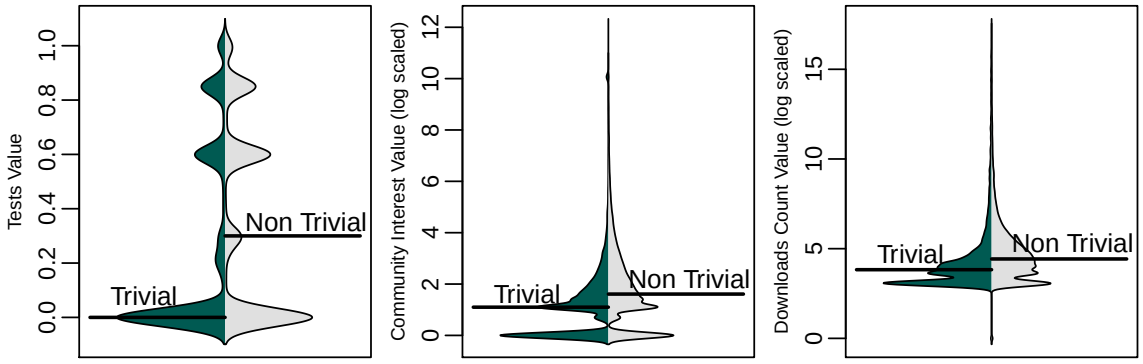
npm requires that developers provide a test script name with the submission of their packages (listed in the package.json file). In fact, 81.2% (31,521 out of 38,845) of the trivial packages in our dataset have some test script name listed. However, since developers can provide any script name under this field, it is difficult to know if a package is *actually* tested.

We examine whether a *npm* package is really well tested and implemented from two aspects; first, we check if a package has tests written for it. Second, since in many cases, developers consider packages to be 'deployment tested', we also consider the usage of a package as an indicator of it being well tested and implemented [208]. To carefully examine whether a package is really well tested and implemented, we use the *npm* online search tool (known as *npms* [45]) to measure various metrics related to how well the packages are tested, used and valued. To provide its ranking of the packages, *npms* mines and calculates a number of metrics based on development (e.g., tests) and usage (e.g., no. of downloads) data. We use three metrics measured by *npms* to validate the 'well tested and implemented' perception of developers, which are[2]:

1) Tests: considers the tests' size, coverage percentage and build status for a project. We looked into the *npms* source code and find that the Tests metric is calculated as: $testsSize * 0.6 + buildStatus * 0.25 + coveragePercentage * 0.15$. We use the Tests metric to determine if a package is tested and how trivial packages compare to non-trivial packages in terms of how well tested they are. One example that motives us to investigate how well tested a trivial package is the response by P-*npm* 68, who says: *"Tests already written, a lot edge cases captured [...]"*.

2) Community interest: evaluates the community interest in the packages, using the number of stars on GitHub & *npm*, forks, subscribers and contributors. Once again, we find through the source code of *npms* that Community interest is simply the sum of the aforementioned metrics, measured as: $starsCount + forksCount + subscribersCount + contributorsCount$. We use this metric to compare how interested the community is in trivial and non-trivial packages. We measure

[2]It is important to note that the motivation and full derivation (e.g., why they put a weight of 0.15 on the test coverage, etc.) of the metrics is beyond the scope of this chapter. We refer interested readers to the *npms* documentation for more details [45]. To make this chapter self-sufficient, we include how the metrics are calculated here.



(a) Tests (b) Community Interest (c) Download Count

Figure 16: Distribution of Tests, Community Interest and Download Count Metrics for *npm* packages management system.

the community interest since developers view the importance of the trivial packages as evidence of its quality as stated by P-*npm* 56, who says: *"[...] Using an isolated module that is well-tested and vetted by a large community helps to mitigate the chance of small bugs creeping in."*

3) Download count: measures the mean downloads for the last three months. Again, the number of downloads of a package is often viewed as an indicator of the package's quality; as P-*npm* 61 mentions: *"this code is tested and used by many, which makes it more trustful and reliable."*.

As an initial step, we calculate the number of trivial packages that have a $Tests$ value greater than zero, which means trivial packages that have some of tests. We find that only 45.2% of the trivial packages have tests, i.e., a $Tests$ value > 0. In addition, we compare the values of the Tests, Community interest and Download count for Trivial and non-Trivial packages. Our focus is on the values of the aforementioned metric values for trivial packages, however, we also present the results for non-trivial packages to put our results in context.

Figure 20 shows the bean-plots for the Tests, Community interest and Download count. The figures show that in all cases trivial packages have, on median, a smaller Tests value, Community interest value and Download count compared to non-trivial packages. That said, we observe from Figure 16a that the distribution of the Tests metric is similar for both, trivial and non-trivial packages. Most packages have a Tests value of zero, then there are small pockets of packages that have values of aprox. 0.25, 0.6, 0.8 and 1.0. In the case of the Community interest and Download count metrics, once again, we see similar distributions, although clearly the median values are lower for trivial packages.

To examine whether the difference in metric values between trivial and non-trivial packages is statistically significant, we performed a Mann-Whitney test to compare the two distributions and

Table 13: Mann-Whitney Test (p-value) and Cliff's Delta (d) for Trivial vs. Non Trivial Packages in *npm*.

Metrics	p**-value**	d
Tests	2.2e-16	-0.119 (small)
Community interest	2.2e-16	-0.269 (small)
Downloads count	2.2e-16	-0.245 (small)

determine if the difference is statistically significant, with a p-value < 0.05. We also use Cliff's Delta (d), which is a non-parametric effect size measure to interpret the effect size between trivial and non-trivial packages. As suggested in [75], 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 $d \geq 0.474$.

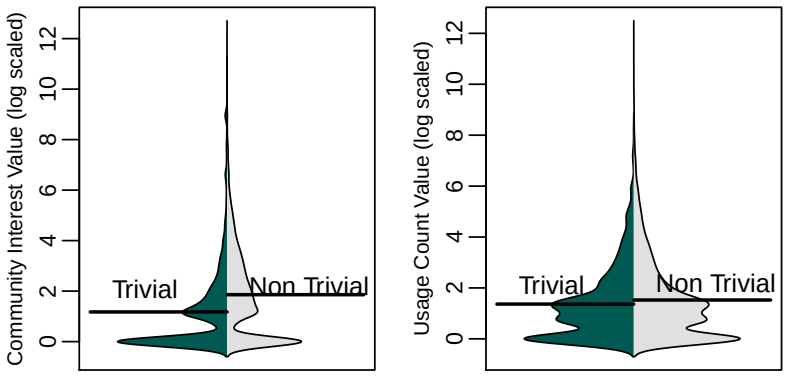
Table 13 shows the p-values and effect size values. We observe that in all cases the differences are statistically significant, however, the effect size is small. The results show that although the majority of trivial packages do not have tests written for them, and have statistically lower Tests, Community interest, and Download count values, their effect size is smaller than non-trivial packages.

Python Package Index (*PyPI*):

Since the *PyPI* does not collect any metadata to show if the Python package is tested or not, we use other data source to examine the well tested perception. To do so, we use two ways to examine whether Python packages are tested or not: 1) we use the source code of the packages that are hosted on GitHub. 2) we relied on information about Python packages collected by the open source service libraries.io [111]. libraries.io monitors and collects the metadata of open source packages across 36 different package managers platforms. It falls under the CC-BY-SA 4.0 licenses and have been used in other research work (e.g,. [55, 54]). We obtain the extracted metadata information related to *PyPI* package management. Once again, we examine the testing perception in three complementary ways.

1) Tests: we examine if the package has any test code written. Since there is no standard way to determine that a Python application has tests (e.g,. there exist more than 100 Python testing tools [154]), we manually investigate whether the *PyPI* package contains test code written or not. The idea is that if the developers writes tests, then they will put these tests in the package repository. One example that motivated us to look for the test code of a package is the developer response: P-*PyPI* 11 who stated *"Shorter code overall, well-tested code for fundamental tasks helps smooth over language nits"*.

Since this is a heavily manual process, we decided to examine a representative sample of the



(a) Community Interest (b) Usage Count

Figure 17: Distribution of Tests, Community Interest and Download Count Metrics for *PyPI* packages management system.

packages. Therefore, we took a statistically significant sample from the 6,759 Python packages that we identified as trivial Python packages (Section 5.4.1). The sample size was selected randomly to attain 5% confidence interval and a 95% confidence level. This sampling process resulted in 364 *PyPI* trivial packages. Then, the first two authors manually examine the code bases of sampled packages looking for test code to identify the packages that has test.

1) Community interest: evaluates the community interest in the packages, using the number of stars on GitHub, forks, subscribers and contributors. We adopted the same formula defined by *npms*, which is basically the sum of the aforementioned metrics, measured as: $starsCount + forksCount + subscribersCount + contributorsCount$. We use this metric to compare how interested the community is in trivial and non-trivial packages. We measure the community interest since developers view the importance of the trivial packages as evidence of its quality.

1) Usage count: represents the number of applications that use a package. The more applications using a Python package, the more popular that package is. This may also indicate that the package is of high quality. For example, P-*PyPI* 11 indicated *"The simple advantages are that they may be trivial AND used by many people and therefore potentially maintained by developers."* Hence, we use the *usage count* metric since it indicates the package quality; thus, many developers use it in their applications.

We found that out of the 364 sampled trivial Python packages that we manually examine, 185 (50.82%) packages do not have test code in them, while 179 (49.18%) of the examined packages have test code written in them. It is important to note that our analysis only examines whether a trivial package has tests or not, whether these tests are actually effective is a completely different issue and is one of the reasons for examine the other two ways Community interest and Usage count.

Figure 17 shows the bean-plots for the Community interest and Usage count values for trivial and

Table 14: Mann-Whitney Test (p-value) and Cliff's Delta (d) for Trivial vs. Non Trivial Packages in $PyPI$.

Metrics	p**-value**	d
Community interest	2.2e-16	-0.251 (small)
Usage count	0.004557	-0.039 (negligible)

non-trivial Python packages in our dataset. The figures show that in the two cases trivial Python packages have, on median, a smaller Community interest value and Usage count compared to non-trivial packages. That said, we observe from Figure 17a that in the case of the Community interest metric, we see clearly the median values are lower for trivial packages. Figure 17b shows that the distribution of the Usage count metric is similar for both, trivial and non-trivial packages. Once again, we examine whether the difference in metric values between trivial and non-trivial packages is statistically significant. We performed a Mann-Whitney test to compare the two distributions and determine if the difference is statistically significant. We also use Cliff's Delta (d) measure the effect size between trivial and non-trivial packages. Table 14 shows the p-values and effect size values. We observe that in the cases of community interest and usage count, the differences are statistically significant, and the effect size is small and negligible, respectively.

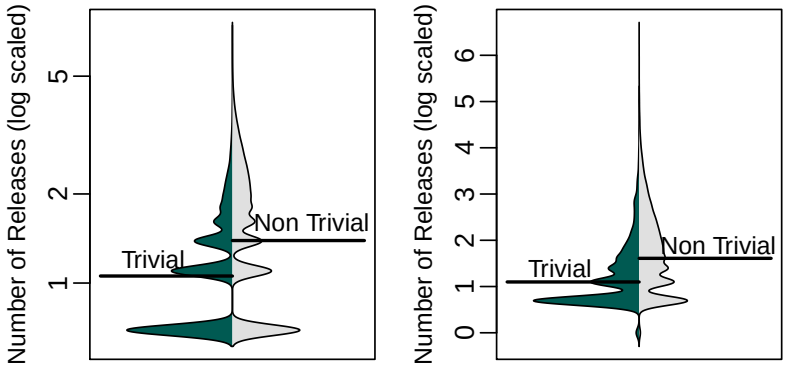
Contrary to developers' perception, only 45.2% and 49.2% of trivial Python and JavaScript packages actually have tests. Although, trivial packages have lower Tests, in terms of Community interest and Download count, there is not a large difference between trivial and non-trivial packages.

5.6.2 Examining the 'Dependency Overhead' Perception

As discussed in Section 5.5, the top cited drawback of using trivial packages is the fact that developers need to take on and maintain extra dependencies, i.e, dependency overhead. Examining the impact of dependencies is a complex and well-studied issue (e.g,. [50, 56, 7]) that can be examined in a multitude of ways. We choose to examine the issue from both, the application and the package perspectives.

Application-level Analysis:

When compared to coding trivial tasks themselves, using a trivial package imposes extra dependencies. One of the most problematic aspects of managing dependencies for applications is when



(a) *npm* (b) *PyPI*

Figure 18: Number of Releases for Trivial Packages Compared to Non-trivial Packages. For *npm* ecosystem, (p-value $<$ 2.2e-16 & Cliff's Delta (d) -0.312 (small)) while For *PyPI* ecosystem (p-value $<$ 2.2e-16 & Cliff's Delta (d) -0.383 (medium)).

these dependencies update, causing a potential to break their application. Therefore, as a first step, we examined the number of releases for trivial and non-trivial packages. The intuition here is that developers need to put in extra effort to ensure the proper integration of new releases. The bean-plot in Figure 18 shows the distribution of the number of releases for our studied package mangers. Figure 18a shows that trivial packages on *npm* have less releases than non-trivial packages (median is 2 for trivial and 3 for non-trivial packages), hence trivial packages do not require more effort than non-trivial packages. In Figure 18b, we also observe that trivial packages on *PyPI* have less releases than non-trivial packages. The fact that the trivial packages are updated less frequently may be attributed to the fact that trivial packages 'perform less functionality', hence they need to be updated less frequently.

Next, we examined how developers choose to deal with the updates of trivial packages. One way that application developers reduce the risk of a package impacting their application is to 'version lock' the package. For example in the JavaScript application that use *npm* packages, version locking a dependency/package means that it is not updated automatically, and that only the specific version mentioned in the packages.json file is used. As stated in a few responses from our survey, e.g., P-*npm* 8: *"[...] Also, people who don't lock down their versions are in for some pain"*. In general, there are different types of version locks, i.e., only updating major releases, updating patches only, updating minor releases or no lock at all, which means the package automatically updates. The version locks are specified in configuration file next to every package name for example *npm* defines it in the packages.json file. We examined the frequency at which trivial and non-trivial packages are locked. For *npm*, we find that on average, trivial packages are locked 14.9% of the time, whereas non-trivial packages are locked 11.7% of the time. However, the Wilcox test shows that the difference is

not statistically significant p-value > 0.05. Hence, we cannot say that developers version lock trivial packages more. On the other hand, in *PyPI*, we find that on average, trivial packages are locked 31.7% of the time, whereas non-trivial packages are locked 36.2% of the time. Also, the Wilcox test shows that the difference is statistically significant with p-value = 9.707e-08.

Our findings show that trivial packages are locked more in *npm* (albeit the difference is not statistically significant) and the reverse is true in *PyPI* where trivial packages are locked less than non-trivial packages. In both cases however, we find that there is not a large difference between the percentage of packages (trivial vs. non-trivial) being locked.

Package-level Analysis:

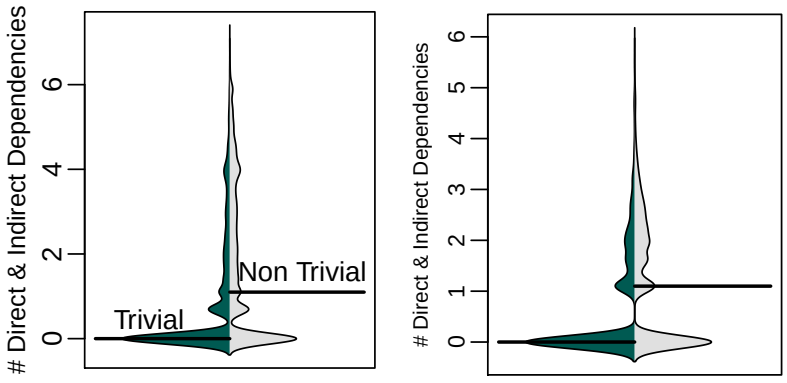
At the package level, we investigate the direct and indirect dependencies of trivial packages. In particular, we would like to determine if the trivial packages have their own dependencies, which makes the dependency chain even more complex. For each trivial and non-trivial package on *npm*, we install it and then count the actual number of (direct and indirect) dependencies that the package requires. Doing so, allows us to know the true (direct and indirect) dependencies that each package requires. Note that simply looking into the `.json` file and the `require` statements will provide the direct dependencies, but not the indirect dependencies. Hence, we downloaded all the packages in our *npm* dataset, mock installed[3] them and build the dependency graph for the *npm* platform.

Similarly, for *PyPI*, we count the actual number of (direct and indirect) dependencies that the package requires. To do so, we leveraged the metadata provided by Valiev *et al.* [190]. In their studied Valiev *et al.* extracted the list of direct and indirect dependencies of each package on *PyPI*. We resort to use the data provided in [190] since it is recently extracted data and covers the history of *PyPI* for more than six years. We then read the dependencies of each packages and build a dependency graph for *PyPI* platform.

Figure 19 shows the distribution of dependencies for trivial and non-trivial packages for the *npm* and *PyPI*. Since most trivial packages have no dependencies, the median is zero. Therefore, we bin the trivial packages based on the number of their dependencies and calculate the percentage of packages in each bin.

Table 15 shows the percentage of packages and their respective number of dependencies for both *npm* and *PyPI*. We observe that the majority of *npm* trivial packages (56.3%) have zero dependencies, 27.9% have between 1-10 dependencies, 4.3% have between 11-20 dependencies, and 11.5% have more than 20 dependencies. The table also shows that *PyPI* trivial packages do not have as much dependencies as the *npm* packages. In fact 63.2% of *PyPI* packages have zero dependencies and approx. 34% of trivial packages have between 1-20 dependencies. Only approximately 3% of the

[3]we modified the *npm* code to intercept the install call and counted the installations needed for every package.



(a) *npm* (b) *PyPI*

Figure 19: Distribution of Direct & Indirect Dependencies for Trivial and Non-trivial Packages (log scale). For *npm* (p-value < 2.2e-16 & Cliff's Delta (d) -0.279 (small)) while *PyPI* (p-value < 2.2e-16) & Cliff's Delta (d) -0.246 (small).

Table 15: Percentage of Packages vs. the Number of Dependencies Used in the *PyPI* package management system.

Packages	**# *npm* Dependencies (Direct & Indirect)**				**# *PyPI* Dependencies (Direct & Indirect)**			
	0	1-10	11-20	>20	0	1-10	11-20	>20
Trivial	56.3%	27.9%	4.3%	11.5%	63.2%	29.6%	4.3%	2.9%
Non Trivial	34.8%	30.6%	7.3%	27.3%	42.5%	39.4%	10.7%	7.4%

PyPI trivial packages have more than 20 dependencies. Interestingly, the table shows that some of the trivial packages in *npm* have many dependencies, which indicates that indeed, trivial packages can introduce significant dependency overhead. It also shows that *PyPI* trivial packages have small number of dependencies. One explanation of such a deference is that Python language has a more mature standard API that provides most of the needed utilities functionalities.

Trivial packages have fewer releases and are less likely to be version locked than non-trivial packages. That said, developers should be careful when using trivial packages, since in some cases, trivial packages can have numerous dependencies. In fact, we find that 43.7% of *npm* trivial packages have at least one dependency and 11.5% of *npm* trivial packages have more than 20 dependencies while 36.8% of *PyPI* trivial packages have at least one dependency and 2.9% of *PyPI* trivial packages have more than 20 dependencies.

5.7 Discussion, Relevance, and Implications

A common question that is asked in empirical studies is - *so what? what are the implications of your findings? why would practitioners care about your findings?* We discuss the issue of relevance of our study to the developer community, based on the responses of our survey and highlight some of the implications of our study.

5.7.1 Relevance: Do Practitioners care?

At the start of the study, we were not sure how practically relevant our study of trivial packages is. However, we were surprised by the interest of developers in our study. In fact, one of the developers (P-*npm* 39) explicitly mentioned the lack of research on this topic, stating *"There has not been enough research on this, but I've been taking note of people's proposed "quick and simple" code to handle the functionality of trivial packages, and it's surprised me to see the high percentage of times the proposed code is buggy or incomplete."*

Moreover, when we conducted our studies, we asked respondents if they would like to know the outcome of our study and if so, they provide us with an email address. Of the 125 JavaScript and Python respondents, 81 (approximately 65%) of them provided their email for us to provide them with the outcomes of our study. Some of these respondents hold very high level leadership roles in *npm*. To us this is an indicator that our study and its outcomes are of high relevance to the JavaScript and Python development communities.

5.7.2 Implications of the Study

Our study has a number of implications on both, software engineering research and practice.

Implications for Future Research:

Our study mostly focused on determining the prevalence, reasons for and drawbacks of using trivial packages in two large package management platforms *npm* and *PyPI*. Based on our findings, we find a number of implications/motivations for future work. First, our survey respondents indicated that the choice to use trivial packages is not black or white. In many cases, it depends on the team and the application. For example, one survey respondent stated that on his team, less experienced developers are more likely to use trivial packages, whereas the more experienced developers would rather write their own code for trivial tasks. The issue here is that the experienced developers are more likely to trust their own code, while the less experienced are more likely to trust an external package. Another aspect is the maturity of the application. As some of the survey respondents pointed out, they are much more likely use trivial packages early on in the application, so they

do not waste time on trivial tasks and focus on the more fundamental tasks of their application. However, once their application matures, they start to look for ways to reduce dependencies since they pose potential points of failure for their application. Hence, our study motivates future work to examine the relationship between team experience and application maturity and the use of trivial packages.

Second, survey respondents also pointed out that using trivial packages is seen favourably compared to using code from Questions & Answers (Q&A) sites such as Stack Overflow or Reddit. For example, P-*npm* 84 stated that *"I'd have to do research on how to solve a particular problem, peruse questions and answers on Stack Overflow, Reddit, or Coderanch, and find the most recent and readable solution among everything I've found, then write it myself. Why go through all of this work when you can simply 'require()' someone else's solution and continue working towards your goal in a matter of seconds?"* When compared to using code on Stack Overflow, where the developer does not know who posted the code, who else uses it or whether the code may have tests or not, using a trivial package that is on *npm* and/or *PyPI* is a much better option. In this case, using trivial packages is not seen as the *best* choice, but it is certainly a better choice. Although there have been many studies that examined how developers use Q&A sites such as StackOverflow [12, 13, 203, 18], we are not aware of any studies that compare code reuse from Q&A sites and trivial packages. Our findings motivate the need for such a study.

Practical Implications:

A direct implication of our findings is that trivial packages are commonly used by others, perhaps indicating that developers do not view their use as a bad practice, especially JavaScript developers. Moreover, developers should not assume that all trivial packages are well implemented and tested, since our findings show otherwise. *npm* developers need to expect more trivial packages to be submitted, making the task of finding the most relevant package even harder. Hence, the issue of how to manage and help developers find the best packages needs to be addressed. For example P-*npm* 15 indicated that *"... we have the problem of locating packages that are both useful and 'trustworthy' in an ever growing see of packages."* To some extent, *npms* has been recently adopted by *npm* to specifically address the aforementioned issue. Developers highlighted that the lack of a decent core or standard JavaScript library causes them to resort to trivial packages. Often, they do not want to install large frameworks just to leverage small parts of the framework, hence they resort to using trivial packages. For example, P-*npm* 35 *"especially in javascript relieves you from thinking about cross browser compatibility for special cases/coming up with polyfills and testing all edge cases yourself. Basically it's a substitute for the missing standard library. And you do not depend one some huge utility library of which you do not need the most part.,"* & P-*PyPI* 23 *"Usually an indication*

of the inadequacy of the standard library. This seems particularly so of JavaScript where you might find yourself using many such modules." Therefore, there is a need by the JavaScript community to create a standard JavaScript API or library in order to reduce the dependence on trivial packages. However, the issue of creating such a standard JavaScript library is under much debate [68].

5.8 Related Work

In this section, we discus the work that is related to the study in this chapter. We divided the related work to work related to code reuse in general and work studied software ecosystems.

5.8.1 Studies of Code Reuse.

Prior research on code reuse has been shown its many benefits, which include improving quality, development speed, and reducing development and maintenance costs [132, 114, 134, 22]. For example, Sojer and Henkel [178] surveyed 686 open source developers to investigate how they reuse code. Their findings show that more experienced developers reuse source code and 30% of the functionality of open source software (OSS) projects reuse existing components. Developers also reveal that they see code reuse as a quick way to start new projects. Similarly, Haefliger *et al.* [78] conducted a study to empirically investigate the reuse in open source software, and the development practices of developers in OSS. They triangulated three sources of data (developer interviews, code inspections and mailing list data) of six OSS projects. Their results showed that developers used tools and relied on standards when reusing components. Mockus [132] conducted an empirical study to identify large-scale reuse of open source libraries. Their study shows that more than 50% of source files include code from other OSS libraries. On the other hand, the practice of reusing source code has some challenging drawbacks including the effort and resource required to integrate reused code [57]. Furthermore, a bug in the reused component could propagate to the target system [58]. While our study corroborates some of these findings, the main goal is to define and empirically investigate the phenomenon of reusing trivial packages, in particular in JavaScript and Python applications.

5.8.2 Studies of Other Ecosystems.

In recent years, analyzing the characteristics of ecosystems in software engineering has gained momentum [23, 30, 120, 56]. For example, in a recent study, Bogart *et al.* [31, 33] empirically studied three ecosystems, including *npm*, and found that developers struggle with changing versions as they might break dependent code. Witter *et al.* [201] investigated the evolution of the *npm* ecosystem in an extensive study that covers the dependence between *npm* packages, download metrics and the usage of *npm* packages in real applications. One of their main findings is that *npm* packages and

updates of these packages is steadily growing. Also, more than 80% of packages have at least one direct dependency package.

Other studies examined the size characteristics of packages in an ecosystem. German *et al.* [71] studied the evolution of the statistical computing project GNU R, with the aim of analyzing the differences between code characteristics of core and user-contributed packages. They found that user-contributed packages are growing faster than core packages. Additionally, they reported that user-contributed packages are typically smaller than core packages in the R ecosystem. Kabbedijk and Jansen [95] analyzed the Ruby ecosystem and found that many small and large projects are interconnected. Decan *et al.* [55] investigated the evolution of package dependency networks for seven packaging ecosystems. Their findings reveal that the studied packaging ecosystems grow overtime in term of number of published and updated packages. They also observed that the increase number of transitive dependencies for some packages.

Other work investigate the challenges of using external package of an ecosystem including; identify conflicts between JavaScript package [147], examine how pull requests help developers to upgrade out-of-date dependencies in their applications [131], study the usage of repository badges in the *npm* ecosystem [188], and the usage of dependency graph to discover hidden trend in an ecosystem [105].

In many ways, our study complements the previous work since, instead of focusing on all packages in an ecosystem, we specifically focus on trivial packages and we studied them in two different package management systems *npm* and *PyPI*. Moreover, we examine the reasons developers use trivial package and what they view as their drawbacks. We study the reuse of trivial packages, which is a subset of general code reuse. Hence, we do expect there to be some overlap with prior work. Like many empirical studies, we confirm some of the prior findings, which is a contribution on its own [91, 170]. Moreover, this chapter adds to the prior findings through, for example, our validation of the developers' assumptions. Lastly, we do believe our study fills a real gap since 65% of the participants said they wanted to know our study outcomes.

5.9 Threats to Validity

In this section, we discuss the threats to the validity of our case study.

5.9.1 Construct Validity

Construct validity considers the relationship between theory and observation, in case the measured variables do not measure the actual factors. To define trivial packages, we surveyed 12 JavaScript and 13 Python developers who are mostly graduate student with some professional experience. However, we find that there was a clear vote for what is considered a trivial package. Also, although our data

suggested that packages with ≤ 35 LOC and a complexity ≤ 10 are trivial packages, we believe that other definitions are possible for trivial packages. That said, of the 125 survey participants that we emailed about using trivial packages, only 2 mentioned that the flagged package is not a trivial package (even though it fit our criteria). To us, this is a confirmation that our definition applies in the vast majority of the cases, although clearly it is not perfect.

We use the LOC and complexity of the code to determine trivial packages. In some cases, these may not be the only measures that need to be considered to determine a trivial packages. For example, some of the trivial packages have their own dependencies, which may need to be taken into consideration. However, our experience tells us that most developers only look at the package itself and not its dependencies when determining if it is trivial or not. That said, when we replicate this questionnaire with another set of participants from the Python language community, we found that developers seem to confirm our definition of a trivial JavaScript/Python packages [11].

Our list of reasons for and drawbacks of using trivial packages are based on a survey of 88 JavaScript and 37 Python developers. Although this is a large number of developers, our results may not hold for all developers. A different sample of developers may result in a different list or ranking of advantages and disadvantages. To mitigate the risk due to this sampling, we contacted developers from different applications and as our responses show, most are experienced developers. Also, there is potential that our survey questions may have influenced the replies from the respondents. However, to minimize such influence, we made sure to ask for free-form responses (to minimize any bias) and we publicly share our survey and all of our anonymized survey responses.

In our first study on *npm*, we used *npms* to measure various quantitative metrics related to testing, community interest and download counts. Our measurements are only as accurate as *npms*, however, given that it is the main search tool for *npm*, we are confident in the the *npms* metrics. We also use libraries.io to calculate the community interested and the usage count metrics for *PyPI* packages, and our measurements are as accurate as libraries.io. We resort to use the libraries.io data since it has been used on other prior work (e.g,. [55, 54]). In addition, we use the dataset provided by Valiev *et al.* [190] to measure the direct and indirect dependencies of the packages on *PyPI*.

We do not distinguish between the domain of studied packages, which may impact the findings. However, to help mitigate any bias we analyzed more than 230,000 *npm* and 74,663 *PyPI* packages that cover a wide range of package domains.

We removed test code from our dataset to ensure that our analysis only considers production source code. We identified test code by searching for the term 'test' (and its variants) in the file names and file paths. Even though this technique is widely accepted in the literature [74, 189, 210], to confirm whether our technique is correct, i.e., files that have the term 'test' in their names and paths actually contain test code, we took a statistically significant sample of the packages to achieve

a 95% confidence level and a 5% confidence interval and examined them manually.

In addition, to examine the well-tested perception for the *PyPI* trivial packages, the first two authors manually examine the source code of the trivial packages to classify whether they have test code written or not. Since this classification process is straightforward (i.,e. to see if the repository has test code or no), we did not apply a formal review process, but we did have all cases that were unclear discussed by the first two authors even though there were not many such cases.

5.9.2 External Validity

External validity considers the generalization of our findings. All of our findings were derived from open source JavaScript applications and *npm* packages and its replication on Python and *PyPI* packages. Even though we believe that the two studied two software package managers are amongst the most commonly used ones, our findings may not generalize to other platforms or ecosystems. That said, historical evidence shows that examples of individual cases contributed significantly in areas such as physics, economics, social sciences and even software engineering [67]. We believe that strong empirical evidence is built from both, studies on individual cases and studies on large samples.

5.10 Chapter Summary

The use of trivial packages is an increasingly popular trend in software development [10]. Like any development practice, it has its proponents and opponents. The goal of our study is to examine the prevalence, reasons and drawbacks of using trivial packages in different package management platforms, namely *npm* and *PyPI*.

Our results indicate that trivial packages are commonly and widely used in JavaScript and Python applications. We also find that while the majority of JavaScript developers in our study do not oppose the use of trivial packages, the majority of Python developers believe that using trivial packages could be harmful. Additionally, based on the developers responses, developers from the two packages management platforms stated that the main reasons for developers to use trivial packages is due to the fact that they are considered to be well implemented and tested. However, they do cite the fact that the additional dependencies' overhead as a drawback of using these trivial packages. Our empirical study showed that considering trivial packages to be well tested is a misconception since more than half of the trivial package we studied do not even have tests written, however, these trivial packages seem to be 'deployment tested' and have similar Community interest and Download/usage count values as non-trivial packages. In addition, we find that some of the trivial packages have their own dependencies and, in our studied dataset, 11.5% of the *npm* and 2.9% of

the *PyPI* trivial packages have more than 20 dependencies. Hence, developers should be careful about which trivial packages they use.

In the previous three chapters (Chapters 3, 4, & 5), we have focused on understanding the use and type of knowledge reused from crowdsourcing platforms through studying some of the well-known crowdsourcing platforms. Surprisingly while the crowd provides tremendous development knowledge that developers can use, we found that supporting such reuse through ensuring the quality is lacking. To facilitate reusing crowdsourcing knowledge, we believe that a quality assurance technique can be improved to suit these development practices. Thus, in the following chapter, we turn our attention to improve the efficiency of reusing crowdsourcing knowledge in the software development process.

Chapter 6

Improving the efficiency of Reusing Crowdsourced Knowledge

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 chapter 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 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.

6.1 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 [196]. Due to their many advantages, Hilton *et al.* showed that CI is used by both, the open source community and in industrial software projects [85, 86] and that as much as 40% of 34,544 of analyzed popular GitHub projects use CI [86].

Despite CI's many benefits and wide popularity, it also has several drawbacks. CI's process can take a very long time to complete [128], especially for large projects [46]. 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.* [60], p. 87). Specially, when these commits are submitted by voluntary contributors known as the crowd.

Most of the previous work on CI focused on the study of its usage and benefits (e.g,. [110, 196]). Other work examined the reason for failing builds [172], and even tried to predict the results of the build result [83]. However, very few studies try 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 and cause 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 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 [26], where developers explicitly skip the build when using Travis CI to understand the

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

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 documentation 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 a savings of 917 minutes per project.

Moreover, to better understand the importance of the CI skip technique from developers, 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 that 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].

6.1.1 Organization of the Chapter

The rest of the chapter is organized as follows. Section 6.2 provides background on the CI process, in particular Travis CI. We detail our data collection and the dataset used in our study in Section 6.3. We describe our approach and the reasons that developers CI skip commits in Section 6.4. We present our case study results, detailing the effectiveness and effort savings of our technique in Section 6.5. In Section 6.6, 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 6.7. Section 6.8 presents our prototype tool, CI-Skipper. The work related to our study is discussed in Section 6.9 and the threats to validity in Section 6.10. Section 6.11 concludes the chapter.

6.2 Background and Terminology

Since the main goal of our study is to detect commit that can be CI skip, 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 in 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 enhances the run time and better utilizes the processing power of the virtual machine[3].

[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 16: 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%

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 [25].

6.3 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.

6.3.1 The TravisTorrent Dataset

The main data source for our study is the TravisTorrent dataset [26]. 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 [25].

Since we are interested in non-toy projects, and similar to prior research [97], 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 [26]. 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 chapter, we focused on the study of the 393 projects written in Java. We chose to focus

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

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 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 [196]. That said, it is important to note that our study is not language dependent and our appraoch & technique can be applied on projects written in any programming language.

We cloned all the 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 that the `.travis.yml`[5] file was 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), which made our final dataset contain 392 Java projects. We analyze the commit history of all the branches for each project and extracted various metrics, which include, 1) the type of change 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.

6.3.2 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 an average number of 620.4 builds (median of 296).

To come up with one result for a build we aggregate the result 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 16 shows the summary statistics for each of the different build outcomes in 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].

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

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

Table 17: Shows Projects in the 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.

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 of 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.

6.3.3 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 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`

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

Table 18: 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	Mean	Max.	Min.	Median	Mean	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.

`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 means 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 17 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.

6.4 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. Using our knowledge, 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.

Table 19: 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

6.4.1 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 determining the commit that 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 18 presents the statistics for the entire dataset for 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 they have 3.06% skipped commits on average. In addition, we observe that the 392 analyzed projects contains 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. The Table 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 [193, 28, 185]. Table 18 shows the time-frame of the studied projects (measured in days). For all the projects the median number of days is 865 (average = 850), while the projects that have at least one CI skip commit have a median of 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 [86]. 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 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.

6.4.2 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 [171], 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 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 [43]. 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 [65].

Table 19 lists the reasons, provides a 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 6.7.

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 reasons for developers to skip 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%):** In 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 through the repository. 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 6.7.
- **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 type 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.

6.4.3 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 [194, 87], 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[9]. 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 6.8).

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

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

6.5 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.

6.5.1 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.* [60], 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 6.3.3. 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 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., 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 data imbalance into account. Similar to prior work [47, 173], 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 commits 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 $2 \times \frac{0.1217 \times 0.5}{0.1217+0.5} = 0.098$.

Table 20: Performance of 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**

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 at 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 20 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 20 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 projects (`Mechanical Tsar`), our technique performs poorly. We manually investigated the data from this project to better understand the reasons for this poor performance. For `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 [172]) 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 6.10.2). 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 6.8).

> 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.

6.5.2 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.

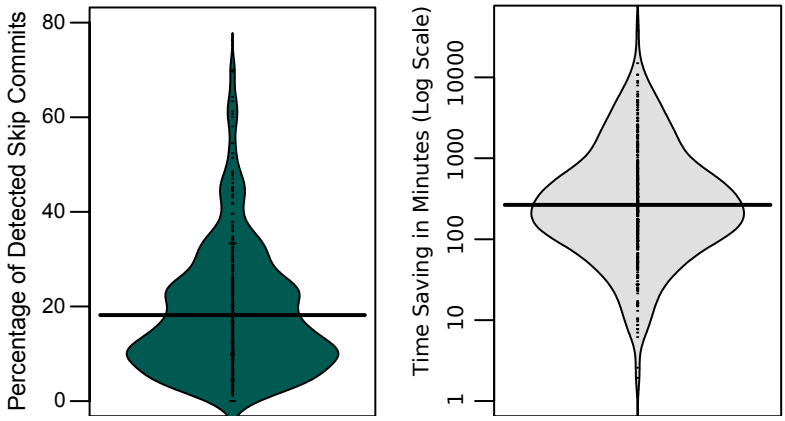
Approach: To address this research question, we evaluate the effort saving, by applying the defined rule-based technique on real build commits from the TravisTorrent dataset. We perform our analysis on projects from the TravisTorrent dataset since it contains build times and results, which enable us 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 save 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 [26, 25]. 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 bt the Travis Torrent dataset [26, 25]. Second, we aggregate the result of the *set of changes*. So, we 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. 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.

Results: *Percentage of saved build commits:* Figure 20a 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 [85].



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

Figure 20: 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.

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, on the projects, `Money and Currency API`, has 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 20b 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. In some cases, 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 21: 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

6.6 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 [177]. 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 higher or comparable to the typical 5% response rate reported in other software engineering surveys [176].

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 21 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

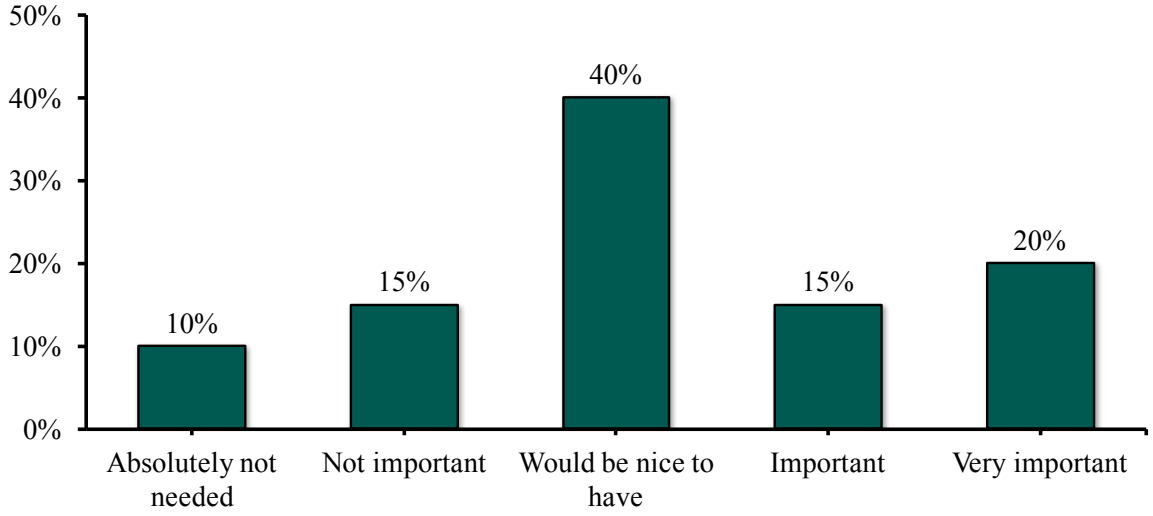


Figure 21: Survey responses regarding the importance of being able to automatically CI skip a commit.

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.

6.6.1 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[115, 121], we asked them to answer the question on a five-point likert-scale, ranging from 1= absolutely not needed to 5= very important. Figure 21 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 indicate that having a technique to help automatically CI skip commits is needed by developers.

6.6.2 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 participants' 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 [43]. 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 following, 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."*

6.6.3 Why do Developers Skip the CI Process?

We followed the same approach mentioned 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 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 take 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 4 (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".*

6.7 Discussion

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

6.7.1 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 why) 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 we 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 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 these types 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 annotation. The following shows an example of 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.

6.7.2 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 [66], 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. Specifically, we extract all the source code Java file pairs (modified and original) for each file touched in a commit. Then, we use

Table 22: 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**

ChangeDistiller to extract the fine-grained source code changes between each pair of revisions. ChangeDistiller takes as input the pair of revision file and creates two Abstract Syntax Trees (ASTs) that are used to compare the 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 on 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.** 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 6.7.1.

- **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 6.7.1.

- **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 6.7.1.

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 17. We measure performance using precision, recall, F-measure, and ROC.

Table 22 shows the result of our rule-based technique with 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 22). Using the additional information from ChangeDistller slightly improved the overall performance of our rule-based approach, 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 was 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.

Overall, although we see that code analysis does help, its improvement in performance is not 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 approach 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 significant, in the next section, we detail a prototype tool that can automatically detect CI skip commits using our devised rules.

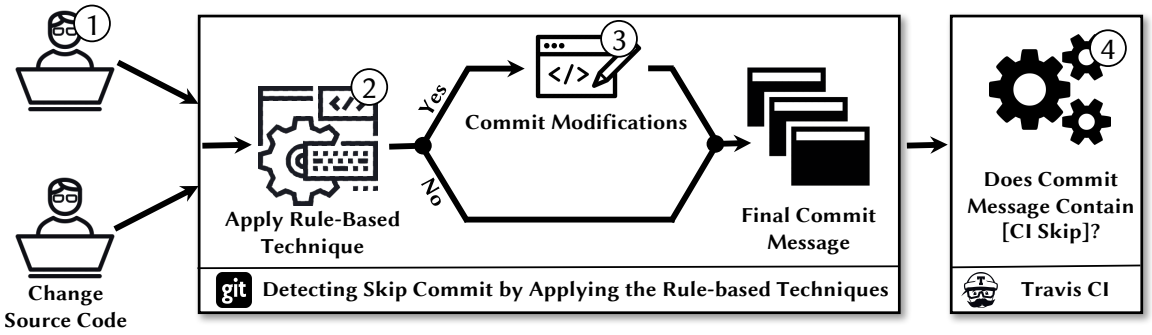


Figure 22: The workflow of the designed CI-Skipper tool.

6.8 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 resource savings. However, as prior work has shown [86], 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 [38].

Figure 22 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 23 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, marking commits that fit our rules with `[ci skip]`.

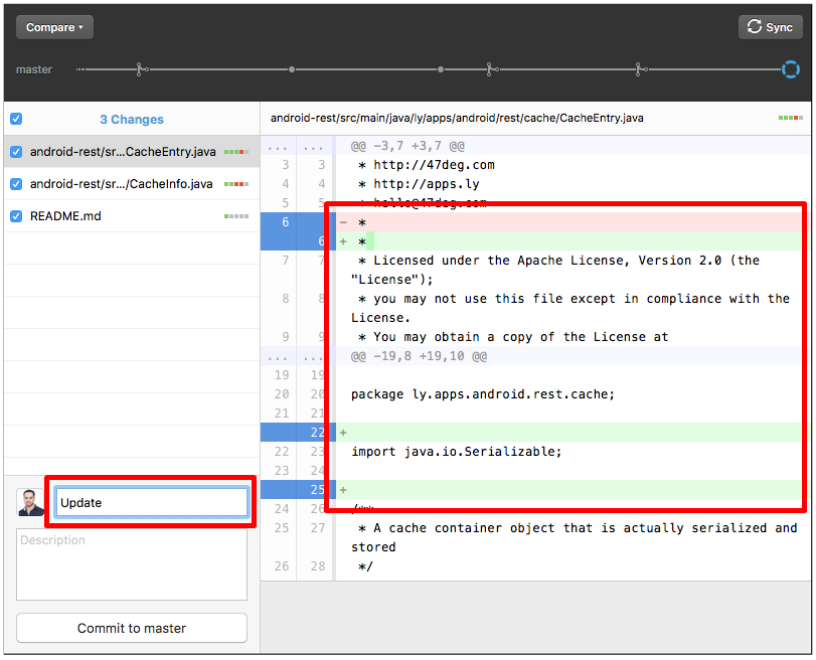
6.9 Related Work

In this section, we present the work most related to the study in this chapter. 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.

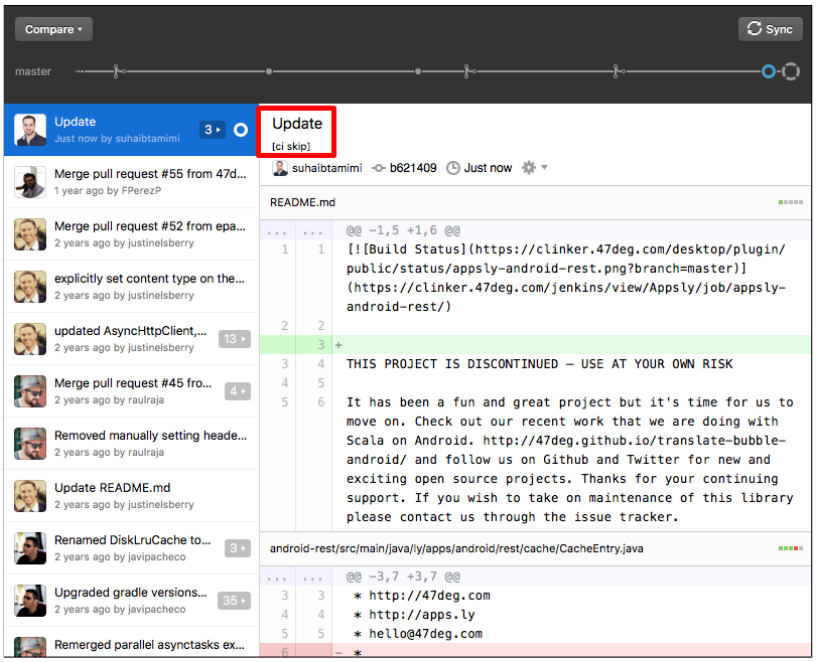
6.9.1 Improvement of CI Technology.

There is a limited number of studies that investigate the possibility to improve CI tools. Brandtner *et al.* [37] 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.* [39] 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 [61, 125]. Downs *et al.* [59] 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 [83] used classifiers to predict whether a build would pass a certification process. Rausch *et al.* [158] 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.* [172] 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 [211] 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.* [209] 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.



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



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

Figure 23: 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.

Miller [129] 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 the aforementioned studied, 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.

6.9.2 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 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.* [86] 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.* [85] 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.* [110] 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.* [25] 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.* [196] 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.* [205] studied the impact of using CI on software quality. CI detected bugs that are in a few files. Santos and Hindle [167] 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.

6.10 Threats to Validity

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

6.10.1 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 [194]. 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 [8].

To gain insight about 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 [176].

6.10.2 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 6.4.2, 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 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 [69]. However, since our findings are based on a large number of projects, we expect the effect of the Travis CI configurations to be minimal.

6.10.3 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 [86]. 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].

[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

6.11 Chapter Summary

In this chapter, we study CI skip commits that developers tend not to build a project on. We analyze the commits history of 392 open source Java projects provided by TravisTorren dataset [26]. 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.

Chapter 7

Conclusions and Future Work

The work presented in this thesis has emerged from the observation that crowdsourcing platforms have become popular in software development and that software developers increasingly rely on these platforms. In this thesis, we focused on some of the most commonly used and representative crowdsourcing platforms in the software engineering domain. We described and reported on a series of empirical studies that investigate the type, impact, and the adoption of crowdsourced knowledge in the software development process. Our finding showed that developers often use crowdsourcing platforms to share and reuse knowledge on these platforms. Furthermore, reusing crowdsourcing knowledge can increase dependency overhead and/or increasing maintenance effort and therefore negatively impact software quality. In this chapter, we conclude the thesis by summarizing the main work and contribution in each chapter of the thesis. At the end of the chapter, we discussed some directions for future research.

7.1 Conclusion and Findings

Crowdsourcing has become an integrated part of many of today's software development processes. Recent studies including the ones presented in this thesis have shown that crowdsourcing improves productivity, reducing time-to-market. However, like any solution, crowdsourcing brings with it many challenges such as quality issues, scalability and performance issues, maintenance issues and even legal issues. The research presented in this thesis focuses on helping software developers, who use the crowd, to build high-quality software. We present several large-scale empirical studies involving some of the most popular and representative crowdsourcing platforms including Stack Overflow and *npm*. This research has shown that reusing knowledge from these crowdsourcing platforms has the potential to assist software development practices, specifically through source code reuse. However, at the same time, this knowledge reuse can also affect the quality of the

software in several ways, such as increasing dependency overhead and/or increasing maintenance effort. Based on these findings, this thesis attempts to improve and mitigate the risks of relying on the crowd in software development. It examines the possibility of enhancing the CI service to ensure the quality of reused code from the crowd. More specifically, the presented research provides the following main contributions:

Chapter 3: Understanding the Usage of Crowdsourced Knowledge.

In Chapter 3, we present the first in-depth study of how developers use the crowdsourcing knowledge from Stack Overflow. We perform a qualitative study by analyzing approximately 1,400 Stack Overflow related commits. Our findings show that Stack Overflow provides software developers with knowledge related to topics such as development tools, APIs usage, and operating systems. Surprisingly, developers use Stack Overflow also as a communication channel to receive user feedback about their software. Our study also shows that the crowd was the most helpful on topics related to development tools and programming language.

Chapter 4: The Impact of Reused Source Code from Crowdsourcing Platforms.

The thesis presents the first empirical study to examine the reuse of source code from crowdsourcing platforms such as Stack Overflow. We first proposed an approach to detect reused code from Stack Overflow in software projects. We show that 1) the amount of reused Stack Overflow code varies for different mobile apps, 2) feature additions and enhancements in apps are the main reasons for code reuse from Stack Overflow, 3) mid-age and older apps reuse Stack Overflow code mostly later on in their project lifetime and 4) that in smaller teams/apps, more experienced developers reuse code, whereas, in larger teams/apps, the less experienced developers reuse code the most. In addition, we found that the percentage of bugs increases after reusing code from Stack Overflow.

Chapter 5: Examining the Type of Constructed Knowledge on Crowdsourcing Platforms.

We define and examine the use of trivial packages in software projects. We show that 1) trivial packages are commonly used in JavaScript and Python projects. 2) Developers believe that trivial packages provide them with well-test and implemented code. 3) We also found that developers believe that using trivial packages tends to increase the dependency overhead of their software. Our analysis also showed that some of the trivial packages have their own dependencies. For example,

11.5% of the trivial packages have more than 20 dependencies. Our findings illustrate that the use of crowd can prove itself in mitigating the lack of standards JavaScript library.

Chapter 6: Improving the efficiency of Reused Crowdsourced Knowledge.

To help developers increase the efficiency of reusing crowdsourcing knowledge, we proposed a technique to improve the continuous integration process through the detection of commits that can be CI skipped. We first apply a qualitative analysis to extract five rules that we used to build a rule-based approach to detect CI skip commits. Our study shows that the proposed technique can save on average 18.16% of required builds and save an average of 917 minutes in build time per projects. Our findings in this contribution show that continuous integration tools can be improved and adapted to help ensure the quality of reused source code from crowdsourcing platforms.

7.2 Future Work

Although this Ph.D. work has made many significant contributions towards understanding the impact of reusing crowdsourced knowledge on software development, many different avenues for future work remain unexplored. We summarize some of the main directions for future work.

7.2.1 Investigation the Relation between Different Crowdsourcing Platform

The first part of this thesis focused on understanding the reuse of source code snippets and knowledge from crowdsourcing platforms such as Stack Overflow. We found that developers copy-and-paste source code from these crowdsourcing platforms into their projects. In the second part of the thesis, we explored different platforms that provide reusable source code packages, Node Package Manager (*npm*). More specifically, we conducted a large-scale empirical study to examine the reasons why developers use trivial packages from *npm*, i.e., packages that contain code that a developer can easily code him/herself and hence is not worth taking on an extra dependency for. We found that developers use these trivial packages from *npm* because they provide them with well-tested code. During our investigation of these two platforms, we observed that there is a common shift in the way developers use code from crowdsourcing platforms (Stack Overflow vs. *npm*). Currently, developers tend to publish reusable code as a package compared to posting code snippet on Stack Overflow. For future work, an additional analysis of the relationship between source code snippet and published packages on *npm* could help to increase our understanding of how developers prefer one type of crowdsourcing platform over others.

7.2.2 Improving the Structure of Crowdsourcing Platforms

In addition to our main findings in Chapter 4, we believe that the proposed technique of detecting code reuse from Stack Overflow can be further extended in future work to improve and enable the investigation of other phenomena of reusing code from crowdsourcing platforms. For example, our approach can be extended to provide a *bi-directional traceability.* Currently, knowledge, including code snippets, from crowdsourcing platforms and software project code repositories remains in information silos, preventing feedback loops across these knowledge resources. As a result, no mechanism is in place to allow for notifications or monitoring changes, which might have an effect on either source. For example, in response to the original question posted on Stack Overflow, an improved solution is posted or a problem (e.g., vulnerability) might have been discovered. However, establishing and maintaining such links will require adequate tool support. Thus, our approach may help improve traceability of Stack Overflow code reused in mobile apps. Another example is that our proposed approach can be used to *improved content and quality rating.* Information of actual code reuse from the Stack Overflow source code can be used to improve the rating mechanism on Stack Overflow. The frequency and context of code reuse from code snippets originating from Stack Overflow can be used to improve and establish more meaningful rating and tagging mechanisms for Stack Overflow posts, depending on actual real-world usage scenarios. Future research could consider the implementation of our approach to examine these ideas.

7.2.3 Replication in an Industrial Setting

The results included in this thesis showed that reusing crowdsourcing knowledge affects the quality of software. However, these results are based on analyzing only open source projects. While we have done our best to select appropriate representative large crowdsourcing platforms such as Stack Overflow and *npm* and data analysis techniques in order to reduce the threats to internal validity, we believe that practitioners need to understand how this reusing crowdsourcing knowledge impact their projects qualities and what challenges arise when relying on such crowdsourcing knowledge. A future research that investigate and study the impact reuse of crowdsourcing knowledge in an industrial setting would allow us to further generalize our results.

7.2.4 Other Type of Crowdsourced Knowledge

While this thesis work has focused on the impact of reusing crowdsourcing knowledge on software quality in form of reusing source code, however, the source code is not the only form of knowledge created by the crowd. For example, on Stack Overflow developers generate crowdsourcing knowledge

in form of discussion about development techniques. This knowledge is in free-text form or/and contains only code snippets that can be reused to support software development decision for example design decisions. Recent work examines the use of textual crowdsourcing knowledge to recommend a specific library for developers [49]. Another example of reusing non-source code crowdsourcing knowledge is the use of apps' user reviews on app stores. In fact, there are several proposed approaches to help app developers improving their apps through using users' reviews (e.,g. [124]). Future work should explore the reuse of such non-source code knowledge in order to identify its impact on software quality and maintainability.

7.2.5 Improving of Using Test Generation and Quality Management for Crowdsourced Knowledge/Code

The Chapter 6 of thesis work showed that the performance of existing continuous integration techniques can be greatly improved to help ensure the quality of software that uses crowdsourcing knowledge. Other quality insurance technique can be used as well, for example, test generation techniques. In fact, more broadly, future research would be conducted to systematically investigate and develop techniques that can improve the quality management of crowdsourced knowledge/code. Ideally, developing a framework that can detect and analyze such reused knowledge/code and suggest the best action to take. For example, the framework may suggest a code review, or other relevant code (since many of these crowdsourced code snippets are incomplete), or a comprehensive test suite, etc. We believe that achieving this goal will significantly improve the quality of reused crowdsourced knowledge and increase developers' confidence in reused knowledge.

7.2.6 Enhancing the Detection of Skip commits Through the Use of Machine learning

In Chapter 6, we propose the use of a rule-based technique to detect CI skipped commits. A major problem with using the rule-based technique is that it is general and it does not consider the specific characteristic of a project. In the case of detecting CI skip commits, for example, different projects have a different type of files used for documentation and also their team's development context is different. To overcome these limitations, future work should explore other detection techniques. For example, investigating the use of supervised machine learning to assist developers in automatically identifying CI skip commits.

Bibliography

[1] Stackexchange api. `https://api.stackexchange.com//`. (accessed: 2016-03-06).

[2] Topcoder | deliver faster through crowdsourcing, 2001. (accessed: 2017-06-19).

[3] Stack overflow. `http://stackoverflow.com/`, 2008. (accessed: 2016-03-06).

[4] npm. `https://www.npmjs.com/`, 2011. accessed: 2017-06-19.

[5] F-Droid | Free and Open Source Android App Repository, https://f-droid.org/, 2015.

[6] utest - the professional network for testers, June 2017. (accessed: 2017-06-19).

[7] P. Abate, R. Di Cosmo, J. Boender, and S. Zacchiroli. Strong dependencies between software components. In *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, ESEM '09, pages 89–99. IEEE Computer Society, 2009.

[8] R. Abdalkareem. Reasons and drawbacks of using trivial npm packages: The developers' perspective. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 1062–1064. ACM, 2017.

[9] R. Abdalkareem, S. Mujahid, E. Shihab, and J. Rilling. Which commits can be ci skipped? *IEEE Transactions on Software Engineering*, pages 1–1, 2019.

[10] R. Abdalkareem, O. Nourry, S. Wehaibi, S. Mujahid, and E. Shihab. Why do developers use trivial packages? an empirical case study on npm. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 385–395. ACM, 2017.

[11] R. Abdalkareem, V. Oda, S. Mujahid, and E. Shihab. npm and pypip surveys responses and dataset. `http://das.encs.concordia.ca/publications/npm-and-pypip-data/`.

[12] R. Abdalkareem, E. Shihab, and J. Rilling. On code reuse from stackoverflow : An exploratory study on android apps. *Information and Software Technology*, 88(C):148–158, 2017.

[13] R. Abdalkareem, E. Shihab, and J. Rilling. What do developers use the crowd for? a study using stack overflow. *IEEE Software*, 34(2):53–60, 2017.

[14] O. Alonso, D. E. Rose, and B. Stewart. Crowdsourcing for relevance evaluation. *SIGIR Forum*, 42(2):9–15, 2008.

[15] A. Bacchelli, L. Ponzanelli, and M. Lanza. Harnessing stack overflow for the ide. In *Proceedings of the Third International Workshop on Recommendation Systems for Software Engineering*, pages 26–30. IEEE Press, 2012.

[16] A. S. Badashian, A. Hindle, and E. Stroulia. Crowdsourced bug triaging. In *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*, pages 506–510, 2015.

[17] K. Bajaj, K. Pattabiraman, and A. Mesbah. Mining questions asked by web developers. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR, pages 112–121, 2014.

[18] S. Baltes and S. Diehl. Usage and attribution of stack overflow code snippets in github projects. *Empirical Software Engineering*, pages 1–37, Oct 2018.

[19] A. Barua, S. W. Thomas, and A. E. Hassan. What are developers talking about? An analysis of topics and trends in Stack Overflow. *Empirical Software Engineering (EMSE)*, 19(3):619–654, Nov 2014.

[20] O. Barzilay, C. Treude, and A. Zagalsky. Facilitating crowd sourced software engineering via stack overflow. In *Finding Source Code on the Web for Remix and Reuse*, pages 289–308. Springer, 2013.

[21] O. Barzilay and C. Urquhart. Understanding reuse of software examples: A case study of prejudice in a community of practice. *Information and Software Technology*, 56(12):1613–1628, 2014.

[22] V. R. Basili, L. C. Briand, and W. L. Melo. How reuse influences productivity in object-oriented systems. *Communications of the ACM*, 39(10):104–116, October 1996.

[23] G. Bavota, G. Canfora, M. D. Penta, R. Oliveto, and S. Panichella. The evolution of project inter-dependencies in a software ecosystem: The case of apache. In *Proceedings of the 2013 IEEE International Conference on Software Maintenance*, ICSM '13, pages 280–289. IEEE Computer Society, 2013.

[24] A. Begel, J. D. Herbsleb, and M.-A. Storey. The future of collaborative software development. In *Proceedings of the ACM 2012 Conference on Computer Supported Cooperative Work Companion*, CSCW '12, pages 17–18. ACM, 2012.

[25] 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.

[26] 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.

[27] C. Bird, N. Nagappan, B. Murphy, H. Gall, and P. Devanbu. Don't touch my code!: examining the effects of ownership on software quality. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering (ESEC/FSE)*, pages 4–14, 2011.

[28] T. F. Bissyandé, F. Thung, D. Lo, L. Jiang, and L. Réveillère. 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.

[29] M. Blais. snakefood: Python dependency graphs. `http://furius.ca/snakefood/`. (accessed: 09-23-2018).

[30] R. Bloemen, C. Amrit, S. Kuhlmann, and G. Ordóñez Matamoros. Gentoo package dependencies over time. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR '14, pages 404–407. ACM, 2014.

[31] C. Bogart, C. Kastner, and J. Herbsleb. When it breaks, it breaks: How ecosystem developers reason about the stability of dependencies. In *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering Workshop*, ASEW '15, pages 86–89. IEEE Computer Society, 2015.

[32] C. Bogart, C. Kästner, J. Herbsleb, and F. Thung. How to break an api: Cost negotiation and community values in three software ecosystems. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 109–120. ACM, 2016.

[33] C. Bogart, C. Kästner, J. Herbsleb, and F. Thung. How to break an api: Cost negotiation and community values in three software ecosystems. In *Proceedings of the 2016 24th ACM*

SIGSOFT International Symposium on Foundations of Software Engineering, FSE '16, pages 109–120. ACM, 2016.

[34] S. Bonnemann. Dependency hell just froze over. `https://speakerdeck.com/boennemann/dependency-hell-just-froze-over`, September 2015. (accessed: 08-10-2016).

[35] Bower. Bower a package manager for the web. `https://bower.io/`, 2012. (accessed: 08-23-2016).

[36] J. Brandt, P. J. Guo, J. Lewenstein, M. Dontcheva, and S. R. Klemmer. Two studies of opportunistic programming: interleaving web foraging, learning, and writing code. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1589–1598, 2009.

[37] 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.

[38] 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.

[39] 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.

[40] K. Charmaz. *Constructing Grounded Theory: A Practical Guide through Qualitative Analysis*. Sage Publications Inc, second edition, 2014.

[41] A. T. Chatfield and U. Brajawidagda. Crowdsourcing hazardous weather reports from citizens via twittersphere under the short warning lead times of ef5 intensity tornado conditions. In *2014 47th Hawaii International Conference on System Sciences*, pages 2231–2241. IEEE, 2014.

[42] K. Chen, P. Liu, and Y. Zhang. Achieving accuracy and scalability simultaneously in detecting application clones on android markets. In *Proceedings of the 36th International Conference on Software Engineering*, pages 175–186. ACM, 2014.

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

[44] J. Cordeiro, B. Antunes, and P. Gomes. Context-based recommendation to support problem solving in software development. In *2012 Third International Workshop on Recommendation Systems for Software Engineering (RSSE)*, pages 85–89. IEEE, 2012.

[45] A. Cruz and A. Duarte. npms. `https://npms.io/`, 01 2017. (accessed: 02-20-2017).

[46] 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.

[47] 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.

[48] J. Davies, D. M. German, M. W. Godfrey, and A. Hindle. Software bertillonage. In *Proceeding of the 8th working conference on Mining software repositories - MSR '11*, page 183. ACM Press, 2011.

[49] F. L. de la Mora and S. Nadi. An empirical study of metric-based comparisons of software libraries. In *Proceedings of the 14th International Conference on Predictive Models and Data Analytics in Software Engineering*, PROMISE'18, pages 22–31. ACM, 2018.

[50] C. R. B. de Souza and D. F. Redmiles. An empirical study of software developers' management of dependencies and changes. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 241–250. ACM, 2008.

[51] L. B. de Souza, E. C. Campos, and M. d. A. Maia. Ranking crowd knowledge to assist software development. In *Proceedings of the 22nd International Conference on Program Comprehension*, pages 72–82, 2014.

[52] A. Decan, T. Mens, and M. Claes. On the topology of package dependency networks: A comparison of three programming language ecosystems. In *Proccedings of the 10th European Conference on Software Architecture Workshops*, ECSAW '16, pages 21:1–21:4. ACM, 2016.

[53] A. Decan, T. Mens, and M. Claes. An empirical comparison of dependency issues in oss packaging ecosystems. In *Proccedings of the 24th International Conference on Software Analysis, Evolution, and Reengineering*, SANER '17, pages 2–12. IEEE, 2017.

[54] A. Decan, T. Mens, and E. Constantinou. On the impact of security vulnerabilities in the npm package dependency network. In *Proceedings of the 15th International Conference on Mining Software Repositories*, MSR '18, pages 181–191. ACM, 2018.

[55] A. Decan, T. Mens, and P. Grosjean. An empirical comparison of dependency network evolution in seven software packaging ecosystems. *Empirical Software Engineering*, pages 1–36, Feb 2018.

[56] A. Decan, T. Mens, P. Grosjean, et al. When github meets cran: An analysis of inter-repository package dependency problems. In *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering*, volume 1 of *SANER '16*, pages 493–504. IEEE, 2016.

[57] R. Di Cosmo, D. Di Ruscio, P. Pelliccione, A. Pierantonio, and S. Zacchiroli. Supporting software evolution in component-based foss systems. *Science of Computer Programming*, 76(12):1144–1160, 2011.

[58] M. Dogguy, S. Glondu, S. Le Gall, and S. Zacchiroli. Enforcing type-safe linking using inter-package relationships. *Studia Informatica Universalis.*, 9(1):129–157, 2011.

[59] 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.

[60] 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.

[61] S. Dösinger, 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.

[62] H. Erdogmus, M. Morisio, and M. Torchiano. On the effectiveness of the test-first approach to programming. *IEEE Transactions on Software Engineering*, 31(3):226–237, 2005.

[63] J. Eyolfson, L. Tan, and P. Lam. Do time of day and developer experience affect commit bugginess? In *Proceedings of the 8th Working Conference on Mining Software Repositories*, pages 153–162, 2011.

[64] J. Fleiss. The measurement of interrater agreement. *Statistics methods for rates and proportions*, pages 212–236, 1981.

[65] 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.

[66] 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.

[67] B. Flyvbjerg. Five misunderstandings about case-study research. *Qualitative Inquiry*, 12(2):219–245, 2006.

[68] T. Fuchs. What if we had a great standard library in javascript? – medium. `https://medium.com/@thomasfuchs/what-if-we-had-a-great-standard-library-in-javascript-52692342ee3f.pw7d4cq8j`, Mar 2016. (accessed: 02-24-2017).

[69] 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.

[70] R. E. Gallardo-Valencia and S. E. Sim. Software problems that motivate web searches. In S. E. Sim and R. E. Gallardo-Valencia, editors, *Finding Source Code on the Web for Remix and Reuse*, chapter 13, pages 253–270. 2013.

[71] D. German, B. Adams, and A. Hassan. Programming language ecosystems: the evolution of r. In *Proceedings of the 17th European Conference on Software Maintenance and Reengineering*, CSMR '13, pages 243–252. IEEE, 2013.

[72] D. M. German, M. Di Penta, Y.-G. Gueheneuc, and G. Antoniol. Code siblings: Technical and legal implications of copying code between applications. In *2009 6th IEEE International Working Conference on Mining Software Repositories*, pages 81–90. IEEE, 2009.

[73] G. Gousios. The ghtorent dataset and tool suite. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 233–236. IEEE Press, 2013.

[74] G. Gousios and A. Zaidman. A dataset for pull-based development research. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR '14, pages 368–371. ACM, 2014.

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

[76] E. C. Groen, N. Seyff, R. Ali, F. Dalpiaz, J. Doerr, E. Guzman, M. Hosseini, J. Marco, M. Oriol, A. Perini, and M. Stade. The crowd in requirements engineering: The landscape and challenges. *IEEE Software*, 34(2):44–52, Mar 2017.

[77] J. Gui, S. Mcilroy, M. Nagappan, and W. G. J. Halfond. Truth in advertising: The hidden cost of mobile ads for software developers. In *Proceedings of the 37th International Conference on Software Engineering*, ICSE'15, pages 100–110. IEEE Press, 2015.

[78] S. Haefliger, G. Von Krogh, and S. Spaeth. Code reuse in open source software. *Management Science*, 54(1):180–193, 2008.

[79] Q. Hanam, F. S. d. M. Brito, and A. Mesbah. Discovering bug patterns in javascript. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '16, pages 144–156. ACM, 2016.

[80] D. Haney. Npm & left-pad: Have we forgotten how to program? `http://www.haneycodes.net/npm-left-pad-have-we-forgotten-how-to-program/`, March 2016. (accessed: 08-10-2016).

[81] R. Harris. Small modules: it's not quite that simple. `https://medium.com/@Rich_Harris/small-modules-it-s-not-quite-that-simple-3ca532d65de4`, Jul 2015. (accessed: 08-24-2016).

[82] B. Hartmann, D. MacDougall, J. Brandt, and S. R. Klemmer. What would other programmers do: Suggesting solutions to error messages. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '10, pages 1019–1028. ACM, 2010.

[83] 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.

[84] Hemanth.HM. One-line node modules -issue#10- sindresorhus/ama. `https://github.com/sindresorhus/ama/issues/10`, 2015. (accessed: 08-10-2016).

[85] 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.

[86] 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.

[87] 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.

[88] R. Hoffmann, J. Fogarty, and D. S. Weld. Assieme: finding and leveraging implicit references in a web search interface for programmers. In *Proceedings of the 20th annual ACM symposium on User interface software and technology*, pages 13–22, 2007.

[89] R. Holmes, R. J. Walker, and G. C. Murphy. Strathcona example recommendation tool. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 237–240, 2005.

[90] J. Howe. Crowdsourcing: Crowdsourcing: A definition. `http://crowdsourcing.typepad.com/cs/2006/06/crowdsourcing_a.html`, jUNE 2006. (accessed: 12-31-2016).

[91] J. E. Hunter. The desperate need for replications. *Journal of Consumer Research*, 28(1):149–158, 2001.

[92] K. Inoue, Y. Sasaki, P. Xia, and Y. Manabe. Where does this code come from and where does it go? -integrated code history tracker for open source systems -. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 331–341. IEEE Press, 2012.

[93] B. Johnson. Ieee software blog: Ieee march/april issue, blog, and se radio summary. `http://blog.ieeesoftware.org/2017/06/ieee-marchapril-issue-blog-and-se-radio.html`. (accessed: 12-16-2018).

[94] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner. Do code clones matter? In *Proceedings of the 31st International Conference on Software Engineering*, pages 485–495. IEEE Computer Society, 2009.

[95] J. Kabbedijk and S. Jansen. Steering insight: An exploration of the ruby software ecosystem. In *Proceedings of the Second International Conference of Software Business*, ICSOB '11, pages 44–55. Springer, 2011.

[96] 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.

[97] 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.

[98] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering (TSE)*, 28(7):654–670, 2002.

[99] C. J. Kapser and M. W. Godfrey. "cloning considered harmful" considered harmful: patterns of cloning in software. *Empirical Software Engineering*, 13(6):645–692, 2008.

[100] N. Kaufmann, T. Schulze, and D. Veit. More than fun and money. worker motivation in crowdsourcing-a study on mechanical turk. In *AMCIS*, volume 11, pages 1–11, 2011.

[101] N. Kawamitsu, T. Ishio, T. Kanda, R. G. Kula, C. De Roover, and K. Inoue. Identifying Source Code Reuse across Repositories Using LCS-Based Source Code Similarity. In *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, pages 305–314. IEEE, 2014.

[102] I. Keivanloo, J. Rilling, and Y. Zou. Spotting working code examples. In *Proceedings of the 36th International Conference on Software Engineering - ICSE 2014*, pages 664–675, 2014.

[103] B. Kogut and A. Metiu. Open-source software development and distributed innovation. *Oxford review of economic policy*, 17(2):248–264, 2001.

[104] R. Koschke. Survey of research on software clones. In R. Koschke, E. Merlo, and A. Walenstein, editors, *Duplication, Redundancy, and Similarity in Software*, number 06301 in Dagstuhl Seminar Proceedings, 2007.

[105] R. G. Kula, C. D. Roover, D. M. German, T. Ishio, and K. Inoue. A generalized model for visualizing library popularity, adoption, and diffusion within a software ecosystem. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, volume 00, pages 288–299, March 2018.

[106] K. Lakhani, D. A. Garvin, and E. Lonstein. Topcoder (a): Developing software through crowdsourcing. *Harvard Business School General Management Unit Case*, (610-032), 2010.

[107] T. D. LaToza, W. B. Towne, A. van der Hoek, and J. D. Herbsleb. Crowd development. In *2013 6th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*, pages 85–88. IEEE, 2013.

[108] T. D. LaToza and A. van der Hoek. Crowdsourcing in software engineering: Models, motivations, and challenges. *IEEE Software*, 33(1):74–80, 2016.

[109] M. Lease and E. Yilmaz. Crowdsourcing for information retrieval. *SIGIR Forum*, 45(2):66–75, 2012.

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

[111] Libraries.io. Libraries.io - the open source discovery service. `https://libraries.io/`. (accessed: 05-20-2018).

[112] Libraries.io. Pypi. `https://libraries.io/pypi`, 2017. (accessed: 03-08-2017).

[113] W. Lim. Effects of reuse on quality, productivity, and economics. *IEEE Software*, 11(5):23–30, 1994.

[114] W. C. Lim. Effects of reuse on quality, productivity, and economics. *IEEE Software*, 11(5):23–30, 1994.

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

[116] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, M. Di Penta, R. Oliveto, and D. Poshyvanyk. Api change and fault proneness: a threat to the success of android apps. In *Proceedings of the 2013 9th joint meeting on foundations of software engineering*, pages 477–487. ACM, 2013.

[117] A. Lozano and M. Wermelinger. Assessing the effect of clones on changeability. In *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on*, pages 227–236, 2008.

[118] F. Macdonald. A programmer almost broke the internet last week by deleting 11 lines of code. ˙&+#http://www.sciencealert.com/how-a-programmer-almost-broke-the-internet-by-deleting-11-lines-of-code, March 2016. (accessed: 08-24-2016).

[119] E. D. S. Maldonado, R. Abdalkareem, E. Shihab, and A. Serebrenik. An empirical study on the removal of self-admitted technical debt. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 238–248. IEEE, 2017.

[120] K. Manikas. Revisiting software ecosystems research: a longitudinal literature study. *Journal of Systems and Software*, 117:84–103, 2016.

[121] 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.

[122] K. Mao, L. Capra, M. Harman, and Y. Jia. A survey of the use of crowdsourcing in software engineering. *Journal of Systems and Software*, 126:57–84, 2017.

[123] K. Mao, M. Harman, and Y. Jia. Crowd intelligence enhances automated mobile testing. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*, ASE '17, pages 16–26. IEEE Press, 2017.

[124] W. Martin, F. Sarro, Y. Jia, Y. Zhang, and M. Harman. A survey of app store analysis for software engineering. *IEEE Transactions on Software Engineering*, 43(9):817–847, September 2017.

[125] 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.

[126] S. McCamant and M. D. Ernst. Predicting problems caused by component upgrades. In *Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE '03, pages 287–296. ACM, 2003.

[127] G. Mezzetti, A. Møller, and M. T. Torp. Type regression testing to detect breaking changes in node. js libraries. In *In the Proceedings of the 32nd European Conference on Object-Oriented Programming*, ECOOP '18. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

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

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

[130] R. Minelli and M. Lanza. Software analytics for mobile applications–insights & lessons learned. In *Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on*, pages 144–153. IEEE, 2013.

[131] S. Mirhosseini and C. Parnin. Can automated pull requests encourage software developers to upgrade out-of-date dependencies? In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*, ASE 2017, pages 84–94, Piscataway, NJ, USA, 2017. IEEE Press.

[132] A. Mockus. Large-scale code reuse in open source software. In *Proceedings of the First International Workshop on Emerging Trends in FLOSS Research and Development*, FLOSS '07, pages 7–7. IEEE Computer Society, 2007.

[133] A. Mockus and L. G. Votta. Identifying reasons for software changes using historic databases. In *International Conference on Software Maintenance.*, pages 120–130, 2000.

[134] P. Mohagheghi, R. Conradi, O. M. Killi, and H. Schwarz. An empirical study of software reuse vs. defect-density and stability. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 282–292. IEEE Computer Society, 2004.

[135] I. J. Mojica, B. Adams, M. Nagappan, S. Dienst, T. Berger, and A. E. Hassan. A Large-Scale Empirical Study on Software Reuse in Mobile Apps. *IEEE Software*, 31(2):78–86, Mar. 2014.

[136] M. Mondal, C. K. Roy, and K. A. Schneider. A comparative study on the bug-proneness of different types of code clones. In *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*, pages 91–100, 2015.

[137] L. Moreno, G. Bavota, M. Di Penta, R. Oliveto, and A. Marcus. How can i use this method? In *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, volume 1, pages 880–890. IEEE, 2015.

[138] S. Mujahid, R. Abdalkareem, and E. Shihab. Studying permission related issues in android wearable apps. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 345–356. IEEE, 2018.

[139] S. Mujahid, G. Sierra, R. Abdalkareem, E. Shihab, and W. Shang. Examining user complaints of wearable apps: A case study on android wear. In *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 96–99. IEEE, 2017.

[140] S. Mujahid, G. Sierra, R. Abdalkareem, E. Shihab, and W. Shang. An empirical study of android wear user complaints. *Empirical Software Engineering*, 23(6):3476–3502, Dec 2018.

[141] D. Mujumdar, M. Kallenbach, B. Liu, and B. Hartmann. Crowdsourcing suggestions to programming problems for dynamic web development languages. In *CHI '11 Extended Abstracts on Human Factors in Computing Systems*, CHI EA '11, pages 1525–1530. ACM, 2011.

[142] R. Musson, J. Richards, D. Fisher, C. Bird, B. Bussone, and S. Ganguly. Leveraging the crowd: How 48,000 users helped improve lync performance. *IEEE Software*, 30(4):38–45, 2013.

[143] S. M. Nasehi, J. Sillito, F. Maurer, and C. Burns. What makes a good code example?: A study of programming Q&A in StackOverflow. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 25–34, Sept. 2012.

[144] npm. What is npm? | node package managment documentation. `https://docs.npmjs.com/getting-started/what-is-npm`, July 2016. (accessed: 08-14-2016).

[145] T. npm Blog. The npm blog changes to npm's unpublish policy. `http://blog.npmjs.org/post/141905368000/changes-to--unpublish-policy`, March 2016. (accessed: 08-11-2016).

[146] H. Orsila, J. Geldenhuys, A. Ruokonen, and I. Hammouda. Update propagation practices in highly reusable open source components. In *Proceedings of the 4th IFIP WG 2.13 International Conference on Open Source Systems*, OSS '08, pages 159–170, 2008.

[147] J. Patra, P. N. Dixit, and M. Pradel. Conflictjs: Finding and understanding conflicts between javascript libraries. In *Proceeding of the 40th International Conference on Software Engineering*, ICSE '18, pages 741–751. ACM, 2018.

[148] L. Ponzanelli. Exploiting crowd knowledge in the ide. Master's thesis, Università della Svizzera Italiana, 2012.

[149] L. Ponzanelli, A. Bacchelli, and M. Lanza. Leveraging crowd knowledge for software comprehension and development. In *Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on*, pages 57–66. IEEE, 2013.

[150] L. Ponzanelli, A. Bacchelli, and M. Lanza. Seahawk: Stack overflow in the ide. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 1295–1298. IEEE Press, 2013.

[151] L. Ponzanelli, G. Bavota, M. Di Penta, R. Oliveto, and M. Lanza. Mining StackOverflow to turn the IDE into a self-confident programming prompter. *Proceedings of 2014 the 11th Working Conference on Mining Software Repositories (MSR)*, pages 102–111, 2014.

[152] L. Ponzanelli, G. Bavota, M. Di Penta, R. Oliveto, and M. Lanza. Prompter: A self-confident recommender system. In *2014 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 577–580. IEEE, 2014.

[153] L. Ponzanelli, G. Bavota, M. Di Penta, R. Oliveto, and M. Lanza. Prompter turning the ide into a self-confident programming assistant. *Empirical Software Engineering*, pages 1–42, 2015.

[154] Python. Python testing tools taxonomy - python wiki. `https://wiki.python.org/moin/PythonTestingToolsTaxonomy`. (accessed: 05-16-2018).

[155] F. Rahman, C. Bird, and P. Devanbu. Clones: What is that smell? *Empirical Software Engineering*, 17(4-5):503–530, 2012.

[156] M. M. Rahman, C. K. Roy, and I. Keivanloo. Recommending insightful comments for source code using crowdsourced knowledge. In *Source Code Analysis and Manipulation (SCAM), 2015 IEEE 15th International Working Conference on*, pages 81–90. IEEE, 2015.

[157] M. M. Rahman, S. Yeasmin, and C. K. Roy. Towards a context-aware IDE-based meta search engine for recommendation about programming errors and exceptions. In *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, pages 194–203. IEEE, 2014.

[158] 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.

[159] B. Ray, D. Posnett, V. Filkov, and P. Devanbu. A large scale study of programming languages and code quality in github. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '14, pages 155–165. ACM, 2014.

[160] P. C. Rigby, D. M. German, and M.-A. Storey. Open source software peer review practices: A case study of the apache server. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 541–550. ACM, 2008.

[161] J. M. Rojas, T. D. White, B. S. Clegg, and G. Fraser. Code defenders: Crowdsourcing effective tests and subtle mutants with a mutation testing game. In *In the Proceedings of the 2017 IEEE/ACM 39th International Conference on Software Engineering*, ICSE'17, pages 677–688, May 2017.

[162] C. Rosen and E. Shihab. What are mobile developers asking about? A large scale study using stack overflow. *Empirical Software Engineering (EMSE)*, 2015.

[163] C. Rosen and E. Shihab. What are mobile developers asking about? a large scale study using stack overflow. *Empirical Software Engineering (EMSE)*, 21(3):1192–1223, 2016.

[164] C. K. Roy and J. R. Cordy. A survey on software clone detection research. Technical report, Technical Report 541, Queen's University at Kingston, 2007.

[165] I. J. M. Ruiz, M. Nagappan, B. Adams, and A. E. Hassan. Understanding reuse in the android market. In *Program Comprehension (ICPC), 2012 IEEE 20th International Conference on*, pages 113–122. IEEE, 2012.

[166] C. Sadowski, K. T. Stolee, and S. Elbaum. How developers search for code: a case study. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 191–201. ACM, 2015.

[167] 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.

[168] F. Sarro, A. AlSubaihin, M. Harman, Y. Jia, W. Martin, and Y. Zhang. Feature lifecycles as they spread, migrate, remain and die in app stores. *Requirements Engineering (RE'15), Ottawa, Canada*, 2015.

[169] T. W. Schiller and M. D. Ernst. Reducing the barriers to writing verified specifications. *SIGPLAN Not.*, 47(10):95–112, 2012.

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

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

[172] 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.

[173] 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.

[174] R. E. Silverman. Big companies try crowdsourcing - wsj. The Wall Street Journal. `http://www.wsj.com/articles/`, January 2012. (accessed: 12-31-2016).

[175] S. E. Sim, C. L. Clarke, and R. C. Holt. Archetypal source code searches: A survey of software developers and maintainers. In *Program Comprehension, 1998. IWPC'98. Proceedings., 6th International Workshop on*, pages 180–187, 1998.

[176] 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.

[177] 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.

[178] M. Sojer and J. Henkel. Code reuse in open source software development: Quantitative evidence, drivers, and impediments. *Journal of the Association for Information Systems*, 11(12):868–901, 2010.

[179] M. Squire. Should we move to stack overflow? measuring the utility of social media for developer support. In *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, volume 2, pages 219–228, 2015.

[180] K.-J. Stol and B. Fitzgerald. Two's company, three's a crowd: A case study of crowdsourcing software development. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 187–198. ACM, 2014.

[181] K. J. Stol, T. D. LaToza, and C. Bird. Crowdsourcing for software engineering. *IEEE Software*, 34(2):30–36, 2017.

[182] J. Stylos and B. A. Myers. Mica: A web-search tool for finding api components and examples. In *Visual Languages and Human-Centric Computing, 2006. VL/HCC 2006. IEEE Symposium on*, pages 195–202. IEEE, 2006.

[183] M. D. Syer, B. Adams, Y. Zou, and A. E. Hassan. Exploring the development of micro-apps: A case study on the blackberry and android platforms. In *SCAM 2011*, pages 55–64, 2011.

[184] M. D. Syer, M. Nagappan, A. E. Hassan, and B. Adams. Revisiting prior empirical findings for mobile apps: An empirical case study on the 15 most popular open-source android apps. In *Proceedings of the 13th Conference of the Center for Advanced Studies on Collaborative Research (CASCON)*, pages 283–297, 2013.

[185] 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.

[186] S. Thummalapenta and T. Xie. Parseweb: a programmer assistant for reusing open source code on the web. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 204–213, 2007.

[187] C. Treude, O. Barzilay, and M.-A. Storey. How do programmers ask and answer questions on the web? In *Proceeding of the 2011 33rd international conference on Software engineering (ICSE)*, pages 804–807, 2011.

[188] A. Trockman, S. Zhou, C. Kästner, and B. Vasilescu. Adding sparkle to social coding: An empirical study of repository badges in the npm ecosystem. In *Proceedings of the International Conference on Software Engineering*, ICSE '18, pages 511–522. ACM, 2018.

[189] J. Tsay, L. Dabbish, and J. Herbsleb. Influence of social and technical factors for evaluating contribution in github. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE '14, pages 356–366. ACM, 2014.

[190] M. Valiev, B. Vasilescu, and J. Herbsleb. Ecosystem-level determinants of sustained activity in open-source projects: A case study of the pypi ecosystem. In *Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE '18. ACM, 2018.

[191] B. Vasilescu. Academic papers using stack exchange data, (accessed: 03-10-2016).

[192] B. Vasilescu, V. Filkov, and A. Serebrenik. Stackoverflow and github: Associations between software development and crowdsourced knowledge. In *Proceedings of the 2013 International Conference on Social Computing (SocialCom' 13)*, pages 188–195. IEEE, 2013.

[193] 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.

[194] 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.

[195] 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*, ESEC/FSE 2015, pages 805–816. ACM, 2015.

[196] 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*, pages 805–816. ACM, 2015.

[197] R. Vliegendhart, E. Dolstra, and J. Pouwelse. Crowdsourced user interface testing for multimedia applications. In *Proceedings of the ACM Multimedia 2012 Workshop on Crowdsourcing for Multimedia*, CrowdMM '12, pages 21–22. ACM, 2012.

[198] T. Wang, G. Yin, H. Wang, C. Yang, and P. Zou. Automatic Knowledge Sharing Across Communities: A Case Study on Android Issue Tracker and Stack Overflow. In *2015 IEEE Symposium on Service-Oriented System Engineering*, pages 107–116. IEEE, 2015.

[199] S. Weber. *The Success of Open Source*. Harvard University Press, Cambridge, MA, USA, 2004.

[200] C. Williams. How one developer just broke node, babel and thousands of projects in 11 lines of javascript. `http://www.theregister.co.uk/2016/03/23/npm_left_pad_chaos`, March 2016. (accessed: 08-24-2016).

[201] E. Wittern, P. Suter, and S. Rajagopalan. A look at the dynamics of the javascript package ecosystem. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, pages 351–361. ACM, 2016.

[202] E. Wong, J. Yang, and L. Tan. Autocomment: Mining question and answer sites for automatic comment generation. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 562–567, 2013.

[203] Y. Wu, S. Wang, C.-P. Bezemer, and K. Inoue. How do developers utilize source code from stack overflow? *Empirical Software Engineering*, pages 1–33, Jul 2018.

[204] S. Yegulalp. Brendan eich: Javascript standard library will stay small, Mar 2016. (accessed: 2017-08-10).

[205] 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.

[206] A. Zagalsky, O. Barzilay, and A. Yehudai. Example overflow: Using social media for code recommendation. In *Proceedings of the Third International Workshop on Recommendation Systems for Software Engineering*, pages 38–42. IEEE Press, 2012.

[207] A. Zagalsky, C. G. Teshima, D. M. German, M.-A. Storey, and G. Poo-Caamaño. How the r community creates and curates knowledge: a comparative study of stack overflow and mailing lists. In *Proceedings of the 13th International Workshop on Mining Software Repositories*, MSR'16, pages 441–451. ACM, 2016.

[208] D. Zambonini. Testing and deployment. In O. Gregory, editor, *A Practical Guide to Web App Success*, chapter 20. Five Simple Steps, 2011. (accessed: 02-23-2017).

[209] 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.

[210] J. Zhu, M. Zhou, and A. Mockus. Patterns of folder use and project popularity: A case study of github repositories. In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '14, pages 30:1–30:4. ACM, 2014.

[211] 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.