Leveraging the Social Knowledge of Software Systems to Improve the Development Process

by

Mariam El Mezouar

A thesis submitted to the
School of Computing
in conformity with the requirements for
the degree of Doctor of Philosophy

Queen's University
Kingston, Ontario, Canada
September 2019

Abstract

The software development process is steadily evolving into a collaborative, distributed, and knowledge-intensive venture. This evolution has led to growing demands towards supportive technologies and tools. These demands have been addressed using a spectrum of social and communication channels, that are continuously evolving and growing. As a result, a wealth of knowledge has emerged around the intensive use of the social and communication channels, and can potentially be leveraged to benefit the software development process.

In this dissertation, we describe the empirical studies conducted to understand how mining the social knowledge of software systems can improve aspects of the development process. We look into three types of social knowledge: *a)* the crowd-sourced feedback from the end-users, *b)* the organizational structure of the developers, and *c)* the developers' wisdom. We investigate the value of monitoring the end-users' feedback posted on non-developer oriented social media platforms, such as Twitter. Our findings demonstrate that the end-users feedback can possibly allow an earlier discovery of the bugs that are critical to a large user base. Next, we look into how the developers self-organize on social coding platforms, such as GitHub. We are able to identify the organizational structures that are associated to better project outcomes, such as faster merge time of code contributions. To learn from the wisdom of the developers, we perform user studies for two purposes. First, we focus on the mechanisms adopted for the informal communication of the developers, specifically the chat-based services. Our investigation highlights the affordances and impacts of such mechanisms, to allow for a more efficient and informed use of the chat-based services.

Second, we ask the developers to reflect on their experiences using a set of communication channels. Results from the survey and from an associated quantitative analysis reveal interesting insights to guide the developers in setting up the communication flow of their projects. Overall, the research conducted in this dissertation highlights the value of accounting for the social aspects of the software development process, to improve the efficiency of the tasks and make informed decisions about the communication mechanisms of the software projects.

Acknowledgments

First and foremost, I would to express my deepest gratitude to my supervisor **Dr. Ying (Jenny) Zou**, for her mentorship, guidance, and insights. Dr. Zou has provided me with an environment that helped me grow into the researcher and person that I am today. She has also shown immense understanding and empathy when I embarked into the motherhood journey during my PhD.

I would also like to thank my **examining committee**, Dr. Michael Godfrey, Dr. Ali Etemad, and Dr. Jurgen Dingel, for dedicating the time to provide me with insightful feedback to improve my thesis.

I would also like to thank the incredible group of researchers at **the Software Re-engineering research group** including Dr. Feng Zhang, Dr. Daniel Alencar da Costa, Yu Zhao, Guoliang Zhao, Dr. Ehsan Noei, Dr. Shaohua Wang, Yonghui Huang, Pradeep Venkatesh, Taher Ahmed Ghaleb, Aidan Yang, Dr. Safwat Hassan, Osama Ehsan, and Omar El Zarif.

Lastly, I would like to thank **my parents** Ahmed and Jamila for being my first teachers in life, and for giving me enough love and support to last a lifetime and beyond. I am also thankful to **my sister** Lina, who by looking up to me pushed me to always be better. I am grateful to my parents-in-law as well for their support, particularly my mother-in-law Aziza who took care of my son when I returned to my PhD work full-time. My deepest gratitude to **my husband** and bestfriend Badrdine, who was truly my backbone throughout this journey. So thank you for always believing in me and being there for me. Finally, to my little boy **Riad**, you only made my PhD journey more colorful, challenging, and exciting. Thank you for teaching me patience, and for bringing me joy.

Statement of Originality

I am the primary author of this thesis. The research papers published in the context of this thesis are co-authored with Dr. Ying Zou, Dr. Feng Zhang, and Dr. Daniel Alencar Da Costa. Dr. Ying Zou supervised all the research work conducted in this thesis. The co-authors were instrumental to the success of my research through frequent meetings, and endless back-and-forths to improve my research approach, and the quality of the manuscripts produced.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The software development process within open source projects has been gradually evolving into a distributed, collaborative, and knowledge-based endeavor. Over the past decade, open source software development has transitioned from a predominantly solo activity of developing standalone applications, to a highly collaborative activity where boundaries between projects and teams are blurred [125]. This transition is further fueled by the social transformations (*i.e.,* social media) brought by the second generation of the Web (*i.e.,* Web 2.0). Social media is characterized by web content that is generated collaboratively and by mass participation [6]. Most social media platforms provide features, such as a profile, networking, private and public messaging, forums, blogging, commenting and media uploading. As a result, social and communication media have infiltrated the work of the developers, and transformed the flow of knowledge in software development to become more transparent and synchronous.

Previous studies [15][128] argue that the intensive use of social and communication media by the developer's community have led to a paradigm shift in collaborative development. Traditionally, mailing lists (coupled with issue tracking systems [14] and

IDEs [58]) have been central to the software development process, particularly the open source projects (*e.g.,* Linux, Apache). However, a shift of interest in favor of new social platforms that support project development was observed by Guzzi *et al.* [69]. This was further confirmed by Storey *et al.* [127], who proposed the timeline of adoption of popular social and communication media over time by the developers. The timeline highlights the shift of software development from a mostly solo technical activity, to a social and transparent technical activity. For instance, Tsay *et al.* [134] found that, in the code review process, both the technical quality of the code submitted and the social connection of the submitter to the project are considered during the evaluation process. Further studies reveal that the adoption of social features in the software development process supports various aspects, including but not limited to project awareness [66] and transparency [41], learning and problem solving [133], building reputations [25], and crowd-sourcing knowledge building [102].

1.1 Types of Social and Communication Channels

The list of the social and communication media used by the developers spans a wide spectrum, and each support different aspects of the software development process. In the course of this thesis, multiple social and communication channels, both developer and non-developer oriented, were studied to answer the raised questions.

1.1.1 Social coding

Social coding is a software development approach that advocates distributed software development using a set of features, such as networking among developers, issue

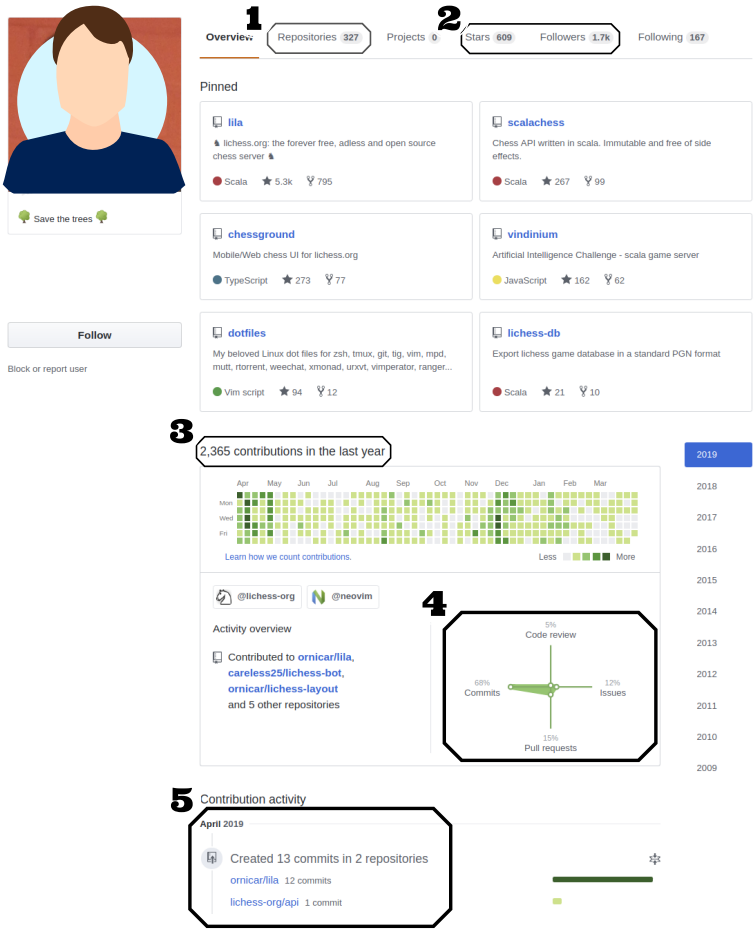


Figure 1.1: Example of a user profile on GitHub

tracking, and code management. GitHub[1] is one of the most famous platforms that support social coding. On social coding platforms, any developer can choose to make a contribution to a project. Then, the contribution is reviewed by the project maintainers before it can be merged. In addition to lowering the barriers for the first-time contributors, social coding platforms help developers build a reputation, and provide infrastructure to allow the developers to follow influential developers and projects. We show in Figure 1.1 an example of a GitHub user profile. The

[1]https://github.com/

highlighted areas *1* and *3* show the activity levels of the user, areas *4* and *5* capture the type of the user's activities, and area *2* highlights the popularity of the user. GitHub also provides issue tracking, pull requests, commits history, subscriptions to other users, and documentation.

1.1.2 Issue tracking systems

Issue tracking systems are important tools used by the development teams to guide the maintenance activities of the software projects. In open source software projects, issues reports are often submitted by the developers or the software users. Issue tracking systems allow their users to report, describe, track, prioritize, assign, and comment on issue reports and feature requests. Bugzilla[2] is one of the most popular issue tracking systems, used by large projects such as Mozilla[3] and Eclipse[4].

1.1.3 Micro-blogging

Micro-blogging is a broadcast medium that facilitates lightweight interactive communication among users. Micro-blogging has emerged as an important channel for real-time news and updates. For software projects, micro-blogging is used to update the users about new features in the system or the release of a new version. Developers use micro-blogging to network with other developers or ask questions. Twitter[5] is one of the most notable micro-blogging social networking service [3], and enables users to post 280-character long messages, called tweets. Twitter users take on to the

[2]https://www.bugzilla.org/
[3]https://www.mozilla.org
[4]https://www.eclipse.org
[5]https://twitter.com/

platform to express opinions, broadcast news, and share content in a real-time setting. As such, tweets can provide instant feedback on products to the product owners after launch of new versions, specifically to the development teams. For instance, an Apple FaceTime bug was reported on Twitter, after an iPhone user stumbled upon an eavesdropping flaw, more than a week before Apple took action [4]. Such incident highlights the need to monitor social media for user feedback, which is precisely what we investigate as part of this thesis.

1.1.4 The modern developer chatrooms

The chatrooms are designed to fulfill the communication needs of the developers, using messaging, file management, code sharing, and presence awareness. Contrary to other communication channels, such as the Q&A platforms, the developer chatrooms bring the developers together in an informal setting with equal participation opportunity for everyone. Some of the most notable chatrooms currently used by the developers are Slack[6] and Gitter[7]. The increasing adoption of the chatrooms reveals the interest of the developers in incorporating this type of communication channel into the software development process, sometimes replacing traditionally used tools such as email. Understanding how software teams use tools, such as Slack and Gitter is important to revisit the widely accepted ideas about communication in software development.

[6] https://slack.com
[7] https://gitter.im/

1.2 Research Challenges

The increasingly social nature of software development adds transparency to development process, enables the collection of more cues about the developers (*e.g.,* developer popularity) to support the code evaluation process [134], and helps the developers connect with each others and the software users. However, information overload can be a resulting challenge from the use of development platforms with social features, possibly affecting the productivity of the developers. Previous studies examined how developers use specific social and communication channels, such as Twitter [121], Stack Overflow [13][133], and GitHub [41]. These studies focused on identifying why developers adopt these channels (*e.g.,* improving awareness, supporting learning), the challenges they face (*e.g.,* information overload), and their coping strategies (*e.g.,* content and network pruning). We envision there are two limitations in the existing work: 1) limited efforts were deployed to explore the potential impact of having easy access to the software users on social media, or the patterns of self-organization of the developers in a social software development environment; and 2) the majority of the existing studies [25][77][136][137] are data-based empirical studies that aim at better understanding the use of the social and communication channels, with few studies [125] gathering the insights of the developers to answer the raised questions or to triangulate the findings.

In this dissertation, we aim to address the two aforementioned limitations. First, we examine the possible associations between using socially-enabled channels (*e.g.,* GitHub and Twitter), and the performance of the software development process (*e.g.,* efficiency of the bug fixing process). Second, we focus on collecting the wisdom of the developers to investigate aspects such as the informal communication (*e.g.,*

through chat-based communication), or to highlight the communication needs of the different development activities (*e.g.,* release planning).

1.3 Thesis Statement

This thesis aims to mine the social knowledge formed around the use of social and communication media by the developers, to understand the uses and impacts of such media in the software development process. In this thesis, we are first interested to examine the value of the crowd-sourced feedback available on Twitter. Second, we study the social organizational structure of teams, such as the team structures formed in GitHub. Lastly, we gather and analyze the developers' wisdom, to understand the developers' motives and uses of the communication media in the software development process. This thesis has the following research goal:

> **Research goal:** The social knowledge of software systems, such as the feedback on social media, the networks formed by the developers in social coding, and the developers' wisdom, contains valuable information that can benefit the software development process. By analyzing the data mined from the social and communication channels, the objective of this thesis is to provide techniques and recommendations to improve the software development activities (*e.g.,* bug fixing and code review), inform the design of the communication tools, and inform the choice of the communication channels of the projects.

By making use of the social knowledge formed around the software systems and gathered from the social outlets used by the developers, we identify the potential,

actual, or perceived impacts of the social and communication media on the development process. Based on our findings, we derive recommendations to the maintainers of software systems on how to best leverage the social knowledge when building software. Specifically, the thesis leverages three types of social knowledge: 1) **crowd-sourced feedback** from Twitter (chapter 3), 2) **social organizational structure** from GitHub (chapter 4), and 3) **developers' wisdom** about informal communication mechanisms (*i.e.,* chatrooms) (Chapter 5), and about the overall communication ecology adopted by the software projects (chapter6).

1.4 Thesis Objectives

We list in this section the three overarching objectives of this thesis. Figure 1.2 shows a high-level outline of the thesis, that maps the objectives listed below to the thesis chapters.

Objective I: Leveraging the crowd-sourced feedback to benefit the bug fixing process

When encountering an issue, technical users (e.g., developers) would usually file a formal issue report to the issue tracking systems. But non-technical end-users are more likely to express their opinions on social network platforms, such as Twitter. For software systems (*e.g.,* Firefox and Chrome) that have a high exposure to millions of non-technical end-users, it is important to monitor and solve issues observed by a large user base. Therefore, our first objective is to show the value of leveraging crowd-sourced feedback, available on social media, raise awareness about the issues affecting the users most, and potentially speed up the bug fixing process.

Objective II: Leveraging the social organizational structure in the code

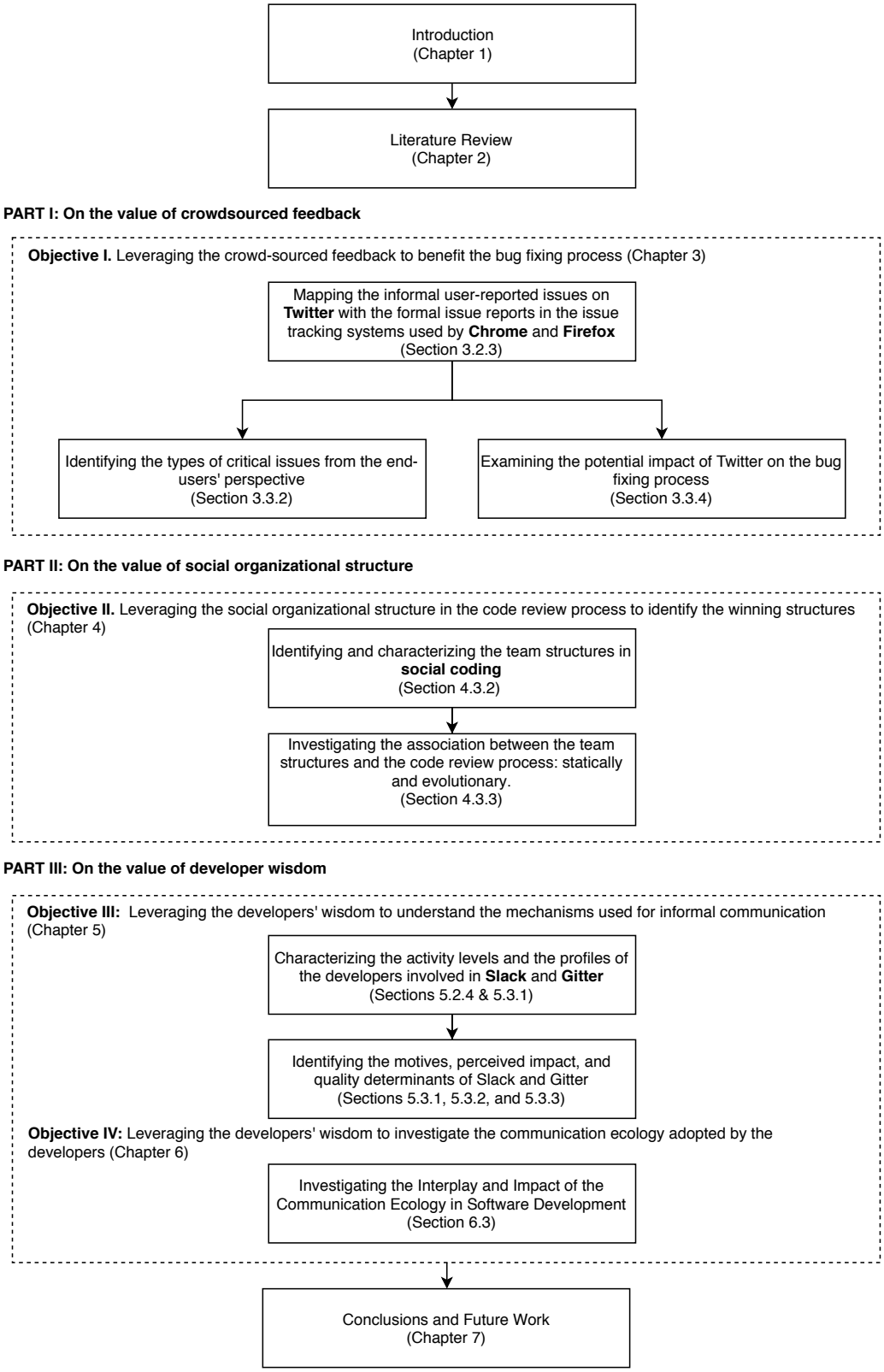


Figure 1.2: Thesis outline

review process to identify the winning structures

Social coding enables collaborative software development in virtual and distributed communities. Within the social coding platforms (*e.g.,* GitHub), the developers come together into teams to collaborate and learn from each other. The social organizational structure formed by the developers in social coding can reveal interesting insights about the developers' collaboration. Therefore, our second objective is to study the team structures formed by the developers, in a socially-enabled environment (*i.e.,* GitHub). The goal of the study is to identify the characteristics of both the desirable and the detrimental team structures, in order to help the project maintainers best organize their teams.

Objective III: Leveraging the developers' wisdom to understand the mechanisms used for informal communication

Software developers rely on communication channels to collaborate, coordinate, and learn from each others. In this thesis, we collect the wisdom of the developers on the use of the social and communication media through surveys. Specifically, we investigate the mechanisms used for the developers's informal communication (*i.e.,* the chatrooms), using Slack and Gitter as our subject systems. Our goal is to highlight when it is best to use both chatrooms, and to report on the impact of the chatrooms as perceived by the developers.

Objective IV: Leveraging the developers' wisdom to investigate the communication ecology adopted by the developers

With the wide spectrum of different communication channels to choose from, project maintainers need to select the best suited channels (in terms of quantity and

type of channels), to effectively discuss the software development activities. Therefore, we aim to ask the developers, through a survey, to reflect about their experiences using a set of communication channels, with respect to a set of software development activities, with the goal to provide recommendations regarding the nature, and number of channels most suited for each software development activity (*e.g.,* release planning).

1.5 Thesis Contributions

In this thesis, we show the importance of mining the social knowledge formed around the use of the social and communication media in software development. We demonstrate that leveraging said knowledge can help project maintainers to improve a set of development activities, and to inform the choices of the communication flow of the projects.

In particular, our main contributions are as follows:

- We propose an automated approach to map the user-reported issues on Twitter to formal issue reports on the issue tracking systems of Firefox and Chrome.The mapping of user-reported issues to formal issue reports confirms the value of crowd-sourced feedback in highlighting the issues most critical to a large user base, and in possibly speeding up the fixing of such issues (Chapter 3).

- We demonstrate that the social organization structure of teams in social coding can explain in part the performance of the code review process. We further systematically define and identify a set of team structures, and examine their popularity and associated performances. We are then able to provide a set of

recommendations to the project maintainers on the red flags of low performing team structures (Chapter 4).

- We collect the developers' experiences on the use of an open source (i.e., Gitter) and a proprietary (i.e., Slack) chat services. We obtain insights about the uses and perceived impact of chatrooms by the open communities (in both Slack and Gitter) and the corporate teams (mostly in Slack). We then provide recommendations that highlight when it is best to use either chatroom, depending on the needs of the projects (Chapter 5).

- From the developers' feedback, we provide recommendations on how to set up an efficient communication flow in software projects, in terms of the number and nature (*i.e.,* social, digital, or nondigital) of channels to be used in every developement activity. We also identify the most frequent and successful combination of channels (*e.g.,* Slack + Pull requests), by measuring the quantitative impact on the projects' performance indicators (*e.g.,* bug fixing time) (Chapter 6).

1.6 Thesis Outline

We proceed by presenting the related work in **Chapter 2**.

Chapter 3 describes our study on leveraging the user-posted feedback on Twitter to benefit the bug fixing process of Firefox and Chrome. We first explain the setup of the study, including the data collection and processing approach and the proposed mapping approach. We evaluate the proposed approach. Then, we use the approach to investigate the differences between issues posted on Twitter, and issues that are

not. Finally, we demonstrate the potential usefulness of monitoring and mining the user-reported feedback on Twitter to speed up the fixing process.

Chapter 4 reports the study conducted to investigate the social organizational structures in social coding platforms, and its potential impact on the code review process of the software projects. In this chapter, we detail the data collection and processing process, leading up to building the developers' collaboration networks. We define and characterize the team structures formed by the developers as they collaborate to review code changes. We report on the teams structures that are associated to the better and lowest performances of the code review process.

Chapter 5 discusses the study investigating the mechanisms used for the developers's informal communication (*i.e.,* the chatrooms), using Slack and Gitter as our subject systems. We collect feedback from the developers through a survey, to understand the reasons behind the use of Slack and Gitter, the perceived impact on the associated projects, and the quality determinants of the two chatrooms. We describe in this chapter the survey design, distribution, and analysis methods. We use the themes resulting from the survey analysis, to answer the investigated questions.

Chapter 6 describes the user study that examines the communication flow adopted by the software projects. We ask the developers through a survey to reflect about their experiences using a set of communication channels, with respect to a set of software development activities. We synthesize the findings from the survey to better inform the decisions of the project maintainers, when they setup up the flow of communication tools to be used. Finally, we study the associations between some combinations of channels (e.g., pull requests + Slack) and certain performance indicators of the subject projects (e.g., lowered bug fixing time).

Finally, we conclude and present the future work in **Chapter 7**.

Chapter 2

Literature Review

In this chapter, we give an overview of prior research related to the work conducted in this thesis. In particular, we focus on prior work that investigate the effect of the social and communication media on the following aspects: (1) disrupting aspects of the software development process, (2) inferring knowledge from the developers' social networks, and (3) supporting collaboration in software development.

2.1 Impact of social features on disrupting aspects of the software development process

In this section, we review the line of work that examined aspects of software development that were and are disrupted by the introduction of social features in the developers' tools. The social features mainly include the ability to set up a profile, to follow others users, to watch projects, to join in conversations with others, and to boost other users with a 'star' or an upvote.

2.1.1 Reputation building

In every developer social network, there is a group of influential developers who are considered as the "stars" of their respective communities. For example, the importance of the developers in Q&A websites is assessed based on their contributions in providing high quality answers to other developers through gamification, *i.e.,*, using elements of game design in non-game contexts [44, 45]. As such, developers are up-voted or down-voted, resulting in a building a reputation mirrored through a score and expertise badges assigned to the participants. In GitHub, every user has a number of follower and number of 'stars' that reflect the popularity of the user.

Bosu *et al.* [25] investigated the gamification system in StackOverflow which reflects the level of expertise of the users. The reputation score is built by providing answers that are accepted by the StackOverflow community. A higher score earns its owner trust and privileges within the StackOverflow community, and is a key motivator for future contributions. The authors analyze the StackOverflow data to understand the dynamics of reputation building in StackOverflow. Their results indicate that the following activities can help to build reputation quickly: answering questions promptly, being the first one to answer a question, answering questions related to tags with lower expertise density, being active during off peak hours, and contributing to diverse areas.

To clarify the role of gamification as a mechanims to sustain the desired user behaviour, Cavusoglu *et al.* [34] assessed whether the answering activity in Q&A websites is incentivized by earning badges on past contributions. Strong empirical evidence is found to support the value of the badges and the effectiveness of gamification in stimulating voluntary participation.

Blincoe *et al.* [24] studied the motivation behind following users on GitHub, and the influence of popular users on their followers. It is found that the popular users strongly influence their followers, in terms of guiding the followers to new projects. Overall, the number of followers and the popularity of a GitHub user is perceived as more important than their contributions, when influencing other developers.

The results reveal an emerging leadership style in GitHub, the number of followers and the popularity of a is perceived as more important than contribution, when influencing others.

2.1.2 Knowledge acquisition

Social and communication media, such as Q&A and micro-blogging websites, are commonly used by the developers for the purposes of learning and problem solving. The use of such media for knowledge acquisition allows to highlight the information needs of the developers, and leverage the uncovered information to better support the developers.

Acquiring knowledge on micro-blogging websites. Micro-blogging is a communication medium characterized by the exchange of short messages. Micro-blogging has appealed to millions of users due to its immediacy and portability. With the introduction of platforms, such as *Twitter*[1] and *Tumblr*[2], micro-blogging took off in 2006 and is still going strong. Researchers seeked to understand how software engineering has embraced the micro-blogging websites, and particularly *Twitter*. The software engineering community is a highly interactive population within *Twitter* [26], which

[1]https://twitter.com/
[2]https://www.tumblr.com/

allows different sub-communities to discuss specific topics related to software engineering. Based on a survey on developers who are active on Twitter, Singer *et al.* [121] find that Twitter can help developers stay up-to-date with the rapidly evolving development technologies, learning, and building professional relationships. The role of Twitter is to disseminate the up-to-date information related to software engineering, as confirmed by Sharma *et al.* [116]. As opposite to general micro-bloggers, micro-bloggers within the software engineering community form tighter communities where the relationships among the micro-bloggers are not highly reciprocal, as reported by Tian *et al.* [131].

Acquiring knowledge on Q&A websites. To uncover the programming concepts that are most confusing to the developers, Treude *et al.* [133] study the role of Q&A websites in the documentation landscape using StackOverflow as a case study. They find that the StackOverflow community answer review, conceptual and how-to/novice questions more frequently than other kinds of questions. The authors claim that understanding the interactions on Q&A websites can clarify the information needs of developers outside closed projects, and would enable recommendations on how to leverage knowledge from such websites. Wang *et al.* [140] extend the work by Treude *et al.* [133] using text mining to find the various kinds of topics asked by developers. They are able to derive many topics from the same question. As a result, the paper shows the distributions of questions and developers belonging to various topics. Allamanis *et al.* [7] analyze the questions asked in StackOverflow by topic, type and code. The authors aim to gain insight about the programming concepts that are most confusing to developers. They present a method to find out

what question types were mostly associated with particular programming constructs/identifiers. Such information can be used within IDEs to provide guidance with the frequently asked questions.

2.1.3 Productivity

The use of social features or networks while building software can constitute both an opportunity and a distraction. Therefore, many researchers wondered how the use of social networks affects the productivity of socially-active developers. Vasilescu *et al.* [136] investigate the association between the development process, reflected by code changes committed to GitHub, and crowdsourced knowledge, mirrored by StackOverflow activities. One of the challenges raised by Vasilescu *et al.* [136] is the ability to track a developer across different platforms. The authors use an email-based approach to map developers and associate their activity on StackOverflow and their productivity on GitHub. They find that active GitHub committers ask fewer questions and provide more answers than others. They also show that despite the work interruption caused by participating in StackOverflow discussions, StackOverflow activity rate correlates with the code changing activity in GitHub.

Ehrlich *et al.* [48] examine the communication network of a large software project. Communication networks, which link two developers who have previously communicated in a discussion thread, are popular in previous studies as they capture the knowledge sharing patterns among the developers [19][145][151]. Ehrlich *et al.* [48] found that developers performed better (*i.e.,* fixed more bugs) when they were central within a team's communication network. However, their performance worsened when they became central within the communication network of the entire project. This

revelation holds true even after controlling for the formal role of the developers, the individual differences in communication, the workload, and other factors that drive communication.

2.1.4 Evaluation of code contributions

As more open source projects welcome code contributions from new developers who are not part of the core team of the projects, the quality of the contributions needs to be carefully reviewed. Within a more transparent environment (*e.g.,* the social coding platforms), social signals(*e.g.,* posted comments, number of followers, or submitted issues) are collected about the contributors, and are suspected to impact the decision process of code contributions.

The evaluation of incoming contributions is a key aspect in the success of distributed software development and involves both social [47][92][139] and technical factors [80][94][109]. More particularly, prior work has found evidence that the social and technical impressions formed through the social coding platform GitHub are used by project maintainers to evaluate incoming contributions [134][135]. For instance, Tsay *et al.* [134] find that project maintainers are likely to consider both the technical quality of the contribution and the social connection of the contributor to the project maintainers, when evaluating a pull request. A similar finding by Gousios *et al.* [61] shows that the time it takes to accept and merge a pull request is influenced by the previous track record of a developer, among other technical factors, such as the code quality [63][134], the adherence to project conventions [63], and the inclusion of test code in the PR [63][134]). Identifying the best person to review a

pull request is also tackled in previous work. Yu *et al.* [149] propose a reviewer recommendation system based on comment networks of projects, as the review process is mostly embedded in the discussion section of the pull request.

> **Summary:** The increasing use of social features in the developers' work has disrupted different aspects in the software development landscape including reputation building, knowledge acquisition, productivity, and the process of evaluating code contributions.

The work conducted in this dissertation explores the impact of the social features on additional aspects that are unexplored by the previous studies. We conjecture that the use of social media and socially-enabled developer channels can also have an effect on the development activities, such as the bug fixing and code review activities. Specifically, we study the potential impact of monitoring the end-users' feedback from Twitter on the bug fixing process. Additionally, we examine the teams structures formed in GitHub, and study their associations on the merge time of code contributions.

2.2 Impact of social features on inferring knowledge from the developers' social networks

The use of the social and communication media has allowed for inferring knowledge from the developers' interactions. Latent social structures are formed by developers in projects as they contribute and communicate in their respective communities, so called as *socially-active communities*. The dyadic interactions among developers used to build the developers' networks can be defined in multiple ways and depend on the context. Communication networks, which link two developers who have previously

communicated in a discussion thread, are popular in previous studies as they capture the knowledge sharing patterns among the developers [19][145][151]. Collaboration networks, which associate a pair of developers if they have previously worked together, have also been extracted to recommend possible collaboration among developers based on compatibility [130]. We review the previous work that infers knowledge from developers' networks for three purposes: (1) extracting technical goals and expertise, (2) assessing the validity of bug reports, (3) predicting future bugs, and (4) identifying socially-prominent developers.

2.2.1 Technical goals and expertise

Social online platforms enable users to follow and track the online trace of a large number of users with no constraint on location or affiliation. The resulting transparency has a potential impact on the collaboration in complex knowledge-based activities. To explore the value of transparency for large scale collaboration platforms, Dabbish *et al.* [41] conduct a series of in-depth interviews with GitHub users. They conclude that many social inferences can be made based on the networked activity information in GitHub. Users can infer someone else's vision and technical goals when they edit a piece of code. Users are also able to guess the group of projects with a higher chance of thriving in the long term. Using these inferences, users in GitHub combine the knowledge acquired to form effective strategies for work coordination, technical skills advancement, and reputation management.

Venkataramani *et al.* [137] propose a model to capture the technical expertise of developers by mining their activities from GitHub in a target domain, in order to help a user posting a question identify the potential developers who can answer

it. They validate the proposed approach by building a recommendation system for StackOverflow.

2.2.2 Validity and triaging of bug reports

In most bug tracking systems, a big portion of the bug reports submitted are marked as invalid because they do no report a reproducible bug that can be fixed, or are duplicate of existing bug reports. Such bugs are usually time consuming as they need time to review. Therefore, researchers look into the possibility of predicting the validity of a bug report based on the history of the person who reported the bug. Zanetti *et al.* [151] propose a method to identify valid bug reports (*i.e.,* bugs that are not duplicate and that refer to an actual sofware issue). The bug reports are classified into valid and invalid based on who reported the bug. A bug is more likely to be valid if the bug reporter is more socially embedded in the collaboration network of the project. The authors find that the position of bug reporters in the collaboration network is a strong indicator of the quality of the bug reports. This study highlights the potential of using measures of social organization in collaborative software development.

Besides deciding on the validity of bug reports, the triaging of bug reports, *i.e.,* selecting the most capable(s) developer to address the bug, is also a time consuming task in the bigger projects. Badashian *et al.* [113] exploit a different source of evidence for the developers' expertise, namely their contributions to Q&A platforms, such as Stack Overflow. The study demonstrates that future bug triaging algorithms should consider other sources of expertise (*i.e.,* contributions to the socially-active communities), in addition to the traditional sources, such as the version control systems and

the bug trackers.

2.2.3 Predicting future bugs

The prediction of future bugs has a long history of research associated with it (*e.g.,* [95][73][97]). Most of the studies focused on the properties of the source code to make the prediction. However, other researchers claim that the human factors should not be omitted and therefore conduct studies to investigate the use of social information to predict bugs. Meneely *et al.* [93] propose to include human factors in building defect prediction models. The authors build developer networks by linking two developers who have modified the same source code file in the same period of time. The assumption behind building such networks is that "well-known" developers who have worked with many developers would be less error-prone. The "well-known" developers are identified using the networks' centrality measures. The authors find that including social information of the developers who modified a source code file is useful for failure prediction. The authors conclude by encouraging further studies about collaboration networks of developers, in order to proactively monitor and assess the organizational structure of the development teams [93].

Wolf *et al.* [145] investigate the prediction of build failures using the communication networks of developers. The authors construct the communication networks of each build of IBM's Jazz project. They compute network measures that mirror the communication structures of the project. The authors find that the combination of network measures can be used to build a prediction model of build failures. Given that communication among developers reflects knowledge sharing among developers, the authors propose to integrate real time feedback to indicate to developers whether

their communication patterns are risky and can lead to build failures.

2.2.4 Identifying socially-prominent developers

The follow networks of developers capture the follow behaviours among developers in social coding platforms, such as GitHub. Schall *et al.* [114] examine the follow network of developers on GitHub to recommend who to follow. The purpose is to help developers build a reputation and a strong network among their peers [114].

Yu *et al.* [150] mine the follow networks, and identify the behaviour patterns of developers from the networks (e.g., star, group, or hub shaped). Yu *et al.* [150]) further claim that the identified behavior patterns can inform the design of assistive tools for developers, such as recommendation systems.

Summary: The latent structures formed by the developers as they collaborate and communicate are a useful source of information for various aspects, such as inferring the expertise and technical goals, assessing the validity and triaging bug reports, and predicting future bugs.

The previous studies have successfully inferred knowledge from different types of developers' networks. However, the collaboration networks in social coding platforms capture a different layer of interactions among the developers, that is hardly investigated. Specifically, in GitHub, the collaboration happens when one developer reviews the pull request (*i.e.,* code contribution) made by others. As part of the research conducted in this thesis, we build the networks formed by the developers in GitHub and compute a set of network metrics, from which we infer and characterize the teams structures formed by the developers. The inferred team structures are found to partly explain the performance of the team in processing the incoming code

contributions.

2.3 Impact of social features on supporting collaboration in software development

Currently, developers use a complex constellation of tools and channels to support the development process. Therefore, it is important to understand the roles, challenges, and impacts of using these tools. In this section, we survey the previous work investigating the importance of the social and communication media, along three dimensions: (1) project awareness, (2) learning and problem solving, and (3) informal communication.

2.3.1 Project awareness

Project awareness is essential for coordinating the task activities both in collocated and distributed development teams. Project awareness implies maitaining knowledge about different aspects, such as the team (*e.g.,* who is doing what?), the tasks (*e.g.,* what are the ongoing tasks?), and the presence (*e.g.,* who is available for discussion?). An early study by Gutwin *et al.* [66] finds that the developers are able to maintain awareness of one another, and of the entire team activities primarily through text-based communication (*i.e.,* mailing lists and chat). Developers are able to maintain awareness by simply overhearing others, and seeing who is talking about what. As a result, the participants in the discussions are able to identify the *experts in an area*, depending on who responds to an initiated discussion.

Rigby *et al.* [107] examine the value of mailing lists to achieve *broadcast-based code review*. The study reveals that large open source projects are able to effectively

broadcast code review requests using mailing lists. To avoid overwhelming the developers with information, different stategies are implemented, such as the use of multiple speciliazed lists with fewer developers, and making explicit requests, *i.e.,* messages sent directly to potential reviewers as well as the entire mailing list. However, a more recent study by Guzzi *et al.* [68] reports that mailing lists are no longer the hub of projects' communication, and there is a shift of interest towards social coding websites such as GitHub. Storey *et al.* [125] recently confirm, through a large scale survey, that the social coding platforms are currently the most used medium to 'watch' project activities.

2.3.2 Learning and problem solving

Learning and *problem solving* are both essential parts of the work of developers. Developer Q&A websites (*e.g.,* Stackoverflow) are used to a great degree in order to support the developers in solving problems and learning the best practices. Mamykina *et al.* [90] found that the tight engagement with the community in Stack Overflow is critical to the success of knowledge sharing within the SO community. Parnin *et al.* [100] report on the steady growth of Stack Overflow as an invaluable *documentation resource* (e.g., APIs). In terms of impact, Stack Overflow participation rate is found by Vasilescu *et al.* [136] to correlate with the code changing activity in GitHub, thus demonstrating the value of problem solving and learning on the developers' Q&A websites.

Twitter is another channel adopted by the developers to *stay informed* on the latest trends and technologies. Singer *et al.* [121] report that Twitter is unique because it allows the developers to keep up with the fast-paced development landscape.

Twitter is found to help developers extend their software knowledge through “casual" learning (*i.e.,* in a serendipitous manner), by following experts, and participating in conversations. Another medium that supports learning and problem solving is the developers' chatrooms (*e.g.,* Slack). Lin *et al.* [86] report from the results of a survey that the most reported use of Slack for personal benefits is discovery and information aggregation.

2.3.3 Informal communication

Although formal communication is essential for tasks, such as triaging bugs or reviewing a code change, informal communication, so called as “corridor talk”, is also vital to maintain awareness, build relationships, and discuss metawork [75]. Informal communication is particularly important for distributed teams, as it is hard to recreate without face-to-face interactions. An inneficient informal communication can lead to misalignement of expections, and eventually to rework [75]. The introduction of socially-enabled tools that attempt to mimick face-to-face interactions was important in support the informal communicaiton among developers.

In a large scale survey ran by Storey *et al.* [125], developers are asked about the channels they use and the activities supported by the channels. The feedback from the survey reveals a reliance on communication channels that support social features, to support several informal communication activities, such as keeping up-to-date, connecting with developers, getting and giving feedback, displaying skills, assessing others, and coordinating with others.

Lin *et al.* [86] investigate how a specific channel (*i.e.,* the chatroom Slack) supports the informal communication of the developers. Collaboration on Slack is found to

support informal communication at two levels: 1) at the community level, participants stay up-to-date with specific frameworks or communities, and 2) at the team level, participants collaborate through remote meetings, file and code sharing, and note taking.

Nurturing relationships with peers is another important aspect fulfilled through informal communication. Singer *et al.* [121] find that Twitter encourages the building of larger communities, but also helps foster trust-based relationships between teammates. Additionally, it is not uncommon for collaborations to be started between random strangers through the micro-blogging platform.

A non-negligeable aspect of the software development process is metawork, *i.e.,* the required preparation before the actual tasks can be understaken. Giuffrida *et al.* [56] reports on the importance of what they refer to as 'social software', specifically chatrooms and forums, in the initial phases of projects when decisions need to be taken, and social protocals need to be negotiated. In later phases of the project, 'social software' becomes a dispatcher to other channels, and the main channel for the discussion of metawork, a key aspect aspect to maintaining successful collaboration.

Summary: The social and communication media plays an important role in supporting diverse aspects of the software development process, such as project awareness, learning and problem solving, and informal communication.

The prior work helped to form a body of knowledge around the use of the communication channels by the developers. Nonetheless, the work conducted as part of this thesis is the first to empirically study the motivation behind the choice for a

particular set of communication channels. Our study is important to help developers take more informed decisions when choosing communication channels for their unique project setting. We also derive recommendations of which set of communication channels works best for specific development activities (*e.g.,* bug fixing or release announcements).

Chapter 3

An approach for Leveraging Crowd-sourced Feedback to Improve the Bug Fixing Process

In this chapter, we describe the research conducted to demonstrate the value of leveraging the crowd-sourced feedback from social media, to improve the bug fixing process. We first present the motivation behind the study, and describe the case study design. We then present the study findings. Finally, we discuss the threats the validity, and provide a summary to the chapter.

3.1 Problem and Motivation

Technical users (*e.g.,* developers) are used to filing a bug report into an issue tracking system, when they encounter a software bug. When reporting a bug, technical users are required to adhere to strict guidelines for bug reporting; otherwise, a bug report may be rejected. For example, the bug reporter needs to provide a detailed description of the bug and the concrete steps to reproduce the bug. Hence, a bug report is well structured. Structured bug reports can help the development team to accelerate bug triaging and fixing. However, non-technical end-users can be intimidated by the

overhead of filing a well structured bug report. Sometimes, non-technical end-users just want to voice their opinions on more familiar channels where they can draw some attention to their problems. In particular, non-technical end-users may prefer to use social networks.

Social networks are good sources to collect timely and crowd-sourced reactions from people on hot and ongoing events. It is very important to promptly collect the feedback from end-users, especially after a new software release. Although a new release is expected to fix bugs and add new features, new bugs can also be introduced inadvertently. For example, a flood of complaining tweets forced Apple to immediately pull the release of iOS 8.0.01 [143]. For a software system, end-users may switch to its competitors, if the development team is not responsive to the end-users' feedback and the reported bugs remain unfixed for a long period of time. To retain the loyalty of the end-users, it is of great interest to study how social networks can be used to improve the bug fixing process.

Another value of collecting feedback from a large user base is that different users experience very diverse real-life scenarios and have varied configurations. Therefore, the development team can better know how their software product runs in every possible setting. We are interested to investigate how the feedback posted by end-users on social networks, and particularly the micro-blogging platform Twitter, could be useful to the bug discovery and fixing process. We specifically focus on Twitter in this study because of how it is commonly used by its users to express opinions, both positive and negative. As such, Twitter is a suitable social networking platform to collect the bugs reported by the users.

Twitter is the largest micro-blogging platform, where end-users post short messages (*i.e.,* at most 140 characters at the time of the study, later increased to 280 characters as of the time of this writing), called tweets, to express opinions or report events. Over the last few years, Twitter has gained a rapid popularity and adoption. As of September 2016, Twitter has 313 million monthly active users. Tweets are free text without a rigorous structure that report instant feedback from end-users. Tweets have been studied heavily using sentiment analysis [82, 99, 57] and opinion mining [99, 88, 98]. These studies aim to help product owners answer questions, such as: *how do people feel about our product?* or *what would people want us to add to the product?* We focus on the feedback on software products only.

Particularly, we perform an empirical study to understand the value of using tweets in the bug fixing process. We choose to study Chrome and Firefox, the two most popular web browsers. The two systems are rapidly released (*i.e.,* approximately every six weeks), thus it is time sensitive to collect feedback promptly for these two systems so that more bugs reported in one release could be resolved in the subsequent release.

Specifically, we investigate the following four research questions:

RQ 3.1 How accurately can our approach map tweets and bug reports?

To the best of our knowledge, there is no well-established approach to map tweets and bug reports. In this paper, we propose an automated approach to map tweets and bug reports based on text similarity. The results show that our approach achieves a precision between 76% and 84% when considering different pre-processing steps, and outperforms the baseline approach involving no pre-processing steps (*i.e.,* 47% to 52%) with a large margin.

RQ 3.2 What web browser issues receive more feedback from end-users through Twitter?

We refer to a bug report as a tweeted bug report, if at least one tweet is successfully mapped to the bug report. Otherwise, we call it a non-tweeted bug report. We compare the topics between the tweeted and non-tweeted bug reports. We find that tweeted bug reports are more likely to contain issues related to performance, security, and audio/video issues in both browsers.

RQ 3.3 Does the development team handle a bug report differently if the problem is mentioned on Twitter?

We compare the tweeted and non-tweeted bug reports in terms of the assigned severity and three fine-grained intervals (*i.e.,* delay before response, delay before assignment, and duration of bug fixing). Our experiments show that end-users' tweets do not appear to be currently leveraged in the bug fixing process for both systems.

RQ 3.4 Can we use tweets to achieve an early discovery of bugs?

Given the strength of tweets in providing timely feedback from a large user base who have very diverse settings (*e.g.,* machine type, operating system, and machine configuration), we are interested to examine if using tweets can lead to an early discovery of bugs. Indeed, we find that 33.4% and 33.5% of the bugs could have been reported earlier to developers averagely by 8.2 days and 7.6 days in Firefox and Chrome, respectively. With a well designed tool for summarizing tweets, the development team could discover bugs earlier, and possibly focus their efforts on the bugs that affect a large user base.

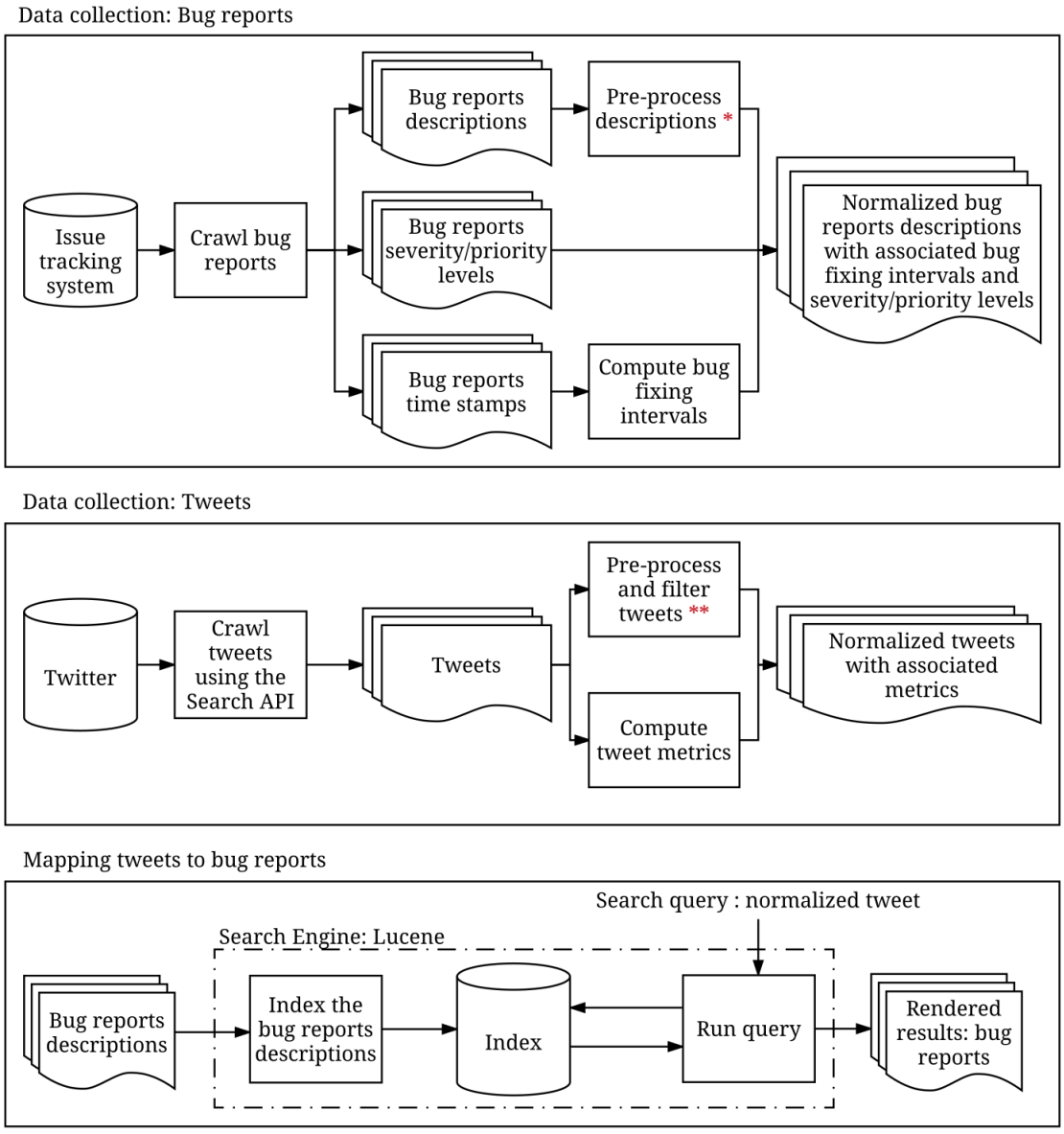


Figure 3.1: Overview of the case study design

3.2 Study Design

In this section, we describe the overview of our case study design. The overall design of our study is depicted in Figure 3.1.

First, we download the bug reports from issue tracking systems (*e.g.,* Bugzilla).

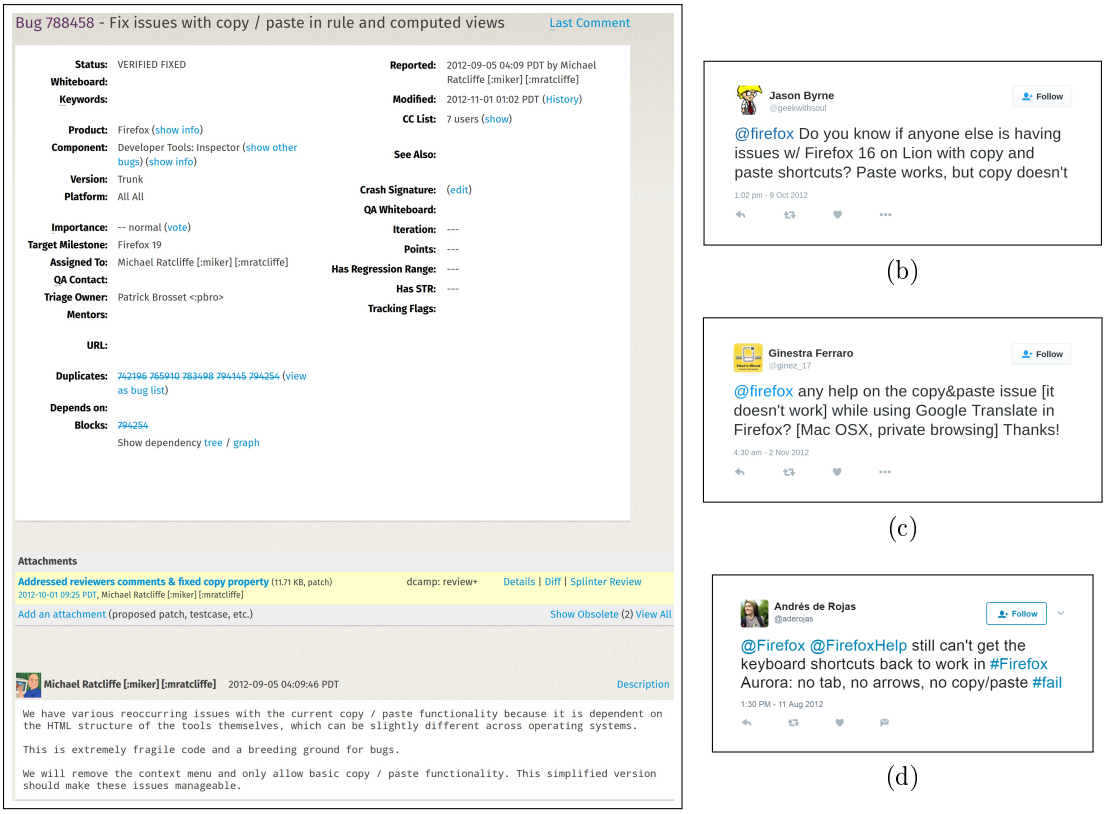


Figure 3.2: An example of a Firefox bug report and three tweets possibly related to the bug

Then, we crawl tweets from Twitter. We pre-process both the bug reports and the tweets, by cleaning and normalizing the collected data. We develop an automated tool to collect, pre-process, and select the relevant feedback from users on Twitter.

Second, we map the bug reports and the tweets using a text-similarity off-the-shelf search engine. Figure 3.2 shows an example of a possible mapping between a bug report (Figure 3.2-a) and a tweet (Figure 3.2-c). Our approach is evaluated by three non-author evaluators (*i.e.*, RQ 3.1).

Third, we perform an empirical study to understand if the bug reports that are

successfully mapped to users' tweets are different from the bug reports without associated tweets, in terms of the bug fixing intervals, the severity/priority levels, and the types of reported bugs. As such, we compare the bug fixing process between the tweeted and non-tweeted bug reports in RQ 3.2 and RQ 3.3. We further get in touch with developers from both Firefox and Chrome to verify the findings of our quantitative study.

Finally, we investigate whether acknowledging the feedback from end-users on Twitter could help the development team discover bugs at an earlier stage (*i.e.,* RQ 3.4).

3.2.1 Bug reports

In this section, we first present the subject systems, describe the background of the bug fixing process, then explain our collection and pre-processing of the bug reports.

Subject systems

A new release of a software product may not be a trending event on a social network, such as Twitter. Therefore, a new release may not receive many tweets, unless the product has a large user base. As such, we choose to study the two most widely[1] used browsers (*i.e.,* Chrome and Firefox) that are released on a regular basis (*i.e.,* approximately every six weeks). In particular, Firefox and Chrome have adopted the rapid release cycle since June 2011 (Version 5.0) and October 2011 (Version 15.0), respectively. The rapid release cycle allows the collection of more frequent feedback (*i.e.,* The developers are likely to hear from the end-users every six weeks).

Firefox is an open source browser developed by Mozilla Foundation and opens

[1]According to W3Schools' browser statistics in March 2016, Chrome ranks the first with 69.9% of the usage share of browsers, followed by Firefox that has approximately 17.8% of worldwide usage share of the browsers. URL: http://www.w3schools.com/browsers/browsers_stats.asp

its issue tracking system to the public. However, Chrome is proprietary software of Google, and its issue tracking system is unaccessible by the public. As the bug fixing process is recorded in the issue tracking system, we turn to the issue tracking system of Chromium which is the open-source version of Chrome. Chromium and Chrome largely share the same code and features, except for some minor differences in features and licenses. We further observe that Chromium has only developmental releases while all official stable releases belong to Chrome. Therefore, issue reports of Chrome can be easily distinguished from those of Chromium using the release version. Specifically, we select issue reports that are associated to the stable releases of Chrome only.

In total, we study 37 consecutive versions of Firefox (from Version 5.0 to Version 41.0), and 34 stable releases of Chrome (from Version 15.0 to Version 48.0). The dates of the Firefox releases can be obtained from the *history of Firefox*[2] Wikipedia page. The dates of the Chrome releases can be found on the *Google Chrome release history*[3] Wikipedia page.

Bug fixing process

Bug reports record the entire bug fixing process. The typical life cycle of the bug fixing process starts from the moment a bug is reported and ends when it is closed. A detailed life cycle of Firefox bug reports is described in the work by [142]. Details of the life cycle of Chrome bug reports can be found on the *Bug Life Cycle and Reporting Guidelines* web page[4]. Although slightly different terminologies are used in different issue tracking systems, the life cycle of a bug report usually goes through the phases shown in Figure 3.3. Details of each state are described as follows:

[2]https://en.wikipedia.org/wiki/History_of_Firefox
[3]https://en.wikipedia.org/wiki/Google_Chrome_release_history
[4]https://www.chromium.org/for-testers/bug-reporting-guidelines

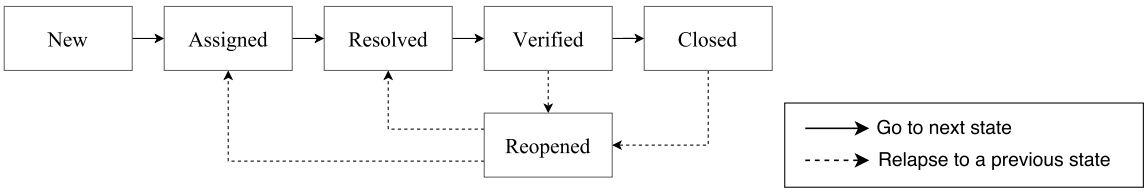


Figure 3.3: Life cycle of a bug report

1) `NEW`: A bug report status is initially set to `NEW` when it is first created. For each new bug report, the development team initiates bug triaging to find the most appropriate developer to work on the bug. Bug triaging may require several iterations until the most appropriate developer is identified.

2) `ASSIGNED`: After triaging, a developer is assigned to work on resolving the bug. The status is changed to `ASSIGNED`. Developers start to inspect and fix the bug.

3) `RESOLVED`: Once the bug is fixed, the status is set to `RESOLVED`. Usually, developers will submit a patch and related test cases. At this stage, the testing phase starts.

4) `VERIFIED`: If the submitted bug fix passes testing, the tester marks the bug report as `VERIFIED`. The bug fix is ready for release.

5) `CLOSED`: The bug is marked as `CLOSED`, which indicates the end of the fixing process. However, a bug report can be reopened if the corresponding fix is unsatisfactory. In this case, the status is set to `REOPEN` and bug triaging is initiated again.

In order to assess the impact that tweets may have on the bug fixing process, we compute fine-grained intervals during the entire bug fixing process. Similar to the

work by [153], we identify and compute four fine-grained intervals: delay before reported (DBR_{eported}), delay before response (DBR), delay before assignment (DBA), and duration of bug fixing (DBF). To compute these intervals, we mine timestamps of all changes made on the status of every bug report. Specifically, we compute each interval in the following way (also shown in Figure 3.4).

1) Delay Before Reported (DBR_{eported}) measures how long it takes the development team to acknowledge a bug after a new release. It is computed as the interval between the date of a version release (*i.e.,* T_{release}) and the date at which a bug report is filed (*i.e.,* T_{reported}). The creation date is recorded in the bug report. However, it is not clear when the bug was introduced. Considering that both Firefox and Chrome are set to auto-update by default on end-users' machines, we assume each reported bug was introduced by the most recent release, or is a dormant bug revealed by the most recent release. In both cases, the bug is only a nuisance to the users after it is triggered by the most recent release. As aforementioned, we extract the date of the most recent release from release history of these two browsers. Then, we compute the interval DBR_{eported} using the following equation:

$$DBR_{\text{eported}} = T_{\text{reported}} - T_{\text{release}} \tag{3.1}$$

2) Delay Before Response (DBR) reflects the delay of the development team on taking initial actions after a bug is reported. It is calculated as the interval between the filing of a report (*i.e.,* T_{reported}) and its first response from the development team. The first response can be captured by several types of initial

actions taken by the development team, which are the addition of a developer to the Carbon Copy (*i.e.,* CC) list of the bug (*i.e.,* the list of developers who receive emails about any updates to the bug report), the change in the status of the bug, or the posting of the first comment. We mine the timestamp of these actions and choose the timestamp of the earliest action as the time of the first response (*i.e.,* $T_{\text{responded}}$). Then, we calculate the interval DBR using the following equation:

$$DBR = T_{\text{responded}} - T_{\text{reported}} \tag{3.2}$$

3) Delay Before Assignment (DBA) captures the time spent in triaging during the bug fixing process. It is computed as the interval between the first response to a bug and its assignment to a developer. As bug triaging may require several iterations, we choose the most recent assignment as the final assignment. We mine the timestamp of the most recent assignment as T_{assigned}. Then, we compute the interval DBA using the following equation:

$$DBA = T_{\text{assigned}} - T_{\text{reported}} \tag{3.3}$$

4) Duration of Bug Fixing (DBF) describes the duration spent on inspecting the bug and performing the corresponding code changes by the assigned developer. It is calculated as the interval between the assignment of the bug and the resolution of the bug (T_{resolved}) when the status of the bug is changed to `RESOLVED`. DBF does not capture multiple resolve actions, but reflects the duration of the successful and final resolution. We calculate the interval DBF using the

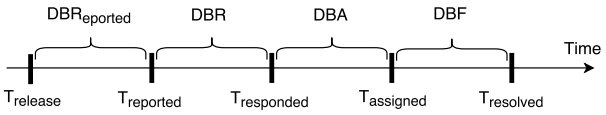


Figure 3.4: Intervals of the bug fixing process

following equation:

$$DBF = T_{\text{resolved}} - T_{\text{assigned}} \tag{3.4}$$

Bug reports collection

Bug reports are stored in issue tracking systems. Firefox uses Bugzilla[5] as its issue tracking system. From Firefox Bugzilla, we crawl 14,489 fixed bug reports associated to the releases under study. The median number of bug reports for a single release is 262 (see Table A.1). Chrome uses the Chromium tracker[6] as its issue tracking system. Although the Chromium tracker contains bug reports for both Chrome and Chromium, they can be distinguished from each other using the *ReleaseBlock* field. The *ReleaseBlock* field can be either *stable* or *developmental*. As aforementioned, bug reports for stable releases belong to Chrome. Therefore, we download all bug reports with the *ReleaseBlock* field set to *stable*. In total, we crawl 15,771 fixed bug reports associated with the stable Chrome releases under study. The median number of Chrome bug reports for a single release is 477.

Bug reports from both Bugzilla and Chromium tracker share a similar structure with a different flavour. For instance, both issue tracking systems record the description of the problem and the severity or priority of the problem. A full list of historical

[5]https://www.bugzilla.org/
[6]https://bugs.chromium.org/p/chromium/issues/list

activities for each bug report is recorded separately from its main page in Bugzilla. In the Chromium tracker, a new comment is posted on the main page of the bug report when the status is updated. We extract both descriptive information and the history information from all the bug reports for the two subject systems.

Bug reports pre-processing

A manual investigation of the bug reports reveals that some bug reports contain logs that are automatically generated for debugging purposes (*i.e.,* automatically generated logs). For example, as shown in Figure 3.5, the text in the automatically generated logs contains mainly timestamps and other technical details describing system events. Considering the limitation on the number of characters in a tweet, we assume that it is unlikely the end-users would copy and paste the automatically generated logs to Twitter. Therefore, we exclude the lines identified as automatically generated logs from the bug report description prior to mapping the bug reports to the tweets. [10] propose a sophisticated approach to classify the content of mailing lists at the line level into text, junk, code, patch, and stack trace. For our work, we opt for a simpler approach tailored to the bug reports, where we classify the lines into 2 classes: text and automatically generated logs.

To identify the lines containing automatically generated logs, we parse the bug reports line by line. We observe that multiple instances of automatically generated logs start with a timestamp of the form `hh:mm:ss`. We use the following regular expression to flag such lines: `^\d{2}[:]+\d{2}[:]+\d{2}[ \t]`. In other instances, we identify the automatically generated logs by looking for lines starting with a list of words such as `stack trace`, `slave`, `test-pass`, or `test-unexpected-fail`. We use the following regular expression to identify such patterns `(^stack\strace)|((^slave))`

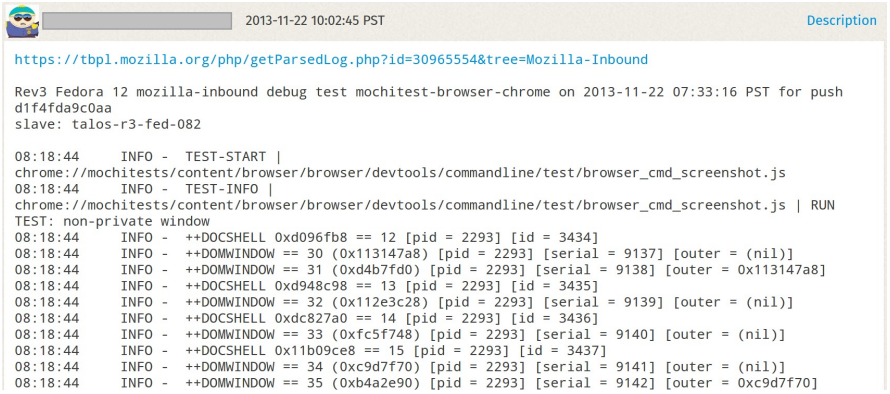


Figure 3.5: Example of automatically generated logs in the description of a bug report.

`|((^test-pass))|((^test-unexpected-fail))`.

Normalization is a process that converts a list of words into a more uniform sequence. The purpose of normalization is to prepare the text for further processing. We perform the normalization process using the following three steps: *1)* removing the non-English words using the Moby words list[7] (the largest list of English words in the world); *2)* removing the English stop words[8] (*e.g.,* about, such, or too); and *3)* reducing all words to their roots, *i.e.,* stemming using the Porter stemmer [105]. We perform the normalization on all bug reports.

3.2.2 Tweets

Twitter[9] is the largest micro-blogging service for social networking. End-users can freely post messages (*i.e., tweets*), with a 140-character limitation for each message.

We assume that the tweets posted around the time when a new version is released

[7] http://icon.shef.ac.uk/Moby/mwords.html
[8] https://www.dropbox.com/s/6s72idyr2s20ho8/stop_words.txt?dl=0
[9] https://twitter.com/

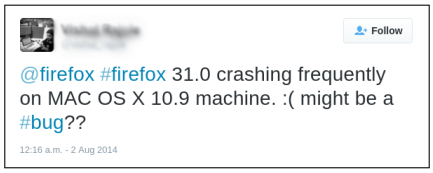


Figure 3.6: Example of a tweet addressed to Firefox using the "@" symbol

are mostly about the new release. This is because both Firefox and Chrome are set to auto-update by default on end-users' machines. As such, most end-users are very likely to run the new version of each browser soon after the release.

Tweets crawling

It has been reported that the velocity of tweet posting is around 6,000 tweets per second[10]. Prior to using tweets in the bug fixing process, the first challenge is to identify the subset of relevant tweets from a flood of tweets from Twitter.

1) Reduce the search space. Millions of tweets are posted on Twitter everyday. We aim to only retrieve the tweets posted by end-users to report bugs encountered in the subject systems. Therefore, we consider two features of Twitter to reduce the search space of tweets related to Chrome and Firefox. One feature of Twitter is the hashtag (*i.e.,* "#"), which is a type of label or metadata tag that is used to group and for end-users to find messages related to a specific topic or theme. End-users can create and use hashtags by placing the hash character "#" in the text of the tweet. Another feature of Twitter is the usage of the "@" symbol. A tweet can be addressed to a specific end-user using the "@" symbol followed by the username of the target end-user (*e.g.,* `@firefox`). Figure 3.6 shows an example of a tweet addressed to

[10]http://www.internetlivestats.com/twitter-statistics

Firefox using both the "@" symbol and the hashtag "#".

Both Firefox and Chrome maintain their official Twitter accounts. The Firefox official account has 2.86 million followers and over 27,000 tweets, as of March 2016. The Firefox account is used to communicate with the Firefox end-users and announce updates about the browser. The Chrome official account has 5.9 million followers and over 1,000 tweets. The Chrome account is mostly used for the promotion of the new features of the browser.

In this study, we identify the tweets addressed to each official account using the "@" symbol. We do not use the hashtag ("#") to identify tweets about Firefox or Chrome. Although tweets containing the hashtags "#Chrome" or "#Firefox" might relate to issues about Chrome and Firefox, using these hashtags result in a large amount of false positives. Two of the authors manually investigate a statistically significant sample of tweets (361 tweets) containing the hashtags "#Chrome" or "#Firefox", to have an estimation of the number of false positives. We find that 60.8% of the tweets are false positives. For example, a false positive is the following: "*Resolved the DCC bug for #Firefox. Still working on #Chrome. Common code, different behaviour. #Javascript*". Another example is as follows: "*Google reports a "high severity" bug in IE/Edge, no patch available #chrome #firefox #opera*". Within the remaining 39.2% of the tweets that are true positives, less than 10% are bug-reporting tweets. It is hard to automatically identify whether the problem included in the tweet is about one browser or the other just by parsing the words in the tweet. Therefore, it is challenging to automatically distinguish false positives from true positives. In this paper, we want to focus on studying the possible use of the end-users' feedback (*i.e.,* the tweets) on the bug fixing process. Therefore, we map tweets to the bug reports.

To improve the accuracy of the mapping, we leave the overhead of filtering tweets with the hashtags `"#Chrome"` or `"#Firefox"` to a future study. On the other hand, the `"@"` symbol ensures that a tweet is directly addressed to the official account of a specific browser.

2) Perform the search. Twitter provides a search API that allows queries against the indices of recent or popular tweets, such as the `"until"` operator that is used to retrieve tweets posted before a given date. An example of a query used to retrieve the tweets associated with Firefox Version 40 is "lang:*en* to:*@firefox* since:d_{start} until:d_{end}", where d_{start} is the release date of Version 40, and d_{end} is the day right before the next release.

Figure 3.7 depicts an example of a Twitter conversation. The main tweet, shown in a bigger font, has 5 replies and is the first tweet in the conversation. For each tweet, we further retrieve the conversation in which the tweet appears. A tweet can be either an original tweet or a reply to another tweet. For each tweet, we extract the ancestors and the replies, if any. We categorize the tweets by releases, based on the dates they were posted. The crawled tweets span over four years. The number of tweets posted after each release of Firefox and Chrome are presented in Table A.1.

Tweets pre-processing

The tweets are subjects to the use of abbreviations and to the occurrence of misspellings. It is common that many words are misspelled or abbreviated. To address this issue, we first handle the anomalies (*i.e.,* abbreviations and misspellings). Moreover, the tweets retrieved are not all relevant to bugs, as some of them merely express opinions or ask general questions. An example of an irrelevant tweet is:

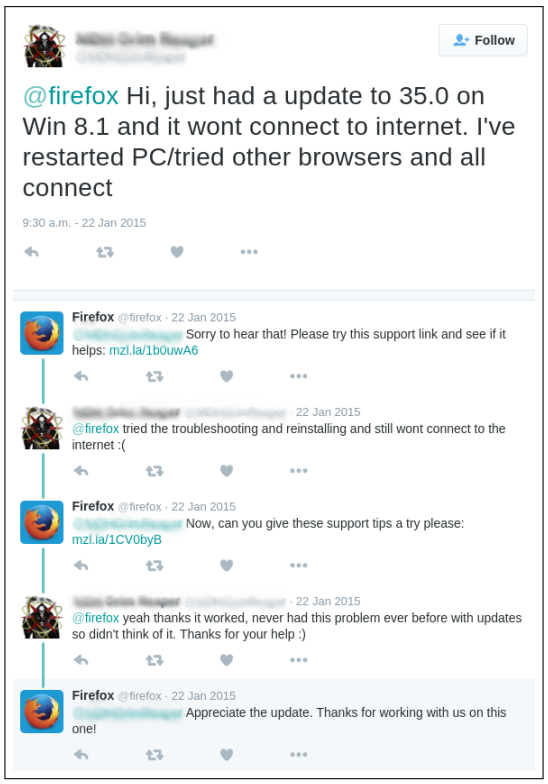


Figure 3.7: Example of a conversation on Twitter between an end-user and the Firefox official account (Usernames are blurred to preserve privacy)

"*@firefox can't believe that the worlds best browser is almost 10!*". Therefore, we further filter irrelevant tweets.

The two steps are depicted in Figure 3.8, as well as the normalization steps of the words in the tweets. We provide the details of each step in the subsequent paragraphs.

1) Correct anomalies. Anomalies are the misspellings, abbreviations and non-standard orthography that exist in the tweets. To correct the anomalies, we collect a list of common abbreviations used by users on Twitter and replace them with the actual correct

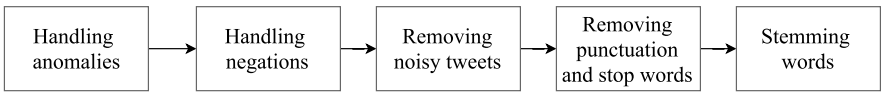


Figure 3.8: Tweet processing steps

Table 3.1: Examples of common twitter abbreviations or acronyms

Twitter abbreviations and accronyms	Meaning
ICYMI	In case you missed it
MTF	More to follow
Twaffic	Twitter traffic
TY	Thank you
TT	Trending topic
RTQ	Read the question
...	

words. The list of abbreviations is gathered from various online sources[11,12,13,14,15]. Examples of common non-standard orthography in Twitter are shown in Table 3.1. Furthermore, we examine a statistically representative sample of tweets to identify some of the most common misspellings in the tweets and correct them before further processing. The sample has 361 tweets that are randomly selected from the entire population, using a 95% confidence level with a 5% interval[16]. Few examples of the most common Twitter misspellings are shown in Table 3.2.

2) Filter out irrelevant tweets. The existence of bug-related terms, such as “lag" or “crash", in a tweet possibly indicates that the tweet reports a bug. We build a

[11]http://www.socialmediatoday.com/content/top-twitter-abbreviations-you-need-know
[12]http://www.webopedia.com/quick_ref/Twitter_Dictionary_Guide.asp
[13]http://www.noslang.com/twitterslang.php,searchcrm.techtarget.com/definition/Twitter-chat-and-text-messaging-abbreviations
[14]http://marketing.wtwhmedia.com/30-must-know-twitter-abbreviations-and-acronyms/
[15]https://digiphile.wordpress.com/2009/06/11/top-50-twitter-acronyms-abbreviations-and-initialisms/
[16]www.surveysystem.com/sscalc.htm

Table 3.2: Examples of common twitter misspellings

Twitter common misspellings	Correct spellings
dont	do not
da	the
B	be
youre	you are
its	it is
cant	cannot
...	

customized dictionary of bug-related terms by manually reviewing the aforementioned statistically representative sample of tweets. Examples of bug-related terms that we obtain are *issue*, *bug*, *problem*, *fail*, *freeze*, and *glitch*. A similar approach is adopted by [138] to define a dictionary of bug-related terms, when they identify user reviews of mobile apps that report bugs.

Nevertheless, tweets containing negated bug-related terms (*e.g.,* "no lag" or "does not crash"), such as the one shown in Figure 3.9, can be misleading when identifying the bug-reporting tweets. [138] address a similar challenge in user reviews by identifying the negated terms using the Stanford parser [122] and discarding them. We adopt the same approach and remove the bug-related terms that are negated. The tweet shown in Figure 3.9 thus becomes "*@firefox 3.6.1 , Its Super fast, And I liked The Mac os x Skin.*". Therefore, it is discarded from the set of relevant tweets.

To detect the instances of negated bug-related terms, we use Part-Of-Speech tagging from the Stanford parser [122]. We use one of the available interfaces of the Stanford parser API[17]. For instance, the sentence "the browser is not laggy" is processed into the following parse tree. The meanings of the parser tags (*e.g.,* NP = Noun Phrase) can be found in [20].

[17] http://projects.csail.mit.edu/spatial/Stanford_Parser

```
(ROOT
  (S
    (NP (DT the) (NN browser))
    (VP (VBZ is) (RB not)
      (ADJP (JJ laggy))
    )
  )
)
```

The parser also returns a list of dependencies (*i.e.,* grammatical relationships among words) as follows. The meanings of the possible dependencies (*e.g.,* neg = negation modifier) are available in the Stanford parser user manual[18]. The number following each word in the list of dependencies (*e.g.,* laggy-**5**) refers to the sequence of the word in the phrase.

```
root ( ROOT-0 , laggy-5 )
det ( browser-2 , The-1 )
nsubj ( laggy-5 , browser-2 )
cop ( laggy-5 , is-3 )
neg ( laggy-5 , not-4 )
```

We then use the "neg" label (*i.e.,* the last item in the list of dependencies above) to identify the negated bug-related term "laggy".

Finally, we obtain 6,044 tweets that are identified as reporting Firefox bugs (out of 54,293 tweets that are originally retrieved) and 6,158 bug-reporting tweets for Chrome (out of 38,349 tweets that are originally retrieved).

[18]https://nlp.stanford.edu/software/dependencies_manual.pdf

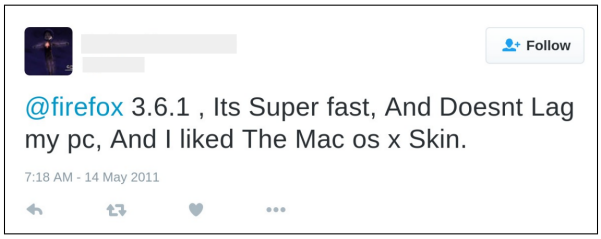


Figure 3.9: Example of a tweet containing a negated bug-related term

3) Normalize tweets. Similar to bug reports, we normalize the text of the tweets. In particular, we remove punctuation (*e.g.,* {, . !}) and special symbols (*e.g.,* {& # $}) from the collection of tweets. Then we remove the non-English words using the Moby words list to retain only the words that are correctly spelled and we exclude all stop words. Finally, we stem the words using the Porter stemmer.

Tweets-related metrics

A bug-reporting tweet that receives higher attention from other users might indicate how critical and popular the problem is among the users. A tweet receives higher attention if it has been liked and shared by many other users. We propose to use the following five metrics to assess the popularity of tweets:

1) # **Favorites** measures the agreement and interest of other users in the content of a tweet. It is defined as the number of times that a tweet is marked as a favorite (*i.e.,* liked by a user);

2) # **Retweets** is a means to measure how much the information in the tweet has been diffused among Twitter users. It is defined as the number of times that a tweet has been retweeted (*i.e.,* re-posted by other users for their followers to see);

3) **Has replies?** is a means to assess the relevance of a tweet through the acknowledgment by the software provider. It is defined as whether the tweet has received a reply from the official Twitter account of the browser. Such replies are usually used to answer a question or acknowledge a problem faced by a user (An example is shown in Figure 3.10);

4) **Duration** reflects the relevance of the tweet over time among the Twitter users. It is computed as the total duration of the conversation that a tweet is part of (*i.e.,* the list of replies to the tweet). An example of a Twitter conversation is shown in Figure 3.7;

5) **Interaction interval** measures the engagement of Twitter users with the tweet, through the velocity of their interactions. It is defined by computing how fast users interact with each others in a conversation. It is possible to have sub-threads within a Twitter conversation. We do not solve the non-trivial threading problem in this case (*i.e.,* identifying the sub-threads within a Twitter conversation). The interaction interval is measured by *a)* identifying all the tweets that are marked as a reply to the initial tweet X; *b)* computing the time difference between each two consecutive tweets in the conversation; and *c)* reporting the average of the time differences.

3.2.3 Mapping tweets and bug reports

To understand whether the users' feedback from tweets is currently leveraged in the bug fixing process, we need to establish the links between bug-reporting tweets and bug reports. With the mapping between tweets and bug reports, we can distinguish

Figure 3.10: Example of a tweet that received a reply from the official Firefox account (Usernames are blurred to preserve privacy)

the bug reports whose issues are reported by Twitter users from the bug reports that are not mentioned on Twitter. Then, we compare the bug fixing process of the two groups of bug reports, in order to understand if the tweets are used in the bug fixing process in the current practice.

When mapping bug reports and tweets, we aim to find pairs of bug reports and tweets that describe a similar problem. Therefore, we compute the text similarity between bug reports and tweets for the mapping. However, a tweet may contain uninformative words that are irrelevant to the described problem. For example, in the following tweet "*Do you know if anyone else is having issues w/ Firefox 16 on Lion with copy and paste shortcuts? Paste works, but copy doesn't.*", we consider words like "*Do you know if anyone else is*" as uninformative. The uninformative words can inflate the similarity score. Therefore, it is important to extract key words that can

capture the main point of a given tweet. Then, we use the extracted keywords to represent the corresponding tweet, and take the extracted keywords as a query to search against all bug reports using an off-the-shelf search engine.

In the following subsections, we provide more details about our approach for mapping tweets to bug reports.

Tweet keywords extraction

In this study, we extract the n-grams of each tweet as keywords to describe its main point. Unlike bag of words, n-grams are sequences of n relevant words that do not appear consecutively by accident (*e.g., neural network* or *call cell phone*).

First, we remove the stop words from the tweets since the stop words do not contribute to describe the main topic of a tweet. Then, we extract the full list of n-grams from the entire collection of bug-reporting tweets using the Natural Language Toolkit[19] (NLTK) [22]. NLTK is a suite of program modules written in Python to support research in natural language processing and computational linguistics. We choose NLTK because it is a well established NLP tool that has been used in numerous prior research studies [89, 104, 76, 146].

After the full list of n-grams is extracted from the entire collection of bug-reporting tweets, we identify the n-gram(s) that appear in each tweet as its keywords. A given n-gram could appear in one or more tweet. For example, the tweet "*@firefox After some testing, it seems that firebug is the cause, since it is causing javascript to hang frequently*" has two matched bi-grams: "*firebug cause*"" and "*javascript hang*". Therefore, we use the two bi-grams as keywords to represent this tweet.

The extraction of n-grams has one parameter n, which is the number of words in a sequence. It can be any positive integer that is greater than one. However,

[19] http://www.nltk.org/

increasing n decreases the number of extracted n-grams, which leads to a lower chance of successfully matching tweets to n-grams. Tweets that do not contain n-grams can not be mapped to bug reports. To choose an appropriate n for this particular task, we compute the percentage of tweets that can be assigned n-grams with varying values of n. With $n = 2$ (a.k.a. bi-grams), we can assign n-grams to approximately 87% of the 6,044 subject tweets related to Firefox, and 90.3% of the 6,158 subject tweets related to Chrome. When setting $n = 3$ and $n = 4$, the percentages of tweets that can be assigned n-grams are respectively 78% and 65% of the 6,044 subject tweets related to Firefox. The percentage of tweets that can be assigned n-grams drops as we increase n. Besides, we observe that the words that are captured by the n-grams of sizes 3 and 4 are always already captured by the n-grams of size 2. For example, from the tweet: "*@firefox Version 37.0.2, no updates available. The problem sites give me: Secure Connection Failed. Same issue on clean install as well*", we obtain the following bi-grams: "*clean instal, connect fail, connect secur, fail issu, problem site, problem updat, updat version*", the following tri-gram: "connect fail secur", and no four-gram. Therefore, we choose $n = 2$ (*i.e.,* bi-grams extraction) in our study.

Tweet keywords querying

To identify the bug reports that are related to a given tweet, we use the extracted keywords (*i.e.,* bi-grams) as a query to search against all bug reports. We choose Lucene as our search engine, since it has been used in a prior study on bug reports [142], and has proved to work well in high profile platforms, such as Twitter, LinkedIn, and Jira [103]. Lucene [20] is a free and open source library for information retrieval.

For each release of each subject system, we build a Lucene database using all

[20] http://lucene.apache.org/core/5_3_1/

the bug reports belonging to the same release. Then, for each tweet, we use the extracted keywords (*i.e.,* bi-grams) as a query in Lucene and retrieve a list of bug reports. Tweets are also assigned to specific releases based on the dates they were posted. Only tweets and bug reports belonging to the same release can be mapped. The retrieved bug reports are ordered by the score of similarity to the query.

The Lucene similarity score is implemented as a variant of TF-IDF. In a nutshell, the similarity score varies with the number of times the query terms (*i.e.,* bi-grams of a tweet) occur in a document (*i.e.,* a bug report), and inversely with the number of times the query terms occur in all documents (*i.e.,* corpus of bug reports). Other factors are also factored in the Lucene similarity score, such as $coord(query, document)$ (*i.e.,* how many of the query terms are found in the document), and $lengthNorm(t, d)$ (*i.e.,* a measure of the importance of a term according to the total number of terms in the field). More details on the components of the Lucene similarity score can be found on the similarity Java class webpage [21]. At present, there is no absolute calibration for the highest score returned. It is difficult to determine from the similarity scores the level of relevance between a query and the matched document [112]. Therefore, we can not set a unique threshold to determine if a query matches a document. Alternatively, we choose top-K retrieved bug reports for each query as its matched bug reports. The smaller K is, the more strict the mapping is. We perform a sensitivity analysis by varying $K \in \{5, 10, 15\}$. We decide not to go beyond 15, since most users of search engines can retrieve a relevant result within the first 15 results returned by a search engine [115]. Otherwise, users would have changed their query. For instance, an average of 71.3% of searches on Google resulted in a click in the 10 first results. In particular, the first 5 results account for 67.6% of all the clicks [115].

[21] https://lucene.apache.org/core/5_3_1/core/org/apache/lucene/search/similarities/TFIDFSimilarity.html

3.3 Study Results

This section presents the approach and finding of the four research questions. For each research question, the motivation, the approach, and a discussion of the findings are presented.

RQ 3.1. How accurately can our approach map tweets and bug reports?

Motivation. Analyzing tweets has the potential to benefit the bug fixing process. For instance, it is likely to improve end-users' satisfaction if the development team fixes the bugs that are tweeted by many end-users as soon as possible. Mining problems reported in tweets can also help the development team catch missed bugs. However, the potential benefits of analyzing tweets rely on establishing accurate links between tweets and bug reports. In this question, we aim to evaluate the accuracy of our approach for mapping tweets and bug reports. Our evaluation provides a basis for further approaches on mapping tweets and bug reports.

Approach. To evaluate the accuracy of our approach, we perform a manual analysis in two steps:

1. We assess the mapping precision resulting from the different Lucene configurations (number of results $K \in \{5, 10, 15\}$). We accordingly select the appropriate number of results K to be used in the remainder of the study, to maximize precision.

2. We evaluate the different preprocessing steps listed under Section 3.2.1 for bug reports and Section 3.2.2 for tweets. We assess the precision gain resulting from each preprocessing step, following a similar approach as [138].

To conduct the manual analysis, we collect a sample of mapped bug reports and associated tweets. We randomly select a sample of 361 bug reports from each subject system. The sampled bug reports can statistically represent the population with a $95\% \pm 5\%$ confidence[22]. For each selected bug report, we identify all the tweets for which the bug report is returned as a match by Lucene.

We ask three non-author evaluators to assess the mappings of the sampled bug reports resulting from our final approach and the multiple baselines (*i.e.,* considering/not considering the different preprocessing steps). The three evaluators are all graduate students majoring in computer science. For each selected bug report, the evaluators are presented with the possibly matching tweets. The evaluators are asked to mark the tweets they believe are a possible match to the bug report. All evaluators work independently, and they are not told whether a baseline approach or our approach is being evaluated. To assess the agreement among evaluators, we use Fleiss' kappa [50]. Fleiss' kappa is a statistic to measure agreement among two or more evaluators for categorical items (*i.e.,* whether or not a pair of <bug report, tweet> is mapped correctly). Larger Fleiss' kappa means more agreement among evaluators, with a maximum value of 1 indicating complete agreement.

We compute the precision to measure the accuracy of the mapping between tweets and bug reports. Specifically, precision is calculated as the proportion of correctly mapped tweets relative to the total number of mapped tweets for each bug report. Higher precision values indicate better performance of the mapping. Given that we have a large amount of bug reports and tweets (*e.g.,* Firefox has 14,489 bug reports and 6,044 tweets), an exhaustive manual evaluation is required to identify all the pairs <bug report, tweet> that should be mapped and to compute the recall. Moreover, it

[22] www.surveysystem.com/sscalc.htm

is a very common practice to use precision to evaluate information retrieval tasks [29]. Therefore, we do not use recall but only precision for the evaluation.

Findings. Setting the parameter K to 5 in Lucene returns the most acceptable precision values. As described in Section 3.2.3, our approach has a parameter K that controls the number of retrieved bug reports for each query. The sensitivity analysis using $K \in \{5, 10, 15\}$ demonstrates that at least 47.8% of bug reports can be mapped to tweets. Specifically, when the number of retrieved bug reports is set to 5 (*i.e.,* $K = 5$), tweets can be mapped to 47.8% and 50.8% of bug reports in Firefox and Chrome, respectively. When increasing K to 10, tweets are mapped to more bug reports (*i.e.,* 65.2% in Firefox and 69.7% in Chrome). Increasing K can always result in more bug reports that are mapped to tweets with lower text similarity scores. Detailed coverage of bug reports is reported in Table 3.3.

The purpose of this analysis is to select a search strategy that achieves an acceptable precision/recall trade-off. Increasing K can always result in more bug reports that are mapped to tweets with lower text similarity scores (*i.e.,* higher recall and lower precision). However, the focus of this work is to study the differences between the bug reports associated to tweets and the bugs reports with no associated tweets. As such, it is more critical in this case to obtain the true links, rather than the totality of the links. The precision of the mapping drops for K values larger than 5, as expected. The average drop in precision shown in Table 3.4 (*i.e.,* -14% for $K = 10$ and -20% for $K = 15$) is significant and might constitute a threat to the validity of the mapping. Therefore, we use the $K = 5$ Lucene configuration in the remainder of the study.

Table 3.3: Coverage of bug reports that are mapped to tweets with varying configuration (*i.e.,* $K \in \{5, 10, 15\}$) of our mapping approach.

Browser	**Firefox**			**Chrome**		
Configuration of K (# of retrieved bug reports)	5	10	15	5	10	15
Coverage (% of mapped bug reports)	47.8%	65.2%	73.5%	50.8%	69.7%	79.8%

Table 3.4: Average precision of the mapping approach based on three different Lucene configurations ($K \in \{5, 10, 15\}$) and based on the results from three evaluators

	$K = 5$	$K = 10$	$K = 15$
Evaluator 1	76%	61%	58%
Evaluator 2	84%	71%	63%
Evaluator 3	78%	63%	59%
Average	**79%**	**65%**	**59%**

Identifying the bug-reporting tweets and extracting n-grams from tweets result in the highest precision gain of the mapping approach. We can observe the impact of each preprocessing step conducted prior to the Lucene mapping in Table 3.5 when: (i) removing noise from the bug reports (+3% precision gain); (ii) correcting anomalies in tweets (+4% precision gain); (iii) identifying the bug-reporting tweets (+11% precision gain); (iv) normalizing the bug reports and tweets (+3% precision gain); and (v) extracting n-grams from tweets (+9% precision gain). Our approach achieves a precision ranging from 76% to 84%, while the baseline approach involving no preprocessing step yields a precision from 47% to 52%. We compute the Fleiss' kappa among the three evaluators. The value of the Fleiss' kappa is 0.72, indicating a substantial agreement, according to [85]. A paired Mann-Whitney U test [91] is used to test the following alternative hypothesis:

H_a^1: *The precision of the final mapping approach is higher than the precision of the baseline approach.*

The Mann-Whitney U test reveals a significant improvement (*i.e.,* p-value is always less than 2.2e-16) in terms of precision between the baseline approach with no text processing and the final mapping approach. Better precision is achieved, thanks to the n-gram analysis and the identification of the bug-reporting tweets. Indeed, not all the tweets describe a bug, and usually only some words in a tweet are relevant to describing a bug.

Even though the overall precision returned by the mapping approach is acceptable, it still results in some false positives. For the purpose of highlighting some remaining challenges in mapping tweets and bug reports, we include here some examples of false mappings between tweets and bug reports. The tweet "*@Firefox is having some problems playing YouTube videos... Script error upon opening video pages as well.*" is matched to a bug report with the following title: "*YouTube Audio and Video downloader causing tab crash when loading Youtube homepage in E10S mode*". Even though the issue in both the tweet and the bug report is related to YouTube, the specificities of the issue are not the same. We find several similar instances in the set of mapped tweets and bug reports. The bug reports are longer in length and contain more details. Therefore, even if the terms in a tweet match parts of the bug report, there might be additional details in the bug report that describe details not included in the tweet, or not matching the content of the tweet at all. Additionally, we observe that the bug reports that contain more technical content (*e.g.,* references to code elements) are generally mismatched to the tweets. An example of a bug report that falls under this category is titled: "*Download indicator toolbar button depends on an odd XBL quirk*". The description of similar bug reports mostly contains references to code elements and objects, and does not follow the traditional structure (*i.e.,*

Table 3.5: Average precision of the mapping approach tweets resulting from multiple baselines

	No text preprocessing	Remove noise from BRs	Correct anomalies in tweets	Identify bug-reporting tweets	Normalize BRs & tweets	Extract n-grams
Evaluator 1	47%	50%	56%	64%	68%	76%
Evaluator 2	52%	54%	58%	71%	75%	84%
Evaluator 3	48%	52%	54%	66%	67%	78%
Average	**49%**	**52%** (+3%)	**56%** (+4%)	**67%** (+11%)	**70%** (+3%)	**79%** (+9%)

reproducible steps, expected results, and actual results). Therefore, similar instances result in false mappings using our approach. Lastly, we observe that the filtering of the bug-reporting tweets has limitations and results in some tweets to be wrongly selected as relevant. For instance, the following tweet "@firefox you claim bug 987323 is fixed on versions 33 and up, but I am having that issue on version 33.1.1. What is going on?" does contain the bug terminology. However, it does not describe what the bug is about, and therefore results in false mappings to the bug reports.

The precision of our mapping approach varies from 76% and 84%, which outperforms the baseline approach with a large margin. Extracting and using keywords from a tweet can effectively represent the tweet compared to using its original form. The precision is also improved by identifying the bug-reporting tweets.

RQ 3.2. What web browser issues receive more feedback from end-users through Twitter?

Motivation. End-users and the development team may view a software product from different perspectives. End-users may be particularly interested in some issues based on their perception of the products, while the development team may focus on

some other issues based on the results of the test plans. The mismatch of the focus might affect the satisfaction of end-users. Therefore, it is interesting to identify what issues are most critical to the end-users in the context of web browsers.

Approach. To address this question, we first extract the topics of bug reports, and then compare the topics between bug reports with associated tweets and bug reports without associated tweets.

Topics are extracted using Latent Dirichlet Allocation (LDA) [23], which has been successfully used to identify software concerns [11] and analyze bug reports [123, 87]. LDA has two major parameters: the number of topics (N_{topics}) and the number of words (N_{words}). Each document (*i.e.,* bug report) is assigned a vector of probabilities (length is N_{topics}) that describe the chance of each topic to appear in the document. Each topic is described by a collection of words (size is N_{words}).

To set an appropriate number of LDA topics (N_{topics}), we use an R package called *Ldatuning*[23]. *Ldatuning* reports the results of four metrics, Arun2010 [9], Cao-Juan2009 [32], Devaud2014 [46], and Griffiths2004 [64]. The optimal number of topics is decided with a majority vote of the metrics. Based on the results of the *ldatuning* package, we set the number of topics to 200 (*i.e.,* $N_{\text{topics}} = 200$). We further set the number of topic words to 20 (*i.e.,* $N_{\text{words}} = 20$) to ease the understanding of each topic. To ensure that extracted topics are comparable across the two subject systems, we perform a single run of LDA on the corpus of bug reports from both Firefox and Chrome.

To comprehend the extracted topics, we ask three people to manually label and categorize the topics. The three people include two graduate students in computer

[23]https://cran.r-project.org/web/packages/ldatuning

science who also contributed to the manual analysis conducted in RQ1, and one software engineer with two years of experience in software development. None of them authors this paper. Each topic is labeled based on the list of 20 topic words. For example, the label *unresponsiveness of the browser* is assigned to the topic with the following 20 keywords: "*hang, freez, stop, firefox, respond, continu, forc, minut, unrespons, quit, wait, complet, time, frozen, kill, happen, respons, recov, exit, sudden*" (all are stemmed). In case of disagreements among the evaluators, a short discussion among the evaluators is conducted to reach a consensus. If no consensus is reached, the label agreed on by the majority is used.

To provide a high-level view of the extracted topics, the topics are further categorized based on the browser feature that they are closest to by the authors. One of the evaluators is asked to review the resulting categorization, which we then refine based on the received feedback. Table 3.6 shows all ten categories of topics and the labels of topics within each category. For instance, the topics *speed of opening a new page* and *memory usage* are assigned to the category *performance*, as both topics describe issues regarding to the performance of the browser. We exclude from the list of resulting topics the following categories of topics: *a)* the 8 topics that could not be labeled (*e.g., "firefox, command, state, theme, agent, expect, interfac, gnome, broken, fedora, notic, wrong, user, persona, regular, exist, skin, aero, area, classic"*); and *b)* the 70 topics that contain only bug reports related keywords (*e.g., "result, actual, step, agent, expect, gecko, build, user, reproduc, window, mozilla, id"*). This results in 122 topics out of the 200 originally extracted topics. During the manual labeling of the remaining topics, we observe that some topics are duplicated. Therefore, we merge the duplicated topics, and end up with the 79 topics shown in Table 3.6.

Table 3.6: All ten categories of topics and topic labels within each category.

Topic category	**Topic labels**
Audio and video	Audio related - Video and flash
Coding and debugging	Test cases - CSS properties - Coding and debugging - Data structures
Configuration and updates	Remote configuration - Packages installation - Synchronization of services - Firefox profiles - Client/server interactions - Versions and upgrades - User privileges - Icon customization
Functionalities	Browsing modes - Search engine - Cookies and privacy - Printing - Bookmarks - Management of multiple windows - Editor and text manipulation - User support - User accounts - Passwords management - Cache - History
GUI appearance	Resolution - Window size- Colors and transparency - Size and positioning of window - Text appearance - Browser themes
GUI logic	Correctness of display - Rendering of files - Mouse actions - keyboard actions - Facebook-related - Correctness of actions in browser - Email-related - Tabs actions - Open and click actions - Pictures and images - Files and directories - Navigation - Page loading - Screeshots - Input fields - Zoom action - Buttons - Menus - Visibility of results - Text suggestions - Session restoration - Progress visibility - Error messages - Address bar - Languages related - Input fields - Visibility of progress - Tabs action
OS and hardware	Management of processes - Devices - OS related - Windows related - MAC related - Drivers
Performance	Browsing modes - Speed of opening a new page - Performance of animations - Starting Firefox - Time performance - Unresponsiveness - Memory usage
Security	Web security - Network security
Tools and extensions	Firebug - Management of extensions - Libraries - Inspector

Finally, we examine what web browser issues receive more feedback from end-users through tweets. We divide all bug reports into two groups: one group consists of all the bug reports that have associated tweets and the other group includes all the remaining bug reports. To study which topics are more appealing to end-users, we compare the probability of each topic category to appear in bug reports between these two groups. We define the null hypothesis as:

H_0^2*: there is no difference in the probability of bug reports to have a particular topic*

category between bugs with and without associated tweets.

To test the null hypothesis, we apply the Fisher's exact test [117] with the 95% confidence level. If there is statistical significance (*i.e.,* p-value < 0.05), we reject the null hypothesis. We further compute the Odds Ratio (OR) [117] to determine if the corresponding topic category has a higher or lower likelihood to appear in the bug reports that have associated tweets (*i.e.,* bugs posted by end-users).

Findings. End-users are more interested in web browser issues related to performance, security and audio/video in both subject systems. Specifically, performance issues have 2.01 times higher chance to appear in bugs with associated tweets than in bugs without associated tweets in Firefox. Similar finding (*i.e.,* 2.44 times for performance issues) is observed for Chrome. Detailed results for all topic categories in both subject systems are presented in Table 3.7. We can clearly see that end-users do not care equally about all issues, and are likely to complain about specific issues. End-users are more concerned about issues related to performance, security and audio/video in both subject systems. In particular, issues related to audio/video are the most sensitive ones to end-users. The development team of the two subject systems may want to pay special attention to code changes that impact modules on audio/video, performance and security, in order to reduce the amount of possible complaints from end-users.

End-users are less interested in web browser issues related to GUI appearance in both subject systems. Issues related to GUI appearance (*e.g.,* text appearance, window size, resolution, and browser themes) are more likely reported in non-tweeted bug reports. In Chrome, bug reports containing *coding and debugging*

Table 3.7: Odds ratio and the corresponding p-value of the Fishers' exact test on the appearance of topic categories. (An OR > 1 indicates that the corresponding topic category is more likely to be posted by end-users on Twitter, and an OR < 1 indicates the opposite; n.s = not significant.)

	Firefox		**Chrome**	
	Odds ratio (OR)	**p-value**	**Odds ratio (OR)**	**p-value**
Audio/video related	**3.08**	**2.08e-05**	**2.64**	**3.98e-05**
Coding and debugging	0.70	n.s	**0.53**	**4.35e-02**
Configuration and updates	1.56	n.s	0.97	n.s
Functionalities	1.35	n.s	1.13	n.s
GUI appearance	**0.42**	**6.55e-05**	**0.55**	**3.38e-04**
GUI logic	0.98	n.s	1.12	n.s
OS and hardware	0.86	n.s	0.68	n.s
Performance	**2.01**	**1.20e-03**	**2.44**	**1.36e-05**
Security	**2.27**	**9.71e-09**	**1.97**	**8.01e-06**
Tools and extensions	0.68	n.s	1.27	n.s

issues are more likely a concern of developers, rather than end-users (as expected). We do not find evidence of this observation in Firefox (OR = 0.70 but p-value $>$ 0.05).

As a summary, end-users are more likely to complain about *how well* the web browser works (*e.g.,* performance, security and audio/video), rather than *how it looks* (*e.g.,* GUI appearance). Although our mined topics are for two web browsers, the approach is generalizable to other subject systems. The development team of other systems can apply our approach to mine the topics that their end-users are interested in. The development team can save their effort, if they prioritize development activities by matching the interest of end-users.

Our results successfully highlight web browser issues that are more sensitive to end-users. For both of our subject systems, end-users are more concerned about how well the web browser works rather than how it looks.

RQ 3.3. Does the development team handle a bug report differently if the

problem is mentioned on Twitter?

Motivation. It is unclear if the current practice in bug fixing is impacted by the feedback posted on Twitter by the end-users. It would be interesting to know whether the information provided by end-users on Twitter is currently leveraged in the bug fixing process. Specifically, we examine whether the bugs reported on Twitter receive a different treatment in the bug fixing process from two aspects: the time taken at each stage of the bug fixing process, and the severity or priority of bug reports that is often used for prioritizing the bug reports.

Approach. To address this question, we define two non-overlapping groups of bug reports based on the presence of associated tweets: 1) non-tweeted group that contains bug reports without associated tweets, and 2) tweeted group that has bug reports with associated tweets. For the second group, we further divide it into two non-overlapping sub-groups: 1) weakly-tweeted group in which bug reports have fewer associated tweets and 2) heavily-tweeted group in which bug reports have more associated tweets. We use the median number of associated tweets as the threshold to obtain the two sub-groups.

Then, we examine whether there is significant difference in the bug fixing process among the different groups in terms of the time aspects and the severity/priority of the bugs.

1) Time aspects: We measure the time aspects of the bug fixing process using three metrics (see Section 3.2.1): DBR that captures how long it takes for a bug to get the first response from developers; DBA that describes how long it takes to assign the bug to a developer; and DBF that measures how long it takes to fix the bug.

Similarly to RQ 3.2, to study the current treatment of tweets in the bug fixing

process, we apply the Fisher's exact test and compute the odds ratio. A contingency table needs to be built for performing the tests. Each contingency table is built using two dimensions (*i.e.,* the number of tweets associated to the bug reports and the speed at which the bug reports are addressed). As we obtain two groups (*i.e.,* non-tweeted and tweeted) or three groups (non-tweeted, weakly-tweeted and heavily-tweeted) from the tweet perspective, we need to obtain several groups for each metric to build the contingency tables. For each metric, we use the median value to separate bug reports into different groups. Then, we can obtain a contingency table based on each metric and the tweet information.

For each metric, we first examine the treatment of bugs based on the presence or absence of tweets associated to the bug reports (*i.e.,* non-tweeted and tweeted). Accordingly, we define and test the following null hypothesis:

H_0^3a*: There is no difference in the probability of bug reports to be addressed (*i.e., *responded to, assigned, or fixed) within a certain time duration between bugs with and without associated tweets.*

Second, we examine the treatment of bug reports with different levels of associated tweets (*i.e.,* non-tweeted, weakly tweeted, and highly tweeted) based on each metric. We define the null hypothesis as:

H_0^3b*: There is no difference in the probability of bug reports to be addressed (*i.e., *responded to, assigned, or fixed) within a certain time duration between bugs with different levels of associated tweets.*

Since we conduct multiple significance tests between the groups of tweeted and non-tweeted bug reports, we correct for the multiple comparisons using the Benjamini-Hochberg correction [16, 17].

To further assess the relationship of tweets with the bug fixing intervals, we build a linear regression model to model the bug fixing intervals given a set of predictors (*e.g.,* # of tweets). The predictors we control for in the regression model are the number of tweets, the severity, the topic, and the affected component of the bug reports. We measure the goodness of fit of the regression model using the coefficient of determination R^2. R^2 measures how close the fitted regression model is to the actual values of the bug fixing intervals. Then, we report the significance of each predictor on the bug fixing intervals DBR, DBA, and DBF. A small significance (p-value) indicates that it is unlikely we will observe a relationship between the predictor (*e.g.,* # of tweets) and the bug fixing interval (*e.g.,* DBF) due to chance.

To verify our findings, we get in touch with developers at Firefox and Chrome. For Firefox, we start a thread at a specialized forum for the contributors in charge of the social media support[24]. Three Firefox contributors were kind to respond to the message posted. For Chrome, we start a thread at the developers forum [25]. Unfortunately, we could not get feedback from the chrome developers. We further investigate the possibility of using tweets to file bug reports. Two weeks after the release of Firefox 53.0.2, we use our approach to collect the bug-reporting tweets. We manually summarize the results from the collected tweets and file 2 bug reports to the issue tracking system Bugzilla[26][27]. We include in the bug reports the links to the users' tweets.

2) Severity/priority: In issue tracking systems, importance tags (*i.e.,* severity and priority) are assigned to bug reports to assist in the bug triaging process. The

[24]https://support.mozilla.org/en-US/
[25]https://groups.google.com/a/chromium.org/forum/#!forum/chromium-dev
[26]https://bugzilla.mozilla.org/show_bug.cgi?id=1361468
[27]https://bugzilla.mozilla.org/show_bug.cgi?id=1361498

severity of a bug report describes how damaging a bug is to the system. The priority, on the other hand, defines the order in which a bug should be resolved and deployed. In many cases, severe bugs are also assigned higher priority. In other cases, a bug could possibly have low severity (such as a misspelled title on the home page of a website), but could be assigned high priority because of the visibility of the bug to the end-users. It is also possible for a bug to have high severity, yet be assigned low priority because the bug only occurs in rare occasions. In most cases, only one importance tag is mainly used by software systems in their issue tracking systems.

In Firefox, where severity is the main importance tag, the following severity levels are used to describe the severity of the bug reports: *blocker*, *major*, *critical*, *normal*, *minor* and *trivial*. We identify the severe bugs using the labels *blocker*, *major*, and *critical*. The bugs with normal severity are identified with the labels *normal*, *minor* and *trivial*. In Chrome, the priority tag is used and has the following values: 0, 1, 2, and 3. We classify the bug reports with the priority values 0 and 1 as the most urgent, and the bug reports with values 2 and 3 as the less urgent.

We assess whether the importance level (*i.e.,* severity or priority) of a bug has any association to its number of associated tweets. In each subject system, we classify the bug reports into the highly important bug reports (high severity in Firefox and high priority in Chrome), and the less important bug reports. We use Fisher's exact test to assess whether the presence and absence of associated tweets has an association to the importance level, using the following null hypothesis:

$H_0^3 c$: *there is no difference in the probability of bug reports to have a certain severity/priority level between bugs with and without associated tweets.*

Table 3.8: Odds ratio and the corresponding adjusted p-value of the Fisher's test on time intervals DBR, DBA and DBF in Firefox and Chrome. (n.s = not significant)

	Firefox			Chrome		
	DBR	**DBA**	**DBF**	**DBR**	**DBA**	**DBF**
Non tweeted vs. tweeted	1.08 (n.s)	0.94 (n.s)	1.19 (n.s)	1.13 (2.5e-04)	1.09 (3.2e-03)	1.10 (2.7e-03)
Non tweeted vs. highly tweeted	1.02 (n.s)	0.63 (n.s)	1.30 (n.s)	1.20 (2.9e-06)	1.16 (7.1e-05)	1.12 (2.8e-03)
Non tweeted vs. weakly tweeted	1.17 (n.s)	0.95 (n.s)	1.04 (n.s)	1.05 (n.s)	1.02 (n.s)	1.08 (n.s)

Similarly, we assess if the different levels of associated tweets are associated to the importance level, by testing the following null hypothesis:

$H_0^3 d$: *there is no difference in the probability of bug reports to have a certain severity/priority level between bugs with different levels of associated tweets.*

Finally, we compute the odds ratio.

Findings. We find no evidence that developers react differently to the bugs that are tweeted in terms of the time aspects of the bug fixing process. Table 3.8 shows the odds ratios and adjusted p-values resulting from the Fisher's test. We find that, for all time aspects in Firefox, there is no significant difference (*i.e.,* p-value > 0.05) between tweeted and non-tweeted bugs. Surprisingly, in Chrome, we find that tweeted and highly tweeted bug reports are likely to be addressed (*i.e.,* responded to, assigned, and fixed) in a slower fashion compared to the non-tweeted bug reports. However, the odds ratios are all between 1.09 and 1.20, indicating a low likelihood for tweeted and highly tweeted bug reports to receive a slower response, assignment, and fixing times.

We find no evidence that the number of associated tweets has a significant relationship with the three bug fixing intervals. We show in Table 3.9 the

Table 3.9: Goodness of fit (R^2) and significance results (p-value) of the linear regression models (n.s = not significant)

	Firefox			**Chrome**		
	DBR $R^2 = 0.56$	**DBA** $R^2 = 0.41$	**DBF** $R^2 = 0.56$	**DBR** $R^2 = 0.39$	**DBA** $R^2 = 0.43$	**DBF** $R^2 = 0.59$
# of tweets	n.s	n.s	n.s	n.s	n.s	n.s
Severity	1.8e-15	3.8e-10	8.5e-10	4.7e-07	9.8e-06	6.5e-07
Topic	1.6e-05	n.s	5.3e-06	2.7e-03	4.1e-04	2.4e-03
Component	n.s	n.s	5.6e-04	n.s	n.s	n.s

goodness of fit of the three regression models. We observe that the regression model associated with the interval DBF returns the highest goodness of fit (*i.e.,* $R^2 = 0.56$ in Firefox and $R^2 = 0.59$ in Chrome). We further report the significance values of the predictors in Table 3.9. While the severity of the bug has the highest significance on the bug prediction intervals (p-value < 0.05), the number of tweets returns the lowest significance, thus confirming the results obtained from the previous analysis using Fisher's exact test.

Firefox contributors report that bug discovery through Twitter could be challenging. Unfortunately, we were not able to receive any feedback from the Chrome developers. Considering the lack of evidence showing a relationship between the tweets and the bug fixing intervals (as observed from the previous findings), we get in touch with Firefox contributors who are in charge of the social media support. The contributors report the following difficulties in possibly using Twitter to collect bugs:

"*Quote 1: Tweets may be of interest in so far as if something is reported as a problem with a lot of tweets, it may indicate a severe issue. However, it is a very difficult media to use for communicating information well.*"

"*Quote 2: Bug discovery using Twitter is not easy. A bug is not as simple as it*

sounds."

"*Quote 3: As a social media contributor, I've filed bug reports for users that have reported issues via Twitter. The unfortunate fact is that users don't get back to us or continue with the bugs that's been reported on their behalf.*"

Indeed, tweets are short messages with at most 140 characters and they may be too generic or lack of details to replicate the reported issues. In addition, the high amount of tweets (as shown in Table A.1) can be overwhelming, and it might be difficult for developers to manually extract useful information from the tweets. Our approach can help the development team to automatically identify useful feedback from the tweets.

The bugs reported based on the Twitter feedback are acknowledged by the developers. The first observed bug is related to the appearance of the URL bar on systems with RTL languages (*e.g.,* Hebrew and Arabic). We receive a response from the Firefox developers 3 days later to inform us that the issue has been fixed and the Twitter user has been notified. The second observed bug after the release of Firefox 53.0.2 is the YouTube music no longer playing after Firefox is sent to the background. We were able to collect complaints from 7 Twitter users. The Firefox developers provided a response on the same day and linked the bug report to an existing bug report. The reported problem has not been fixed yet as of the time of this writing.

In Firefox, bug reports with associated tweets are more likely to be severe than the non-tweeted bug reports. However, such trend is not observed in Chrome. Table 3.10 presents the detailed results of the Fisher's exact test and the odds ratio. In Firefox, bug reports with associated tweets are 1.86 times

Table 3.10: Fisher's test results regarding the relation between severity/priority level and the level of tweet involvement.

	Firefox (Severity)		Chrome (Priority)	
	Odds ratio	***p*-value**	**Odds ratio**	***p*-value**
Non tweeted vs. tweeted	1.86	2.02e-03	0.98	n.s.
Non tweeted vs. highly tweeted	2.41	1.28e-05	0.99	n.s.
Non tweeted vs. weakly tweeted	1.18	n.s.	0.97	n.s.

more likely to be treated as more severe than bug reports without associated tweets. The chance for a bug report to be labeled as severe increases to 2.41 times, if the bug report has a high number of associated tweets (*i.e.,* highly-tweeted). Among the tweeted Firefox bug reports, those that are associated with the more popular tweets (*i.e.,* tweets that are retweeted, marked as favorites, receive a reply from Firefox, and generate active and longer conversations) are likely more severe. The popularity metrics are explained in Section 3.2.2. Indeed, one of the Firefox contributors suggests in *Quote 1* that a bug could be perceived as severe if reported by many users on Twitter.

We do no find evidence of the impact of tweets on the bug fixing process in both subject systems, with the exception of the Firefox tweeted bug reports which are more likely to be severe. Therefore, an automated approach could be used to leverage the feedback from end-users and possibly benefit the bug fixing process

RQ 4.3. Can we use tweets to achieve an early discovery of bugs?

Motivation. The findings of RQ 3.3 indicate that either tweets have no impact on the current practice of the bug fixing process or tweets are not considered useful by the development team. The feedback we receive from the Firefox contributors points towards the second possibility. We are wondering if tweets can still provide some

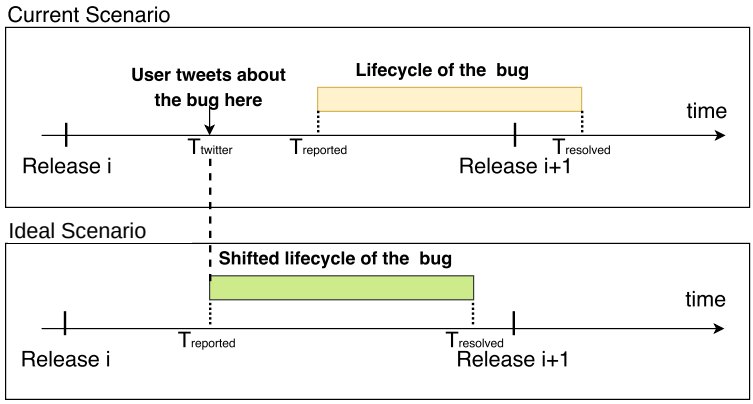


Figure 3.11: An illustration of the real scenario (a bug is not reported at the time the problem appears on Twitter) and the ideal scenario (a bug is reported immediately after the problem is posted on Twitter) for the bug fixing process.

value to the bug fixing process. As tweets provide fast feedback from end-users, the development team may get to know the problems described in tweets earlier than when the problems are currently reported and filed in the issue tracking systems. Even if a 140-character tweet may not sufficiently describe a problem, developers can contact the end-user who posted the tweets for more details. Therefore, we are interested to find out if tweets can lead to an early discovery of bugs.

Approach. To address this question, we consider two scenarios as described in Figure 3.11. The first scenario is the one that we observe in the current practice of the bug fixing process: no relation between the time a bug is reported (*i.e.,* $T_{reported}$) and the time a tweet related to the bug is posted (*i.e.,* $T_{twitter}$). In such a case, a bug can be reported before or after the posting of related tweets. The second scenario describes an ideal case where a bug is filed into the issue tracking system immediately after the problem is posted on Twitter. If the total bug resolution time (*i.e.,* from creation to resolution) remains the same, earlier discovery of bugs might lead to a subset of the bugs to be fixed earlier.

The ideal scenario is achievable, as our approach can automatically link tweets and

bug reports with a substantial precision. A newly posted tweet may reveal a problem that is unknown to the development team, if no bug reports are retrieved for the tweet. Developers could further contact the end-user who posted the tweets to obtain more details and file a bug report. Even though the Firefox twitter account is quite interactive with the users and Chrome is as well but to a lesser degree, many tweets addressed to both accounts are irrelevant and would require resources to extract, process, and filter. Our collection and filtering approach to essentially collect bug-reporting tweets could help focus the efforts resources towards the relevant feedback from the Twitter users. This would be particularly important after a new release is deployed and users are quick to report problems they face after the browser is auto-updated on their diverse environments. The data collected from Twitter shows that the official Twitter accounts of Firefox and Chrome are only able to reply to 11.9% and 7.4% of the tweets from the users, respectively.

First, we investigate to what extent bugs can be discovered earlier by monitoring tweets. We compute the maximum number of bugs that are reported after their corresponding tweets are posted. Each bug report is associated to zero or more tweets. For each bug report, we use the time of the earliest associated tweet as the earliest time when the bug was reported by an end-user on Twitter. We denote the time as T_{twitter}. The bug reports without associated tweets are excluded from this experiment, as they could not have been discovered through Twitter. We compare the timestamp T_{twitter} to the timestamp T_{reported} when a bug is reported, and count the number of bugs that satisfy the condition $T_{\text{twitter}} < T_{\text{reported}}$.

Second, we examine the number of bug fixes that can be delivered in the next release assuming that *a)* a bug is reported immediately after the problem is posted

Table 3.11: Summary of RQ 3.3 results.

Subject system	**Firefox**	**Chrome**
# of all bug reports	14,489	15,771
# of mapped bug reports (% of bugs among all bugs)	6,926 (47.8%)	8,012 (50.8%)
# of bugs reported earlier on Twitter (% of bugs among all bugs) (% of bugs among all mapped bugs)	4,839 (33.4%) (69.9%)	5,283 (33.5%) (65.9%)
Average improvement (days)	8.4	7.6

on Twitter, and *b)* the total bug resolution time remains the same. We calculate the updated number of bugs that can be fixed before the next release. For all bug reports with $T_{\text{twitter}} < T_{\text{reported}}$, we intentionally replace the time when a bug is currently reported with the time of the earliest associated tweet (*i.e.,* setting $T_{\text{reported}} = T_{\text{twitter}}$). We keep the duration for each step in the bug fixing process (*e.g.,* bug triaging, fixing, and verification) unchanged. We count the number of bugs that can be "fixed" before the next release, including both the bugs currently fixed before the next release, and the bugs that can be potentially solved before the next release (*i.e.,* when setting $T_{\text{reported}} = T_{\text{twitter}}$).

Findings. With the Lucene mapping that retrieves five bug reports in each query (*i.e.,* $K = 5$), 33.4% and 33.5% of bugs could have been reported earlier in Firefox and Chrome systems, respectively. Moreover, Firefox bugs could have been reported averagely 8.4 days in advance, and Chrome bugs could have been reported on average 7.6 days earlier. Table 6.6 shows the percentage of bugs that can be discovered earlier on Twitter and the average time that can be saved to discover bugs.

A subset of bugs could be fixed earlier, in cases where the development team has the resources to address the extra bugs. For instance, there are on average 188 bugs fixed before the next release in the 37 Firefox releases; 24 extra bugs (*i.e.,* 212 bugs in total) could possibly be fixed before the next release. In the 34 Chrome releases, the average number of bugs fixed before the next release is 209. An extra 32 bugs (*i.e.,* 241 bugs in total) could possibly be fixed before the next release. The above is an ideal scenario that assumes the availability of resources to fix the extra bugs. In a real life scenario, developers might not have the resources to address all the extra bugs before an upcoming release.

For each subject system, we further perform a paired Mann-Whitney U test to verify the following alternative hypothesis:

H_a^4: *The number of bugs that could be fixed before the next release is statistically higher than the current number of fixed bugs.*

The p-value is always less than 2.2e-16 in both subject systems, thus confirming the alternative hypothesis. The number of bugs that could have been fixed before the next release is always greater than the currently fixed bugs in both the Chrome and the Firefox releases. To quantify the magnitude of the observed increase, we calculate the effect size using Cliff's delta [111]. Cliff's delta is non-parametric, thus it does not make assumptions about the distribution of the data. It is reported to be more robust than Cohen's d [111]. Cliff's delta reflects the degree of overlap between the two distributions. It ranges from -1 (if all values in the first distribution are larger than the second) to +1 (all values in the first distribution are smaller than the second). When the two distributions are equal, it is equal to 0 [36]. Following the guidelines of prior work [65, 132, 37], we interpret the effect size e as small for

$0.147 < e < 0.33$, medium for $0.33 < e < 0.474$, and large for $e \geq 0.474$. In both browsers, we find a *large* effect size between the two distributions (*i.e.,* the currently fixed number of bugs and the possibly fixed number of bugs).

At least a third of the bugs could have been reported to the development team earlier, and a subset of the bugs could possibly be fixed earlier (if resources permit). To achieve this goal, we encourage end-users to promptly report problems on Twitter and the development team to utilize our approach to collect the bug-reporting tweets.

3.4 Threats to validity

We discuss the threats to validity of our case study following the common guidelines provided by [147].

Threats to conclusion validity concern the relation between the treatment and the outcome. The conclusion validity threat mainly comes from the inherent errors that come with processing the natural language. Both bug reports and tweets are written by users and are very prone to errors. Tweets are more particularly prone to the use of abbreviations, misspelled words, and non-standard orthography. We attempt to address these concerns by identifying common abbreviations and typos in tweets and correcting the identified misspellings before further processing. Another threat to the validity of the conclusions comes from the use of the bug fixing intervals (*e.g.,* time to fix a bug) to investigate bug reports. Open source projects are mainly maintained by volunteers who contribute based on their time availability. Therefore, bug fixing intervals are impacted not only by the nature of the bugs, but also by the developers' availability (which we do not consider in this study). To mitigate this

threat, we propose to evaluate in future work the impact of end-users' feedback on the bug fixing process using other techniques. Last but not least, since we propose an approach with an average precision value of 79% and unknown recall, we introduce a threat to the validity of the conclusions in RQs 3.2 to 3.4. In RQs 3.2 to 3.4, we mostly distinguish two groups of bug reports: tweeted and non-tweeted. The main threat of having an unknown recall is mistakenly classifying a bug report as non-tweeted, when in reality it does have tweets associated to it that were missed by the mapping approach. The precision value, on the other hand, indicates that slightly over 20% of the mapped bug reports and tweets are mistakenly mapped. Thus, some of bug reports in the tweeted group might have in reality either *a)* no tweets associated to them, or *b)* a lower number of associated tweets. The latter case is not critical as a bug report with less associated tweets remains a tweeted bug report. However, the former case presents a threat to the validity of the comparisons in RQs 3.2 and 3.3. As this is the first attempt to link bug reports and tweets, we wish to improve in our future work the accuracy of the mapping. Besides, we also attempt to verify some of our findings by reaching out to the developers, and by submitting bug reports based on the data we automatically collect from Twitter.

Threats to internal validity concern the selection of subject systems and analysis methods. To lower the bias in the manual evaluation in our work, we invite three evaluators (non-authors) to evaluate the mapping between tweets and bug reports, and to manually label the topics generated from the bug reports. It is possible that evaluators could be uncertain about the correctness of a mapping between a tweet and a bug report. Therefore, it is a possible threat to the validity of our results to adopt a binary evaluation (*i.e.,* mapping is either correct or incorrect). We further

assess the agreement among the evaluators. We find a substantial agreement, thus adding confidence to the evaluation results. As far as the analysis methods used to pre-process the bug reports (particularly the removal of noise from the bug reports), we are aware of the existence of more sophisticated methods to classify the content of similar content (*e.g.,* development emails) at line level [10,]. We follow a heuristic-based simpler approach in this study that does not require the training of a large dataset at line level, and is tailored to the content of the bug reports. However, we will refine our approach in our future work.

Threats to external validity concern the possibility to generalize our results. In this study, we mainly focus on the web browsers Firefox (releases V5.0 to V41.0) and Chrome (releases V15.0 to V48.0), as subject systems. Our findings are specific to these two subject systems, although one of them is a popular example of an open source software, while the other is a proprietary software. Some of the findings might not be directly applicable to other software systems, such as the types of bugs most discussed by users on Twitter. However, our approach can be applied to map tweets to bug reports, and identify the bugs with a large impact on the end-users. In the future, we plan to apply our approach on different types of the subject systems that do not adopt the rapid release cycle.

Threats to reliability validity concern the possibility of replicating the study. We attempt to provide all the necessary details to replicate our study. Bugzilla and the Chromium tracker are publicly available to obtain the bug reports. Tweets can also be obtained from Twitter through their search features.

3.5 Summary

In this chapter, we investigate the usefulness of crowd-sourced feedback from Twitter to benefit the bug fixing process. We propose an approach to map bug-reporting tweets to bug reports. Second, we identify the types of bugs that are most critical to users. We find that end-users are more concerned about how well the systems work (*e.g.,* performance or streaming quality), rather than the appearance of the systems (*e.g.,* resolution or text appearance). Then, we study whether developers respond faster to the bugs that are associated with tweets. We find that the tweeted bugs do not get special attention from developers, suggesting that either there is no close monitoring of bug-reporting tweets or that tweets are not considered as a useful source of information. Finally, we find that at least 33% of Firefox bugs and Chrome bugs can potentially be discovered earlier based on the prompt tweets from end-users after a new release. This might allow the repair of more bugs before the release of the next browser version.

Chapter 4

An Empirical Study on the Teams Structures in Social Coding

In this chapter, we investigate the value of social organizational structure on the code review process of the software projects. We specifically investigate the team structures formed in social coding websites. We first identify the motivation behind this study, and present how we designed the study. We then present our findings, and discuss the threats to the validity.

4.1 Problem and Motivation

Social coding websites (*e.g.,* GitHub), provide a friendly platform for source code management, issue tracking, and networking among distributed communities [41]. The open source software development benefits from social coding websites, by improving collaboration [41]. The core of many social coding websites is the pull request feature (a.k.a. the pull-based development model) [12]. A developer (*i.e., contributor*) is free to create a local copy of the project repository, make code changes, and

submit a pull request to the project owner. A project owner, maintainer, or integrator is responsible to respond to a pull request by reviewing the code changes and determining if the pull request can be integrated into the main branch of the project.

The pull-based development model eliminates the need for a shared repository, lowers the upfront coordination, and decreases the barriers for the first-time contributors [61]. As such, many projects adopt the the pull-based development model, as a substitution to the past collaboration channels, such as submitting patches via issue tracking systems and/or mailing lists [21][55]. In terms of popularity, a study by [61] reports that pull requests and shared repositories are equally used among GitHub projects ($\approx$ 14% of the projects each), with the remaining projects being single-developer projects. The pull-based development model is particularly appreciated for separating the development effort from the decision making process about the submitted changes [61].

The performance of the pull-based development model depends not only on the quality of the submissions made by the contributors but also on the processing of the pull requests by the project maintainers (particularly, the integrators of pull requests) [63]. A qualitative study by [62, 63] shows that the integrators struggle to review or motivate other developers to review the submitted changes. The lack of responsiveness of the project maintainers is a common complaint from the contributors[62]. The low responsiveness in processing pull requests delays the integration of code changes on new features and bug fixes, therefore it weakens the power of the pull-based development model.

In this chapter, we study the types of team structures that possibly impact the performance of the pull-based development model. We build the pull-based networks

of 7,850 GitHub projects. In a pull-based network, two developers are connected if one of them has merged at least one pull request that was submitted by the other. We describe the pull-based networks by a set of network metrics, such as the centralization, and the reciprocity. The network metrics capture the roles of developers as integrators or contributors or both (*i.e.,* reciprocity). The network metrics can also identify the existence of core developers, or equally important participants (*i.e.,* centralization). We use the network metrics of the pull-based networks to infer the team structures formed in the GitHub projects. A team structure reflects how a development team self-organizes as they submit and review the pull requests. We systematically identify the set of existing team structures based on a set of influential network metrics from the open source projects.

Specifically, we investigate the following four research questions:

RQ 4.1 What are the influential network metrics on the performance of the pull-based development model?

We compute the network metrics of the pull-based networks. Then, we compute four performance metrics that reflect the productivity and efficiency of a team in managing the pull requests. We build a regression model to identify the influential network metrics on the performance of a team. We find that three metrics (*i.e.,* density, out-degree centralization and reciprocity) are significantly associated with all four performance metrics.

RQ 4.2 What are the common team structures in the pull-based development model?

We capture the team structures using the three influential network metrics that are identified in RQ 4.1. We define the possible team structures by discretizing the values of the three influential network metrics (*e.g.,* the out-degree centralization metric is discretized into 3 levels based on its distribution). We observe that 8 dominant team structures are adopted by over 90% of the projects. More than a third of the projects follow a team structure characterized by developers taking dedicated roles, and disconnected sub-teams working on different parts of the project.

RQ 4.3 Are there team structures that yield higher performance in processing the pull requests?

We attempt to rank the 8 dominant team structures based on the performance of the associated projects. The team structures describing well-connected teams with a small number of core contributors exhibit the highest performance.

RQ 4.4 Does changing the team structure over time have an impact on the performance of the pull-based development model?

The team structure of projects evolve over time. We compute the team structures and the performance metrics of a project at different temporal snapshots. The adoption of more desirable team structures that we identify in RQ 4.3 is strongly associated to an improvement in the performance of the pull-based development model.

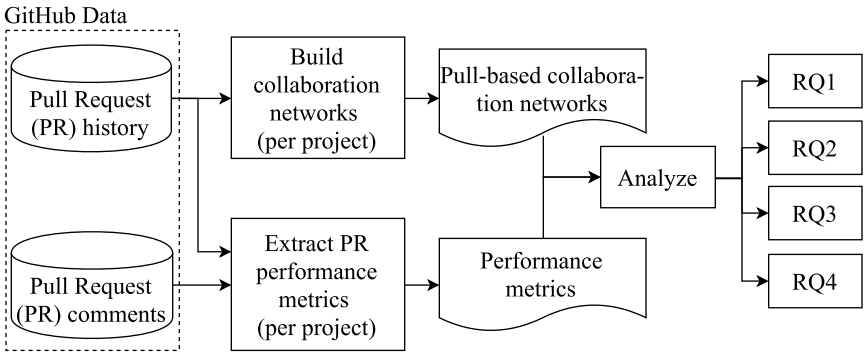


Figure 4.1: Overall approach to study the team structures formed within the pull-based networks, and their performances

4.2 Study Design

In this section, we provide details on collecting and processing the GitHub data, building the pull-based networks, and computing the network and performance metrics. Fig. 4.1 depicts our experimental setup including the overall approach.

4.2.1 Collecting the data

GitHub[1] is not only the largest code host (over 38 million repositories), but also a very popular social coding platform. GitHub provides issue tracking, pull requests, commits history, subscriptions to other users, and documentation. Developers can easily share their profile and their activities through GitHub.

To collect the GitHub data, we use GhTorrent [60], an off-line mirror of the GitHub data. GhTorrent has been collecting data since February 2012 and is updated periodically, *i.e.,* every two to three weeks. We download eight temporal snapshots (*i.e.,* 2014-01-02, 2014-08-18, 2015-01-04, 2015-08-07, 2016-02-16, 2016-03-01, 2016-06-01, and 2016-11-01) of the GitHub data dump. We intentionally keep

[1]https://github.com/

approximately a 6-month interval between each two snapshots whenever possible. The multiple snapshots enable us to study the evolution of the pull-based networks. We apply the following three filters to select the subset of subject projects:

F1. Programming language filter. We choose the projects that are written in the ten most popular programming languages on GitHub: JavaScript, Java, Python, CSS, Php, Ruby, C++, C, Shell, and C#.

F2. Type of project filter. We only extract the non-forked projects. A non-forked project is an original repository that was started from scratch, as opposite to forked projects which are copies of other repositories. At this step, we obtain over six million projects.

F3. Activity level filter. An almost equal number of projects use pull requests and shared repositories for distributed collaboration ($\sim$ 14%) [61]. The remaining projects that do not use either collaboration approches (over 60%) are single-developer projects [61]. We focus on the most active projects in terms of the number of recorded pull requests, as we need to build pull-based networks. We select the projects in the top 95% percentile, with over 100 recorded pull requests. In total, we obtain 7,850 projects with a total of 2,854,917 pull requests.

4.2.2 Building the pull-based networks

The pull-based development model has become the de facto standard of collaboration within open source projects [63]. There are two types of roles for developers to participate in a pull-based model: 1) *contributors* who make the code changes and submit the pull requests; and 2) *integrators* who are responsible to review pull requests and decide whether to merge the pull requests to the main code base. A contributor

can either be part of the project maintainers or an external developer to the project. An integrator is, on the other hand, necessarily part of the team that maintains the project. In some projects, the project maintainers can directly commit their changes to the code base; while external developers need to create pull requests to submit their changes. In other projects, the project maintainers and external developers can both solely use pull requests to submit code changes. In this case, pull requests are used to track, review, and discuss all the code changes [63].

We define a pull-based network as a directed and weighted graph. Each node represents a developer. An edge between two nodes signifies that two developers have engaged in a <contributor, integrator> collaboration. The edges of the network are weighted by the number of times that two developers collaborated in the past. We conjecture that the <contributor, integrator> relationship constitutes collaboration between two developers, because the review process of a pull request involves both a review of the code submitted, along with back-and-forth discussions to request changes if needed. The network includes all the developers who have either submitted a pull request, reviewed and integrated a pull request, or both (regardless of the developers' level of participation in the project). For each project, we build a pull-based network, as shown in Fig. 4.2. We represent the pull-based networks as a set of vectors with each vector in the form of *Contributor, Integrator, Number_ collaborations*. We show in Table 4.1 a descriptive summary of the pull-based networks of the 7,850 GitHub projects.

It is possible to mirror the network structure of development teams using other types of relationships, such as the co-editing of files or the participation in communication threads. However, in this chapter, we purposefully investigate the high level

Table 4.1: A descriptive summary of the pull-based networks extracted from the GitHub projects

Number of developers	less than 5	between 5 and 10	between 11 and 20	between 21 and 50	between 51 and 100	more than 101
# projects	1918	1848	1717	1578	525	264
Percent (%)	24.43	23.54	21.87	20.10	6.69	3.36
Avg # contributors	2.53	6.38	12.79	29.25	65.34	218.49
Avg # integrators	1.91	3.16	4.29	4.98	6.56	11.06
Avg # of developers acting as both contributors and integrators	0.97	1.79	2.47	2.76	3.98	8.04
Avg # connections	3.35	8.92	17.96	37.19	83.31	299.16
Avg # bi-directional connections	0.58	1.03	1.50	1.58	2.41	5.42
Avg weight of connections	5.56	3.98	3.15	2.50	2.23	2.35

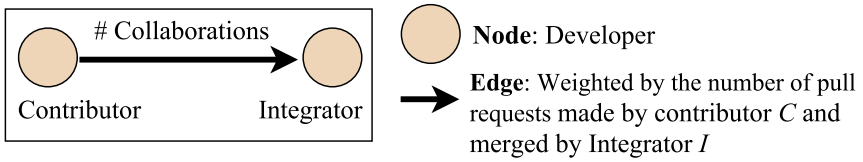


Figure 4.2: Construction of the pull-based collaboration network

structure of the development teams, as reflected by the review process of the pull-based development model (*i.e.*, the <contributor, integrator> relationship). Our goal is to infer from the constructed networks the structures of the development teams in the pull-based development model.

Table 4.2: The extracted network metrics

GLI Category	GLI	Description	Purpose
Global structure [54]	Density	The ratio of the number of edges to the number of possible edges.	Measures the sparsity of the connections in a graph.
	Reciprocity	The fraction of edges which are symmetric (reciprocal edges). It can include the null edges (*i.e.,* `reciprocity_1`) or only the mutual edges (*i.e.,* `reciprocity_2`).	Measures the likelihood of nodes in a directed graph to be mutually linked.
	Transitivity	The fraction of triangles in a graph relative to the total number of connected triples of nodes in the graph. It is computed in two ways: strong transitivity (*i.e.,* `transitivity_1`): $(i,j),(j,k) \in E \Rightarrow (i,k) \in E, for(i,j,k) \in V$); and weak transitivity (*i.e.,* `transitivity_2`): $(i,j),(j,k) \in E) \Rightarrow (i,k) \in E$) (where E is the set of graph edges and V is the set of graph vertices).	Measure the tendency of the nodes to cluster together. High transitivity means that the network contains communities or groups of nodes that are densely connected internally
Centralization [51][52]	In-degree centralization	The metric is computed at the graph level as the total deviation from the maximum observed in-degree centrality score	Measures the presence of central nodes based on incoming edges.
	Out-degree centralization	The metric is computed at the graph level as the total deviation from the maximum observed out-degree centrality score	Measures the presence of central nodes based on outgoing edges.
Informal organization [83]	Connectedness	The fraction of all nodes pairs which are not strongly disconnected (*i.e.,* there exists a path that connects the two nodes)	Describe the extent to which the structure of a graph approaches that of a tree.
	Efficiency	Essentially, the degree to which the graph uses as few links as possible to connect the nodes which are already connected in the graph. An index of the number of extra lines in the graph.	
	Hierarchy	The fraction of nodes pairs in the graph which are neither strongly connected nor strongly disconnected, *i.e.,* one node can reach the other through some path, but the other node cannot reach it.	

4.2.3 Computing the pull-based network metrics

We compute the network metrics (shown in Table 4.2) to describe the pull-based networks. From the network metrics, we can infer information such as the centralization of the developers, or how densely the developers in a network are connected. Specifically, network metrics are used to describe the structural properties of a network in its entirety [8], in terms of centralization [51, 52], informal organization [83], and general structure [54]. We compute 10 commonly used network metrics to understand the team structures in the context of the pull-based development model. We capture the team structures based on a discretization of the most influential network metrics, as detailed in RQ 4.1 and RQ 4.2. As such, we are able to systematically define the different structures formed by the developers as they collaborate through the pull-based model. Table 4.2 shows the list of the network metrics and the corresponding descriptions.

We process the pull-based networks using the *R* package *SNA* (*S*ocial *N*etwork *A*nalysis) developed by Butts [31]. The *SNA* packages transforms each network into a matrix, and provides a set of functions (*e.g.,* `grecip()`) to compute the network metrics listed in Table 4.2. Moreover, it is important to include other project measures that have shown to have strong predictive power in the previous studies [95, 96]. Therefore, we include the **number of commits** and the **number of developers** overtime, to control the impact of the activity level of a project and the size of the project team. To control the impact of the number of developers, we use a normalized metric $\frac{nodes}{edges}$, where the number of nodes is a simple count of the developers in a project, and the number of edges describes the sparsity of collaborations among the developers.

Reason for the normalization. When two networks have different sizes, it is not recommended to directly compare the values of their associated network metrics [8][31][43]. For instance, we assume that two networks N_1 and N_2 have the same centralization value C, but different sizes ($N_1 > N_2$). The centralization of a network measures the importance of the different nodes based on the number of edges. As a network grows in size, its centralization value inevitably changes as well. A centralization value equal to C is within the norm for N_1, compared to other networks of the same size. However, the same centralization C is larger than what is usual for the smaller network N_2. A prior study [8] has shown that the interaction between network metrics and the size of a network can not be ignored. Considering the intrinsic dependence on the size of a network, it is likely that the difference in the metric values can be partly explained by the difference in the network sizes. Therefore, it is important to normalize the network metrics by controlling the effect of the network size. The normalized metrics allow for a more sound interpretation of the network metric values, and a fair comparison of graphs with different sizes [43].

The CUG test for normalization. To control the effect of size, we perform the Conditional Uniform Graph (CUG) hypothesis test [8], a simple model that fixes certain properties of a network (*e.g.,* the number of nodes) at particular values, and treats all networks meeting the selected properties as equally probable. The CUG test is adequate for the task of controlling the effect of size on the remaining network metrics, as the effect of size is the only substantial effect reported by the literature [8, 31, 43]. In the CUG test, a baseline model is built and used as the null hypothesis. Under the baseline model, a number of networks of the same size are used as the input network and are simulated using Monte Carlo simulation [71].

Monte Carlo simulation shuffles edges while fixing the number of nodes to simulate the networks for the baseline model. The test generates the distribution of a network metric under the baseline model, and compares the observed network metric to the baseline distribution. To perform the CUG tests, we use *Statnet*, an *R* package developed by [71]. For each network metric value, the CUG test returns the probability of the observed value to be *greater than or equal to* the values under the baseline model (*i.e.,* $Prob_{greater} = Prob(X <= Observed)$), and the probability of the observed value to be *less than or equal to* the values under the baseline mode (*i.e.,* $Prob_{less} = Prob(X >= Observed)$).

Normalizing the metrics. To normalize the values of the network metrics, we choose to transform each metric value into $Prob_{greater}$, as we find it easier to interpret. When $Prob_{greater}$ is closer to 1, the value of the network metric is unusually high for networks of the same size. The closer $Prob_{greater}$ is to 0, the smaller is the observed value of the network metric compared to the baseline. For instance, assuming $Prob_{greater} = 0.9$ for the metric centralization in a network, we can conclude that the network is particularly centralized compared to other networks of the same size. Thus, the normalized metric values help us compare the strength of a network property across networks with different sizes.

4.2.4 Computing the pull-based performance metrics

It is important to process the incoming pull requests in an efficient and productive manner in order to maximize the benefits of the pull-based model. In previous studies on the pull-based development model [61, 148], models are built to predict the decision to merge a pull request, and the time it takes to process it.

In our study, we focus on the responsiveness of the team in processing the pull requests. Since it is not our goal to identify the factors behind a pull request acceptance, we do not consider the decision to merge a pull request as an outcome metric, but we include the time to process a pull request. Moreover, we add three metrics, *i.e.,* the ratio of long running pull requests, the number of pull requests closed daily, and the response time. We explain the performance metrics in more details below.

Productivity. We compute the following two metrics to capture the productivity of a development team. The productivity metrics are designed to assess whether the developers are able to produce the intended results, i.e. closing the pull requests, within a time period.

- *The ratio of long running pull requests.* GitHub defines a long running pull request as one that has lived for more than a month, with some activity (*e.g.,* a comment) within the past month [106]. This metric helps us assess whether the team leaves pull requests lingering for an extended period of time. The higher the ratio of the long running pull requests, the lower the productivity of the team.

- *The average number of pull requests closed daily.* The higher the average, the more productive the team is. As more pull requests are closed, more issues are fixed and more new features are introduced to the project.

Efficiency. We extract the following two metrics to quantify the efficiency of a development team. The efficiency metrics are meant to measure whether the developers process the pull requests using the least amount of resources, *i.e.,* time.

- *The average response time.* The time it takes project maintainers to provide a first response to the pull request. The sooner project maintainers provide an

initial feedback to the contributor, the more likely the contributor is motivated to work on the requested reviews to improve the quality of the code change.

- *The average processing time.* The time it takes the team to process and close a pull request. The lower the processing time, the sooner the integrators can focus on processing other pull requests, and the sooner contributors can work on new code changes.

To ensure that the performance metrics can capture distinct information, we compute the pairwise correlation among the collected metrics using the Spearman's rank coefficient. We choose Spearman's rank correlation test over other non-rank correlation tests (*e.g.,* Pearson's coefficient) because rank correlation is more robust to data that is not normally distributed [152]. For each pair of metrics, we find that the value of the Spearman's rank coefficient is always less than 0.7 (*i.e.,* the recommended threshold by [152]). Therefore, we use all four metrics to measure the productivity and the efficiency of a development team.

4.3 Study Results

RQ 4.1. What are the influential network metrics on the performance of the pull-based development model?

Motivation. In distributed software development, the social and organizational aspects have an impact on the individual and collective performance of the developers [48]. As such, the performance of the pull-based development model is governed by both technical factors (*e.g.,* the quality of the code changes), and social factors (*e.g.,* the team structure). However, it is unclear which team structure properties have the

highest impact on the performance of processing the pull requests. In this research question, we identify the network metrics (described in Section 4.2.3) that have a significant association with the performance metrics of the pull-based development model (listed in Section 4.2.4).

Approach. For each subject project, we first build a pull-based network. Second, we compute and normalize the network metrics to describe the structural properties of the pull-based network (see Section 4.2.3). Finally, we conduct the following steps to identify the influential network metrics.

Reduce highly-correlated metrics

In the presence of highly-correlated metrics, the estimate of the impact of one metric on the dependent variable tends to be less precise, thus weakening the classification model. Therefore, we use the R function `cor()` to generate the correlation matrix of the number of vertices and edges in the network, in addition to the ten network metrics. If the correlation between two metrics is more than 0.7 (*i.e.,* the recommended threshold by [152]), we select the one which is easier to interpret in the context of the pull-based development model.

Build a regression model

The purpose of the analysis is to model the relationship between the response variable (*i.e.,* the performance metrics, such as the average response time) and the predictors (*i.e.,* the network metrics, such as the density). Therefore, we use linear regression to determine which predictors are statistically significant and how changes in the predictors relate to changes in the response variable.

For each performance metric, we build a separate regression model and use the R^2 metric to assess the fit of the model. The R^2 measures the "variability explained" of

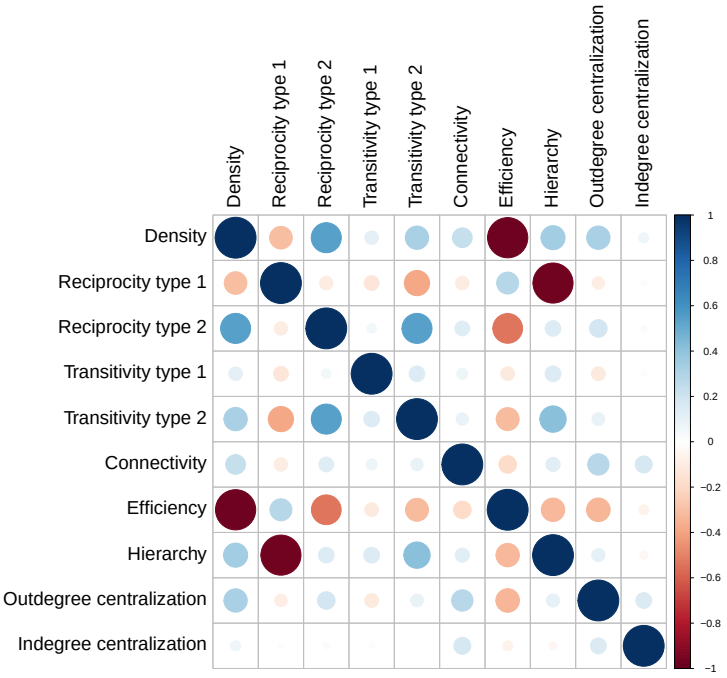


Figure 4.3: Correlation analysis of the network metrics

the response variable that is analyzed [124]. For instance, an R^2 of 0.5 indicates that 50% of the variability of the response variable is being modeled (*i.e.,* "explained") by the predictors. The remaining 50% of the variability may be due to external factors that are not being modeled or cannot be controlled. The interpretation of R^2 values depends on the analysis that is being performed. For example, when the main goal is prediction, the R^2 values should be very high (*e.g.,* around 0.7 to 0.9) [35]. Low R^2 values (e.g., around 20%) may also generate interesting insights in fields such as social sciences or psychology [18].

Identify the influential network metrics

We identify the predictors (*i.e.,* network metrics) that show the highest association with the response variables (performance metrics). The influential predictors can then be used to define the team structures. Therefore, we identify the significant predictors

(p-value < 0.05). We also report the regression coefficients of the predictors, to assess the influence of each predictor on the response variable. The regression coefficient tells us how much the response variable (*e.g.,* the response time) is expected to increase when the predictor variable (*e.g.,* the density) increases by one, holding all the other predictors constant. The regression coefficients of different predictors are not always comparable because the predictors have different types of unit. For example, the response time is measured in seconds, while the number of pull requests closed daily is counted as units of pull requests.

Findings.

The correlation analysis leads to the removal of two network metrics (*i.e.,* `hierarchy`, and `efficiency`). We show in Fig. 4.3 the results of the correlation analysis. We retain the metric density (over the efficiency) because it reflects whether developers collaborate with the entire team or only a subgroup of developers in the team. The metric efficiency measures whether the network uses as few edges as possible to connect the developers (see Table 4.2). Low network efficiency means two developers are indirectly connected more than once, which is not as easy to interpret in this context as the density. Finally, we choose the reciprocity (over the hierarchy) because the reciprocity measures whether the developers take single or multiple roles in the team (*i.e.,* contributors and integrators). The notion of `hierarchy` is not applicable in the pull-based development model because a directed edge from a contributor to an integrator does not indicate hierarchy levels, but rather collaboration.

The four most influential network metrics in terms of their association with the performance of pull-based development model include: the

Table 4.3: Regression coefficients of the significant metrics from the linear regression models

	Regression coefficients			
	Long running pull requests $R^2 = 0.25$	**Pull requests closed daily** $R^2 = 0.28$	**Response time** $R^2 = 0.32$	**Processing time** $R^2 = 0.18$
vertices_over_edges	-31.85	-	-87228.75	-
commits	0.0036	0.0041	-	16983.65
reciprocity_2	-64.99	18.62	-	-456455.56
outdegree_centralization	-0.7126	5.9078	-	-78974.6
density	-		-34455.69	-12465.85

vertices over edges, reciprocity type 2, out-degree centralization, and the density. To select the top influential network metrics, we identify the metrics that return a $p-value < 0.05$. We further report the regression coefficients of the selected metrics, to measure the influence of each predictor. Table 4.3 shows the regression coeffients of the significant predictors, for each of the linear regressions models (a model is built for each performance metric defined in Section 4.2.4). We also show in Table 4.3 the coefficient of determination R^2 of the trained models. The resulting R^2 values are low (*i.e.,* 0.25 or less), therefore, the network metrics can only explain up to 25% of the variability of the performance metrics. Therefore, the team structure properties (as measured by the network metrics) can only partly explain the productivity and efficiency of the development team in processing the pull requests. The remaining variability is likely due to other factors, such as the complexity of the code change in the pull requests and the time availability of developers. However, we can still infer interesting insights about the relationship between the network metrics and the performance metrics. For instance, an increase in one unit of the network density is associated to the decrease by 12465.85 seconds (3.46 hours) in the processing time of the pull requests. A more dense network implies a developers would collaborate

at the pull request level with diverse developers, instead a reduced number of developers. In other words, encouraging more collaboration links among the developers who process the pull requests would have a positive impact on reducing the processing time of the pull requests. Unsurprisingly, the processing time could be increased when the number of commits a projects receives is higher. Every additional commit is associated with an increase of 16983.65 seconds (4.72 hours) in the processing time. A higher reciprocity is possibly associated to lower processing time. A reciprocal link between two developers indicates prior connection between the two developers. Therefore, this result confirms a previous finding by [134] regarding the role that a contributor's prior connection to the project integrators has on the processing of the pull requests.

Pull requests could experience faster response and processing, when the developers take multiple roles and collaborate densely. It may be possible to close more pull requests and reduce the number of long running pull requests, in the presence of central developers that act as both contributors and integrators.

RQ 4.2. What are the common team structures in the pull-based development model?

Motivation. In RQ 4.1, we identify the network metrics that have the highest influence on the performance of processing pull requests. In this research question, we attempt to capture the different structures formed by the developers as they submit or review pull requests, within a large set of GitHub projects (7,850 projects). We use the term team structure to describe the self-organization of developers within the pull-based development model using the contributor and integrator relationship.

From the pull-based networks of the GitHub projects, we infer the existing frequently adopted team structures.

Approach. First, we capture the team structures within the pull-based development model from the selected GitHub projects. We then investigate the frequency of each team structure among the selected GitHub projects, and describe the most frequent team structures found in the selected pool of projects.

To identify the existing team structures adopted by the studied projects, we characterize the pull-based network associated to each project using the influential network metrics identified in RQ 4.1. We present below the interpretation of each network metric in the context of the pull-based development model.

a) Out-degree centralization: measures the importance of contributors based on the activity levels (*i.e.,* the number of submitted pull requests). High out-degree centralization indicates the existence of core contributors; while low out-degree centralization shows that the contributors participate equally.

b) Density: measures how connected the network of developers is. High density reflects a strongly connected team where the developers have prior interactions with many of their teammates. Low density characterizes teams with sparse connections.

c) Reciprocity: measures the likelihood of developers to both contribute and integrate pull requests. High reciprocity indicates that developers are more likely to take both roles (*i.e.,* integrator and contributor). Low reciprocity shows that developers are likely to have dedicated roles.

Each network is assigned a discrete representation that reflects the team structure adopted by the project, by discretizing the values of the influential network metrics.

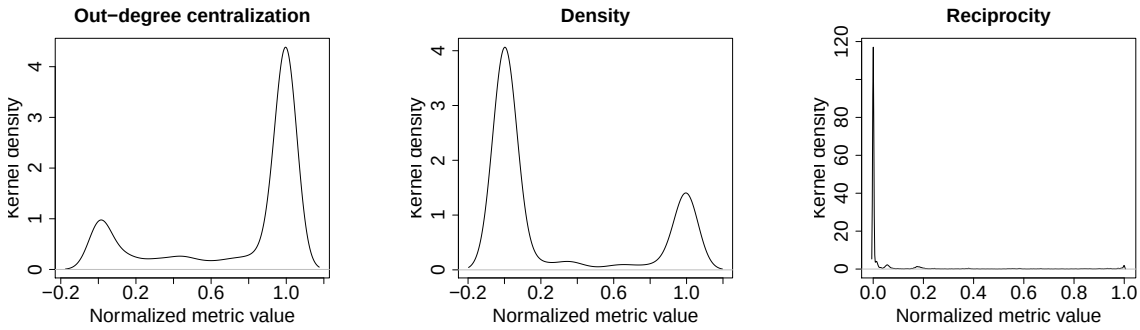
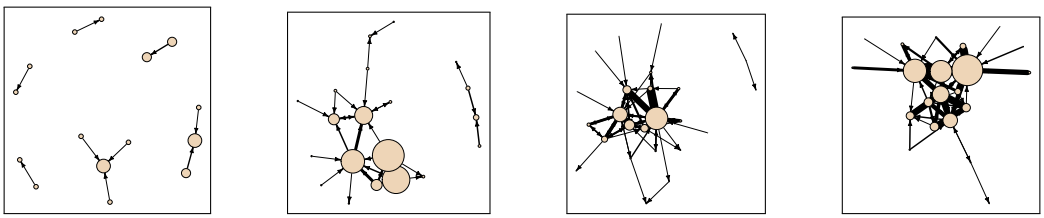


Figure 4.4: Distribution of the influential network metrics

The discretization is performed in two steps: 1) we normalize the values of the network metrics to the range 0-1 (as explained in Section 4.2.3), and 2) we perform the transformation into n scale depending on the distribution of each metric. We examine the distributions of the three selected metrics to identify the proper discretization scale. Fig. 4.4 shows the distributions of the network metrics. We observe that the out-degree centralization and the density both roughly follow a multimodal distribution; while the reciprocity follows an exponential distribution.

We discretize the two metrics **Out-degree centralization** and **Density** using a 3-level scale that captures the two local maxima and the flat area between them. The **Reciprocity** is discretized using a 2-level scale to mirror the initial peak and the flat area that follows. We compute the Spearman correlation among the discretized network metrics, and the metrics that measure the size of projects (*i.e.,* the # of developers, the # of commits, the # of LOC), and we report the coefficients of the pairs that show significant correlation (*i.e.,* $p - value < 0.05$). With the discretized network metrics, we generate 18 (3 x 3 x 2) possible team structures. Each network is assigned a team structure encoded in the form $\text{O}_{\text{utdegree}}$-$\text{D}_{\text{ensity}}$-$\text{R}_{\text{eciprocity}}$. The team structure reflects the strength of each metric (*e.g.,* reciprocity) in the network. We



(a) (b) (c) (d)

Figure 4.5: Illustrating example of project pull-based networks and their assigned team structures in the form Outdegree-Density-Reciprocity. The size of the node reflects the importance of the developers. An egde goes from a contributor to an integrator.

Table 4.4: Spearman's correlation coefficients between the network metrics and the activity metrics of the projects

	# of developers	# of commits	# SLOC
Centralization	-0.24	0.29	0.19
Density	-0.56	0.07	0.09
Reciprocity	-0.47	0.04	-

Table 4.5: The most frequent team structures shown in the form Outdegree-Density-Reciprocity.

Group	Group frequency	Team structure	Freq.	# of Projects	Median # of developers	Median # of commits	Median # SLOC
X-1-2	29.2%	3-1-2	21.4%	1680	28	1508	19433
		2-1-2	4.4%	345	21	706	13702.5
		1-1-2	3.7%	290	36	842	4839
X-1-1	39.9%	3-1-1	19.1%	1499	26	961	15439.5
		1-1-1	14.3%	1122	35	459	5288
		2-1-1	6.5%	510	28	732	9139
3-3-X	24.3%	3-3-2	18.7%	1468	9	1205.5	19757
		3-3-1	5.6%	440	8	784.5	22056

show in Fig. 4.5 illustrating examples of pull-based networks and their assigned team structures.

Findings.

There is a negative moderate correlation between the network metrics

and the size of the development team. As a team grows in size, it tends to be less centralized, less dense, and less reciprocal. Prior work has shown that centralization scores are negatively correlated with the number of developers who contributed to the bug reports [38, 78]. In the context of contributions to pull requests, our analysis result in similar conclusions related to centralization ($r = -0.24$ and $p < 0.05$). A possible interpretation of this finding is that in a large project, it might not be possible for a single developer to be involved in processing every pull request. As projects grow, they tend to become more modular, with different developers responsible for different modules. A similar finding is observed for the metric *density* ($r = -0.56$ and $p < 0.05$). This further suggests that as a project grows, it becomes challenging for a given developer to maintain collaboration with everyone in the team, and would build relationships with a reduced set only resulting in a less dense network. To conclude, the analysis of the pull-based networks confirms some of the findings resulting from other types of developer networks that are built based on different communication venues (*e.g.,* issue tracking systems).

Out of the 18 possible team structures, 16 structures exist in our selected set of projects with varying frequencies. Half of the existing team structures (*i.e.,* 8) cover more than 90% of the studied projects. We focus on the 8 most frequent team structures that account for the majority of the projects, in order to study a reduced set of team structures. Table 4.5 shows the 8 frequent team structures, their associated frequencies, and the number of projects. The 8 most frequent team structures can be grouped within three groups. We describe the three groups in the form Outdegree-Density-Reciprocity, where a metric is assigned an X if it varies within a group.

1) Sparse team with multi-role developers (X-1-2): This group includes 3 of the team structures shown in Table 4.5, with a total frequency of **29.2%**. In a sparse team (*i.e.,* very loosely connected), developers collaborate with a subset of the team only. The developers are likely to act as both contributors and integrators. Within this group, the most frequent team structure (*i.e.,* 3-1-2) describes a development team with few core contributors, who submit most of the pull requests. As shown in Table 4.5, the team structure 3-1-2 is mainly associated to the smaller projects in terms of the number of developers (*median* = 14.64), and the size of the source code (*median* = 31552.76). It is expected for the smaller open source projects to have a centralized structure, with developers participating as both contributors and integrators of the pull requests. The connections in the pull-based network of this type of team structure are sparse, as would be expected for a smaller or newer open source project. In the remaining and less frequent team structures, we observe varying levels of out-degree centralization ($\text{Out}_{\text{degree}}$[1-2]-$\text{Density}$[1]-$\text{Reciprocity}$[2]), indicating development teams that have different ratios of core contributors.

2) Sparse team with single-role developers (X-1-1): This group is the most frequent (*i.e.,* **39.9%**) and covers 3 of the team structures shown in Table 4.5. It is similar to the first group as it also describes sparse teams. However, the developers are more likely to take on a single role only, for example as integrators. Within this group, the most frequent team structure (*i.e.,* 3-1-1) describes a sparse development team with few core contributors who submit most of the pull requests, and with mostly single-role developers. The second most frequent team structure in this group (*i.e.,* 1-1-1) highlights sparse development teams with contributors who contribute equally, and take on single roles. Both team structures (*i.e.,* 3-1-1 and 1-1-1) are associated to

larger projects in terms of team size (see Table 4.5). As reflected by the reciprocity metric, the fact that developers take dedicated roles reflects the decision making process in large projects. By assigning developers to specific roles (*e.g.,* deciding or not to integrate the pull requests), the development team is more strict in its structure, in order to maintain the code quality. This observation is conformant with prior work on the characteristics of open source projects [53][108], which speculates that a set developers has more power than other developers in making executive decisions. Within this group of team structures, the centralization degree varies from projects with a more centralized power structure (similar to Linux), to more decentralized organizations, as was similarly reported by [53].

3) Well-connected team with core contributors (3-3-X): This group covers 2 of the team structures shown in Table 4.5, with a total frequency of **24.3%**. In a well connected team, the developers tend to collaborate with many team members. The team is very centralized around core contributors, who are responsible of submitting most pull requests. The most frequent team structure within the group (*i.e.,* 3-3-1) describes well-connected development teams with core contributors and single role developers. A study on the email communications between developers in the Apache project reveals that a set of core developers self organize into sub-groups that communicate intensely in completing the project [110]. This finding describes a team structure similar to the structure 3-3-X. Additionally, we find that the Apache Ignite project hosted on GitHub[2] also follows the team structure 3-3-2. Therefore, the analysis performed in prior work on other types of developer networks generates similar findings as the pull-based networks.

[2]https://github.com/apache/ignite

We capture three groups of team structures from the GitHub *projects: sparse teams with multi-role developers, sparse teams with single-role developers, and well-connected teams with core contributors.*

RQ 4.3. Are there team structures that yield higher performance in processing the pull requests?

Motivation. The 8 frequent team structures are found in over 90% of the studied projects. It is unclear if all of them are associated to differing performance of the pull-based development model. In this question, we rank the team structures in terms of the productivity and efficiency of the associated projects. Identifying the efficient and productive team structures can provide useful insights to practitioners to improve their current approach to processing the incoming pull requests.

Approach. We investigate the performance of the 8 most frequent team structures. We first identify the projects associated to each team structure. Then, we assign to every team structure the values of the performance metrics (*e.g.,* average response time) of the associated projects. For each performance metric, we rank the team structures using the associated distributions of the performance metric.

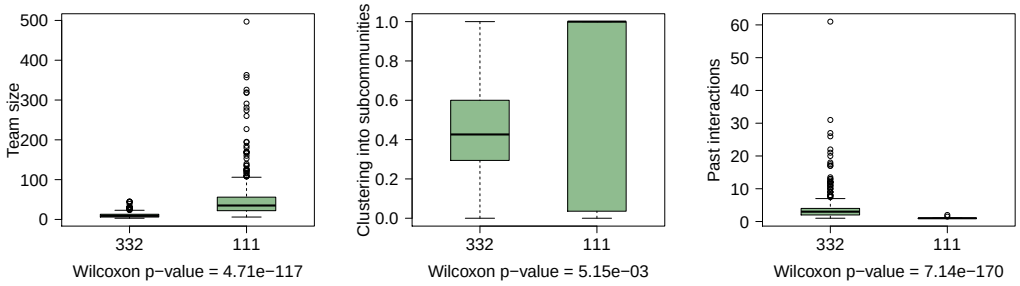
We further examine the significance of difference among the team structures. We first perform a Kruskal-Wallis test [84] on the distributions of the metric values of all the team structures. The Kruskal-Wallis test is a non-parametric statistical test to evaluate whether two or more distributions have equally large values. The advantage of using non-parametric statistical methods is that they make no assumptions about the distribution of the data. If the distributions are statistically different (p-value < 0.05), we conclude that at least one team structure is different from the others in terms of the tested performance metric. We determine which team structures are

different by performing a multiple comparison test with pairwise comparisons, and by adjusting the p-values for multiple comparisons. We specifically use the R function KRUSKALMC [119] from the package PGIRMESS. The multiple comparison test returns a significance value for each pair of team structures. The significance value indicates if one team structure outranks the other.

Finally, we attempt to examine further characteristics of some team structures associated with the highest and lowest performances. Specifically, we examine three aspects of the projects and their relation with the team structures: a) the size of the projects teams, b) the clustering of the projects networks into cliques (using the transitivity metric explained in Table 4.2), and *c)* the past interactions among the developers by computing the median weights of the edges between the developers (*i.e.,* the nodes). The higher the weight of an edge between two developers, the more interactions the two developers had in the past.

Findings.

The rankings across the different metrics show that some team structures, such as 3-3-2 and 3-3-1, are associated to higher performance of the pull-based development model, regardless of the investigated metric. Other team structures characterized by both lower density and reciprocity (*e.g.,* 3-1-1, 2-1-1, and 1-1-1) appear towards the bottom of the ranking. Fig. 4.7 shows the ranking of the team structures based on the 4 performance metrics. The significance tests confirm that some team structures (*i.e.,* 3-3-X) are significantly superior to other team structures, under all performance metrics. Table 4.6 lists the team structures that are superior in terms of all performance metrics (column 1), in comparison to the team structures shown in column 2.



(a) Team size

(b) Clustering into subcommunities

(c) Past interactions

Figure 4.6: Significant differences between the high performing team structure 3-1-2 and the low performing team 1-1-1 in terms of 3 aspects.

We observe that the most frequent team structures do not appear high in the rankings obtained based on all the performance metrics. The most frequent team structures (*i.e.,* 3-1-2 and 3-1-1), that characterize loosely connected teams with very high centralization, rank respectively in the middle and towards the bottom of the 4 rankings shown in Fig. 4.7. Our ranking results also show that the best team structures according to the 4 rankings (*i.e.,* 3-3-2 and 3-3-1) characterize projects where developers form well-connected teams, and where there are core contributors who take charge. Yet, this group of team structures that ranks high in all 4 rankings accounts for only 24.3% of our pool of projects. Based on the information shown in Table 4.5, the group 3-3-X is associated to the projects that are smaller in terms of the number of developers compared to others, but comparable in terms of the number of lines of code. We conclude that the development team is able to maintain a superior performance in processing the pull requests with a reduced set of developers. As the number of contributors grows such is the case for the team structure 3-1-1, it takes longer to respond, process, and close the pull requests.

We find that the highest (*e.g.,* 3-3-2) and the lowest (*e.g.,* 1-1-1) performing team structures are statistically different in terms of the three investigated aspects. With regards to the size of the development team, the projects that follow the structure 1-1-1 tend to be larger in size, compared to the projects belonging to the structure 3-3-2. As shown in Fig.4.6-a, the two distributions of the projects are statistically different based on the results of the Wilcoxon-Mann-Whitney test [91] (p-value < 0.05). Specifically, the median number of developers in the projects associated to the structure 3-3-2 is 9, versus a median of 35 developers in the projects following the structure 1-1-1. Therefore, despite controlling for the effect of size when defining the team structures, some team structures (*e.g.,* 3-3-2) only exist in the smaller projects; possibly because it is not possible to sustain a central core and dense interactions with the increasing number of developers. Second, we measure the clustering into cliques using the transitivity metric [141], a measure that varies from 0 when there is no clustering, to 1 for maximal clustering, which happens when the network consists of disjoint cliques. The projects in the first distribution (3-3-2) show moderate clustering (median = 0.42), indicating the existence of some cliques within the network of developers. The second distribution of projects, on the other hand, returns a transitivity median equal to 1, showing that most projects in this distribution exhibit a strong clustering into disjoint cliques (as shown in Fig.4.6-b). The stronger clustering in projects associated to the structure 1-1-1 could be possibly attributed to different reasons. One possible reason is the modularity of the code base of the projects as they grow, leading to cliques of developers focusing on specific modules of the projects. A second possible explanation is the affinity of developers to work with specific people. Finally, we investigate the difference between the two

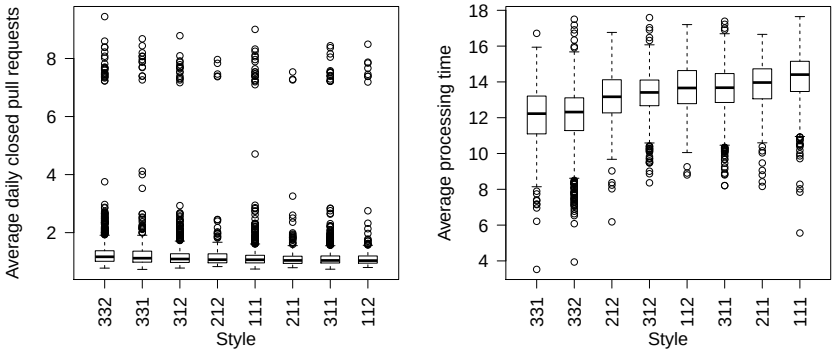
distributions in terms of the past interactions among the developers. For each project, we compute the median weight of the pull-based network edges. A weight of an edge between 2 developers that is equal to 3 indicates that the two developers interacted 3 times in the past in the review process of the pull requests. We find that the projects associated to the higher performing structure 3-3-2 show higher numbers of past interactions (median = 3), compared to a median of 1 in the second distribution (1-1-1). Therefore, the low performing team structures are more likely to experience the existence of cliques, coupled with the drive-by contributors. This finding agrees with the previous work by Joblin *et al.* [81], which also finds evidence regarding the co-existence of cliques and the drive-by contributors. The developers' networks built by Joblin *et al.* [81] are based on mailing lists and version control systems.

Therefore, we conclude that:

- One of the highest performing team structures (*e.g.,* 3-3-2) only exist in small to medium projects, despite controlling for the effect of size. It is easier for smaller teams to maintain a strong core and dense interactions, and thus achieve better processing of the pull requests.

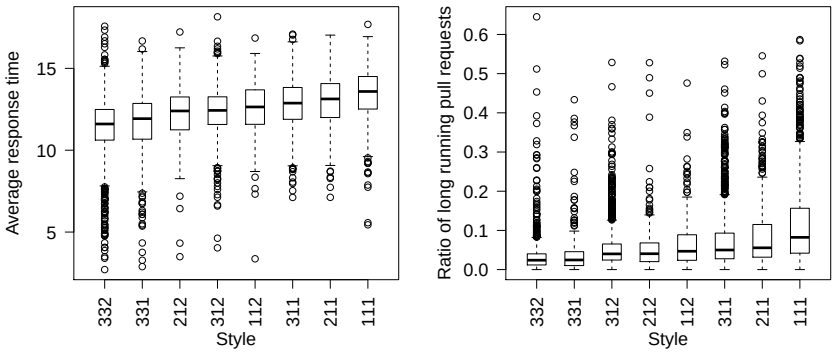
- The formation into disjoint cliques is more present in projects associated with lower performance in processing the pull requests. Projects that maintain a moderate formation of cliques achieve better processing of the pull requests.

- The processing of the pull requests could be faster due to the past interactions between the developers (*i.e.,* building of trust) in the structure 3-3-2. However, in team structures such as 1-1-1, integrators receive most contributions from drive-by and possibly unknown contributors (*i.e.,* the median of past interactions is equal to 1) .



(a) Average number of pull requests closed daily (higher is better)

(b) Average processing time (lower is better)



(c) Average response time (lower is better)

(d) Ratio of long running pull requests (lower is better)

Figure 4.7: Boxplots showing the ranking of the team structures from best to worst. The team structures are encoded as **Outdegree-Density-Reciprocity**.

The projects, characterized with a well-connected, centralized team around core contributors, are associated to higher response, processing and closing of the pull requests.

RQ 4.4. Does changing the team structure over time have an impact on the performance of the pull-based development model?

Motivation. The team structure of a team might evolve over time. We are interested

Table 4.6: Summary of the results of the pairwise comparisons. **Group 1** is the set of team structures associated to significantly higher performance compared to the team structures in **Group 2**, in terms of all the performance metrics

Group 1 (higher performing structures)	**Group 2 (lower performing structures)**
3-3-2	3-1-2, 2-1-2, 1-1-2, 3-1-1, 2-1-1, 1-1-1
3-3-1	3-1-2, 2-1-2, 1-1-2, 3-1-1, 2-1-1, 1-1-1
3-1-2	3-1-1, 2-1-1, 1-1-1
2-1-2	3-1-1, 2-1-1, 1-1-1
1-1-2	1-1-1
3-1-1	1-1-1
2-1-1	1-1-1

in studying if the evolution of the team structures has an impact on the performance of the pull-based development model. We want to examine if an improvement in the team structures is linked to an improvement in the performance of processing the pull requests.

Approach. First, we collect the GitHub snapshots at different points in time as explained in Section 4.2.1. Second, for each project, we build a pull-based network based on each temporal snapshot; thus capturing the evolution of the pull-based networks over time. Third, for each project network at time t, we assign the associated team structure based on the network metrics. Lastly, we compute the performance metrics (listed in Section 4.2.4) of each project at the different points in times to assess the improvement (or decline) of a project in processing pull requests.

To address this question, we use different time intervals (0.5, 1, 1.5, and 2 years) to study the impact of the team structures evolution on the performance of the pull-based development model. Therefore, given an interval $[t_i, t_{i+\Delta}]$, we first decide whether there was an improvement, deterioration, or no change in the evolution of

Table 4.7: The number of GitHub projects with an improvement, deterioration, insignificant change, or no change in terms of the team structure

Evolution type	**6 months**	**1 year**	**1.5 years**	**2 years**
Improvement	389 (4.95%)	452 (5.76%)	530 (6.75%)	715 (9.11%)
Deterioration	106 (1.35%)	131 (1.67%)	176 (2.24%)	240 (3.06%)
Insignificant change	212 (2.70%)	359 (4.57%)	472 (6.01%)	380 (4.84%)
No change	7,143 (90.99%)	6,908 (88.00%)	6,672 (84.99%)	6,515 (82.99%)

Table 4.8: Results of Fisher's test and Odds ratio. **P1**: Ratio of long running pull requests, **P2**: Average number of pull requests closed daily, **P3**: Average response time, **P4**: Average processing time

Interval	Impact of improved team structure on the performance			
	P1	**P2**	**P3**	**P4**
	OR (p-value)	OR (p-value)	OR (p-value)	OR (p-value)
6 months	7.22 (1.37e-12)	Inf (2.91e-04)	7.04 (2.75e-11)	5.98 (7.01e-09)
1 year	5.03 (6.36.e-05)	3.75 (1.12.e-04)	3.13 (9.18.e-06)	6.45 (7.12e-08)
1.5 years	6.17 (5.56.e-10)	2.12 (4.58e-08)	2.86 (3.45e-06)	5.12 (8.08e-10)
2 years	4.86 (7.08e-11)	2.85 (3.54e-09)	2.12 (1.12e-08)	5.69 (8.12e-08)

Interval	Impact of deteriorated team structure on the performance			
	P1	**P2**	**P3**	**P4**
	OR (p-value)	OR (p-value)	OR (p-value)	OR (p-value)
6 months	3.61 (2.11e-05)	1.76 (n.s)	3.69 (1.85e-06)	2.48 (1.89e-04)
1 year	1.22 (n.s)	2.15 (3.12.e-04)	1.55 (1.36.e-03)	1.78 (2.15e-03)
1.5 years	1.64 (3.67.e-03)	1.13 (n.s)	1.63 (9.12.e-04)	2.07 (5.64e-06)
2 years	1.78 (2.16e-03)	1.25 (n.s)	1.31 (n.s)	1.55 (3.69e-03)

the team structure. Second, we examine whether the performance of the pull-based development model has improved, deteriorated or has not changed between t_i and $t_{i+\Delta}$.

Evolution of the performance metrics:

For each performance metric, we use the values at times t_i and $t_{i+\Delta}$ to measure the change.

1. **Improvement**: If the value at time $t_{i+\Delta}$ is $X\%$ better than the value at time t_i, we consider that an improvement has occurred in terms of the given metric. We conjecture that projects of different sizes are able to achieve different

efficiency and productivity levels. Therefore, we define X based on the median improvement of similarly-sized projects.

2. **Deterioration**: If the value at time $t_{i+\Delta}$ is $X\%$ worse than the value at time t_i, we consider that a deterioration has occurred in terms of the given metric. X is set based on the median decline of projects of similar sizes.

3. **Constant**: Otherwise, we consider that no change has happened in terms of the performance metric.

Evolution of the team structure:

1. **Improvement** or **deterioration**: In RQ 4.3, we find that some team structures are associated to significantly higher performance than others at a given time t (see Table 4.6). If the team structure at time $t_{i+\Delta}$ belongs to group 1 in Table 4.6 (*e.g.,* 3-3-2) and the team structure at time t_i belongs to the list of structures with significantly lower performance in Group 2 (*e.g.,* 1-1-1), we consider the change as an **improvement**. If, on the other hand, the team structure changes the other way (*e.g.,* from 1-1-1 to 3-3-2), we consider the change as a **deterioration**.

2. **Insignificant change**: If the structure changes but the associated performances are not significantly different based on the findings of RQ 4.3 (*e.g.,* 3-3-2 and 3-3-1), we cannot decide whether there is an improvement or deterioration, and therefore, we do not include such instances in this experiment.

3. **Constant**: If the team structure has remained the same (3-3-1 at both t_i and $t_{i+\Delta}$), we consider that there is no change in the team structure.

Table 4.9: Counts of projects used to compute the Odds Ratio ($OR = \frac{Count_1 * Count_4}{Count_2 * Count_3}$)

		Team structure	
		Improved	**Unchanged**
Metric P	Improved	$Count_1$	$Count_2$
	Unchanged	$Count_3$	$Count_4$

Next, we examine the impact of a change in the team structure on the performance of the pull-based development model. Accordingly, we define the following null hypotheses:

"H_0^4a: There is no difference in the probability of projects to witness a performance improvement between projects with an improvement or no change in the team structure".

"H_0^4b: There is no difference in the probability of projects to witness a performance deterioration between projects with a deterioration or no change in the team structure".

To test H_0^4a and H_0^4b, we apply the Fisher's exact test [118]. We reject the null hypothesis if there is statistical significance (*i.e.*, p-value < 0.05). We further compute the Odds Ratio (OR) [118] to determine if a change in the performance of the pull-based development model has a higher or lower likelihood to occur for projects whose team structure has evolved. To compute the OR within a duration Δt with respect to a performance metric P, we use the counts shown in Table 4.9. For instance, $count_1$ is the number of projects that witnessed both an improvement in the performance metric P, and an improvement in terms of the team structure within the duration Δt.

Findings.

The majority of the projects (82.99% after 2 years) do not witness a

change in terms of the team structure. We show in Table 4.7 the number of projects that witness an improvement, deterioration, insignificant change, or no change in terms of the team structure. Only 9.11% and 3.06% witness a positive and negative change, respectively, after 2 years. 4.84% of the project experience a non-significant change in terms of their team structures. The low percentage of projects that experience a change towards a significantly better team structure might be an indication that the change is not necessarily a result of the natural evolution of projects. Nevertheless, it is not possible to conclude from the data at hand whether the change is conscious or coincidental.

The improvement of the team structure shows strong association with the improvement of the performance of managing the pull requests. The better team structures characterize well-connected teams that are centralized around core contributors. We find that it is highly likely for a development team whose pull-based network shows more desirable properties to improve its efficiency and productivity in managing the pull requests. Our findings (shown in Table 4.8) are consistent across the different time intervals and across the four performance metrics. For instance, the average processing time of a project is ≈ 6 times likely to decrease after 6 months if the team structure is significantly improved. Therefore, we reject the null hypothesis $H_0^4 a$, and conclude that there is a strong association between the team structures and the performance of the pull-based development model.

A worsening of the team structure is associated in many cases to a deterioration in the performance of managing the pull requests over time. In cases where the team structure of a project deteriorates (*i.e.,* displays higher sparsity and lower centralization around core contributors), we observe the likelihood

of a decline in most performance metrics. In some cases (shown in Table 4.8), the difference is not significant (*i.e.,* p-value > 0.05 and OR < 1.5), indicating no strong association between the deterioration in the team structure and the performance (especially for the average number of pull requests closed daily). The overall results lead us to reject the null hypothesis $H_0^4 b$, and conclude that it is likely for a team to witness a decline in the efficiency and productivity of managing the pull requests when the density and centralization of the pull-based network decreases. We further inspect the impact of an increasing size of the development team on the performance metrics. We find a stronger association between the increasing number of developers and the increasing processing time of the pull requests, compared to the impact of a deteriorating team structure on slowing down the processing time of pull requests. For instance, pull requests are processed 8.96 times slower (p-value = 2.45e-18) with an increasing number of developers, over a period of 2 years.

*We find that the move of a development team towards a structure that is more centralized (*i.e., *the presence of a set of core contributors), more dense (*i.e., *each developer interacts with a larger set of developers), and more reciprocal (*i.e., *more developers are involved as both integrators and contributors), is associated with a significant improvement in the speed of processing the pull requests.*

4.4 Threats to validity

This section discusses the threats to validity of our study.

Threats to conclusion validity concern the relation between the treatment and

the outcome. The performance in managing pull requests is impacted by many factors, other than the team structure, such as the turnover of the developers, and the amount and complexity of pull requests received by the project. Therefore, the performance of the pull-based model cannot be fully explained by one of the factors only. However, we are still able to obtain interesting insights from the relationship between the team structures and the performance of the pull-based development model. Our future work will consider additional factors, such as the amount and quality of pull requests received by the project, to properly model the performance of the pull-based development model. Additionally, the conclusions and recommendations presented in this study are a result from a quantitative analysis. Whether or not the recommendations presented are actionable on the side of the developers should be further validated through user studies. Related to that, other factors such as cost could have an association with how a team is managed. Therefore, a cost-benefit analysis is needed in order to measure the value of making changes to the structure of a team, depending on the cost associated to the change.

Threats to internal validity concern our selection of subject projects and analysis methods. The completeness and popularity of the team structures identified depends on the selected pool of projects. Since we study the pull-based networks, we select the projects with the highest number of recorded pull requests (over 100 pull requests). To ensure the stability of our results, we vary the threshold of pull requests used to select the projects, and find that the most frequent team structures are similar with different numbers of selected projects. As shown in Table 4.1, the majority of our subject projects (69.8%, *i.e.,* 5,483 out of 7,850) have 20 or less developers. This could indicate that our dataset could be biased towards the smaller projects. However, as

reported by [61], around 70% of the projects hosted on GitHub are single-developer projects. As such, the resulting set of projects only mirrors the widespread of the smaller projects hosted on GitHub, in terms of the number of developers. Besides, our study design attempts to control for size using both the CUG test for normalization, and by including control metrics (*i.e.,* the number of developers and the number of commits) in the models built. In our study, we consider the evolution of the team structures using temporal snapshots of the projects separated by at least 6 months. We do not study duration less than 6 months because we observe that 5% or less of the projects experience a change in the team structure in shorter durations.

Threats to external validity concern the possibility to generalize our results. GitHub is the number one social coding platform with millions of active repositories. We study the 7,850 most active projects, and therefore our findings are based on a very active pool of projects. Besides, our findings can be generalized to other open source projects using different social coding platforms, such as BitBucket, because the pull-based development model works the same across platforms. In many cases, open source projects likely follow a flat organization and agile development practices. As such, we cannot claim that our conclusions and recommendations are applicable to different types of team organizations following other development practices.

4.5 Summary

In this chapter, we build pull-based networks for the most popular projects on GitHub, and propose a way to identify the existing team structures based on a set of influential network metrics. We find that over a third of the projects are characterized by loosely connected teams, with single-role developers. Then, we attempt to rank

the different team structures and find that the most desirable structure characterizes projects where developers are well connected (*i.e.,* developers collaborate with many (if not most) of their teammates), are centralized around key contributors, and possibly take roles as both integrators and contributors. Finally, we study the evolution of the team structures along with the performance of the projects in processing the pull requests. Our findings reveal a strong association between the improvement in the team structure and the increase in the performance of the pull-based development model.

Chapter 5

An Investigation of the Use of Chatrooms in Software Development

In this chapter, we draw from the wisdom of developers to bring to light how the use of chatrooms supports the information communication among the developers. We discuss the motivation of our study, then, we describe the methodology followed to run the user study. We present the results from the study, and discuss its limitations.

5.1 Problem and Motivation

In software development, the communication and coordination of software development teams are key factors for the success of the software projects [28][39][74]. Specifically, informal communication among developers promotes project awareness [67]. However, as software development teams grow in size, it becomes harder to prevent misunderstandings in the developers' interactions. Examples of such misunderstandings are confusions regarding the software requirements, uncertainty about the right developer to contact for questions, and ambiguity about the work progress. In an attempt to address such challenges, communication channels and social media are often

integrated with the modern development tools [30] (*e.g.,* chat, email, or microblogging services) to facilitate the communication between the developers [126].

The developer chatrooms are designed to fulfill the communication needs of the developers, including messaging, file management, code sharing, and video calling. Contrary to other communication channels, such as, the mailing lists or the Q&A platforms, the developer chatrooms bring the developers together in an informal setting with equal participation opportunity for everyone. The developer chatrooms can be either "public" or "private". Public chatrooms are generally used by open source projects, and much emphasis is placed on the formation of a friendly community who talks and shares knowledge. For example, Gitter [1] is an open source chatroom geared towards the open communities. The particularity of Gitter is the possibility of a Gitter room to be associated to a GitHub repository. The private chatrooms are usually used by private software projects for team communication. Slack[2] is an example of a proprietary chatroom that has gained rapid uptake recently. It was initially designed for corporate teams, but is more and more adopted by the open communities as well. More details on the two subject systems follow in Section 5.2.

Research efforts have been invested to better understand the role of the chatrooms in software development. First, the chatrooms are considered important in supporting the informal communication among the developers [67]. In a large scale survey conducted by Storey *et al.* [126], private and public chatrooms were deemed as some of the most important channels by nearly 15 percent of the survey participants. Second, the chatrooms (in this case Internet Relay Chat) are used by developers for discussions of technical nature [72]. Lastly, the chatrooms (specifically Slack) are used

[1] https://gitter.im/
[2] https://slack.com/about

for personal, team-wide, and community-wide purposes, based on a survey with 104 developers by Lin *et al.* [86].

Despite the valuable results obtained by prior research, the developer chatrooms still need a more thorough investigation for the following reasons: *1)* The adoption of Slack since the most recent study by Lin *et al.* [86] in 2016 has more than doubled, from 3 million to over 8 million users. As such, the uses of Slack might have evolved over time; *2)* another rich community of developers has been built around the usage of Gitter chatrooms, and remains unstudied; *3)* the inclusion of both Slack and Gitter in the study lets us compare two different types of chatrooms (*e.g.,* proprietary and open source), and two different types of communities (*e.g.,* the open communities in both Slack and Gitter, and the corporate teams mostly in Slack), and *4)* beyond the uses of the chatrooms, the impact of using the chatrooms on the software projects, and the quality determinants of the chatrooms are still unclear.

We designed and conducted a survey with developers who adopt Gitter or Slack. Our survey received 163 responses. We analyzed the survey responses, using thematic analysis [27]. We further conducted follow-up interviews with 21 developers to gain more insights and validate our results. Furthermore, we mined the chat data archives from 770 Gitter rooms and 11 Slack rooms, and compute the chat activity metrics, to compare the perception of the developers with the reality. We organize our study into the following three research questions:

RQ 5.1 Why do developers use the chatrooms?

We investigate the reasons that motivate the developers to use the chatrooms to ask and answer questions. Access to experts and a fast response time are the main reasons as to why developers ask questions in the chatrooms. In return, the developers take the time to answer questions to further build the project community, and to build their own personal reputation.

RQ 5.2 What is the perceived impact of the chatroom use on the software development process?

The most recurrent reported impacts of Slack and Gitter are (respectively) managing the communication (*e.g.,* team updates) and guiding the project tasks (*e.g.,* new issues). The developers are also able to learn the best practices of the projects, and produce better solutions. Finally, there appears that the chatrooms improve the productivity of the developers.

RQ 5.3 What defines the quality of the chatrooms?

In terms of quality determinants, a 'good' chatroom is first characterized by a knowledgeable and friendly community. Besides the community, the improvement of some features (*e.g.,* search management) could help the chatrooms achieve their full potential. The majority of the surveyed developers (81.1%) report a *good* to *excellent* user experience.

5.2 Study Design

In this section, we present the methodology that we use to address our research questions. To study (i) the motives to use modern chatrooms, (ii) their impact on

the development process and (iii) the characteristics of high quality chatrooms, we design and conduct a survey that we distribute to Slack and Gitter users (*i.e.,* our respondents). We further interview a number of respondents (*i.e.,* our interviewees) to clarify and verify our findings from the survey. We also use the interviews as a means to obtain further insights. Finally, we quantitatively analyze data directly collected from Slack and Gitter to verify if the perceptions of our respondents are in agreement with the data. The bulk of our analysis results come from the survey, with the inteviews and the quantitative analysis used mainly for support of our observations.

5.2.1 Subject Systems

Table 5.1: Comparison of Slack and Gitter

	Slack	**Gitter**
Type and pricing	Proprietary and freemium (free for the basic features)	Open source and free
Intended audiences	Built for corporations, expanded to open communities.	Built for open communities, expanded to corporations.
Joining	Can only join by requesting or receiving an invite from a team.	Logging in with existing Twitter or GitHub account.
Identity	A different identity for every joined team.	Single identity - Uses Twitter or GitHub username and avatar.
Open communities	To read discussions in open communities on Slack, an invitation is required.	All rooms on Gitter have addresses like: https://gitter.im/nodejs/node that anyone can access and read. Joining a room allows you to post.
Major integrations	Google Drive, Trello, Dropbox, Box, Heroku, IBM Bluemix, Crashlytics, GitHub, Runscope, Zendesk and Zapier	GitHub, Trello, Jenkins, Travis CI, Drone, Heroku, and Bitbucket

In our work, we focus on the Slack and Gitter chatrooms. Table 5.1 shows a comparison of Slack and Gitter. Slack is a collaboration tool that offers features, such as

persistent chatrooms (*channels*) organized by topic, private groups, and direct messaging. Slack was launched in August 2013, and is reported to be "the fastest-growing business application in history" with 8 million daily active users, and 500K organizations that use the tool to collaborate and communicate [2]. Slack allows searching, indexing and integration with external applications. The basic Slack version is free, with the possibility to upgrade to advanced priced features.

Gitter is an open source instant messaging tool for developers and users of GitHub repositories, which came out of beta in 2014. Gitter enables the creation of chatrooms (called *rooms*) for GitHub repositories, i.e., each Gitter room can be linked to a GitHub repository page. Gitter has over 800K users, and 300K rooms from 100+ countries [1]. The main feature of Gitter is a seamless integration with GitHub through GitHub flavored markdowns in chat messages. Gitter has a unique activity feed showing all changes to the associated GitHub repository (*e.g.,* commenting on a pull request or closing an issue). Another integrated feature between Gitter and GitHub is the user hovercards, which are based on the GitHub profiles and statistics (*e.g.,* the number of followers). Similarly to Slack, Gitter is also integrated with other applications, such as Jenkins[3], Travis CI[4], and Bitbucket[5].

Overall, Slack is a proprietary chatroom, geared for the corporate teams but also accessible to the open communities. An important capability of Slack is the search feature of the chat data. However, the feature is limited in the free plans (*i.e.,* only up to 10K messages). An important issue in Slack is the discoverability of the communities (*i.e.,* a team can only be joined by finding its link and sending a request to join), which limits the 'openness' of Slack to the public. On the other hand, Gitter

[3]https://jenkins.io/
[4]https://travis-ci.org/
[5]https://bitbucket.org

is an open source platform suited for the open communities. Gitter rooms can be easily found and joined. However, the content is lost quite easily, as searching the past messages is not as good as in Slack.

5.2.2 Developers' Survey

In this section, we describe the process of 1) designing, 2) distributing, 3) analyzing and 4) validating our survey. We design the survey in four parts: *a)* demographics, *b)* motivations, *c)* impact, and *d)* quality determinants.

We obtained 163 responses to our survey, 114 and 44 from Slack and Gitter users, respectively. We codified the responses to perform our analysis. We conducted a set of 21 follow-up interviews to validate the results from the previous step. We provide more details of the four steps in the sections below.

1) Survey Design

We designed our survey in four main parts [6] (see Appendix B). In the first part (Survey part 1), we inquire about the respondents' **demographics** and their roles in their software projects. For example, we aim to understand if our respondents use the chatrooms from the perspective of a user, maintainer, or developer. If a respondent collaborates in many software projects, we asked him/her to answer the survey based on the project in which he/she is most active.

The remaining three parts are based on the three RQs presented in Section 5.1. In part 2 of our survey, we asked our respondents what motivates them to both answer and ask questions on Slack and Gitter. We further inquire about the instances when the chatrooms are used, instead of other channels (*e.g.,* mailing lists).

[6] https://goo.gl/forms/oX4UqWUDRcykBP372

Prior studies [120][121][126][127] report productivity concerns related to the use of the communication and social channels by the developers. Consequently, we design part 3 to ask the respondents whether the chatrooms have any impact on their productivity.

Hahn *et al.* [70] report that a developer is more likely to join a project when they have collaborative ties with the project initiator. This finding was confirmed by Casalnuovo *et al.* [33] who found that developers preferentially contribute to projects in GitHub where they have prior social links. Accordingly, we inquire whether the use of the modern chatrooms has a similar impact (*i.e.,* attracting developers to the project associated with the chatroom).

In the final part of the survey (Survey part 4), we focus on the **chatroom quality** as perceived by the respondents. First, our respondents are requested to provide a satisfaction rating of the chatrooms. Then, we inquired about the features that they like most in the chatrooms as well as the features that they believe need improvement. Lastly, we asked our respondents if they were willing to be contacted for follow-up interviews. The survey has 21 questions in total, 14 of which are open-ended questions. The estimated time to complete the survey is 20 minutes.

2) Survey Distribution

We used a combination of two methods to contact the respondents: 1) we posted our invitation in the chatrooms; and 2) we sent emails to the developers. We include the invitation letter sent to the potential participants in Appendix B.2. Before proceeding with the survey distribution, we obtained clearance from the ethics board of Queen's University. We communicated to the board the recruitment plan and the

risks associated with the two recruitment methods. To be in compliance with the ethical conduct for research involving humans, we contacted the potential respondents one time by using publicly available information. The potential respondents were informed in the invitation letter about the possibility to contact the ethics board, given any ethical concerns.

Posting in the Chatrooms: In Gitter, we manually joined the public rooms that are visible on the *explore* page of Gitter (*i.e.,* 770 rooms). In Slack, we were able to join a set of 29 public Slack rooms, by sending requests to join to the project owners. The only selection criterion to find respondents is whether they were members of the chatrooms at the time. For example, we did not consider activity levels of the respondents or membership duration, since it is our goal to obtain feedback from all kinds of respondents. We posted messages in the chatrooms to contact the respondents in a less intrusive manner. More specifically, we identified the admins of the chatrooms, whom we contacted directly and asked for permission to distribute the survey. Some admins posted the survey request themselves, while others gave us the permission to post it in one of the room channels (*e.g.,* the *random* or *general* channels). A number of admins (*i.e.,* 11) declined our request. As a result, we received 47 responses from the Gitter respondents, and another 47 from the Slack respondents, bringing the total to 94 survey responses. With our survey distribution method, we cannot assume that all the members in a chatroom have read the survey request. Therefore, it is hard to assess the response rate.

Email: Next, we contacted the developers through emails. Since we are not able to specifically email the developers that use Slack or Gitter, the selection criterion in the second phase is simply the developer's participation in a GitHub repository (*i.e.,*

code commits). The initial selection resulted in over 4 millions GitHub developers. We then chose a statistically significant random sample (confidence interval = 2 and confidence level = 95%), resulting in 2400 developers. We emailed the survey request to the selected developers, receiving an additional 69 responses. At the end of both phases of deployment, we obtained a total of 163 survey responses. The only statistical difference between the demographics of the two deployments is an increase in the number of female respondents (from 3.2% to 13.3%, p-value = 0.016).

3) Survey Analysis

After receiving the responses to our survey, we performed thematic analysis [27] to analyze the collected data. Thematic analysis is a process involving the qualitative examination of a dataset set to generate a set of themes that capture the intricacies of meaning within the data set. In the first phase of the thematic analysis (*i.e.,* coding), a set of initial codes is generated by collapsing the data into labelling concepts. In our study, a survey response can be collapsed into one or more codes, depending on the complexity and richness of the response. In the second phase of the thematic analysis, the codes are combined into overarching themes that accurately depict the data. For example, the codes '*interactive*' and '*real-time*' can be categorized under the theme '*response time*'. We show examples of the thematic analysis of our data in Appendix B.3.

The thematic analysis was performed by Mariam El Mezouar (the writer of the present thesis), and Dr. Daniel Alencar Da Costa (the postdoctoral fellow at the software reengineering lab at the time) for every survey response, until consensus was reached. Saturation is achieved after analyzing approximately 30 to 35 survey responses. The first phase of the thematic analysis took place in five coding sessions

for Gitter and four coding sessions for Slack of about 1.5 hours each. We obtained 56 and 137 codes for the survey part 2 for Slack and Gitter, respectively. An additional 67 and 122 codes resulted from part 3, for Slack and Gitter, respectively. Finally, part 4 of the survey yielded 33 and 24 codes, for Slack and Gitter, respectively. We then generate higher level conceptual themes in the second phase of the thematic analysis to answer our research questions (more details are shown in Tables 5.2, 5.4, and 5.6). In the remaining of the paper, we refer to the developers who participate in the survey as *the respondents*, and to the developers with whom we conducted the interviews as *the interviewees*. We further refer to the individual respondents/interviewees using a code of the form $S\#$ for Slack and $G\#$ for Gitter ($\#$ is a unique ID of each respondent, *e.g.,* S15). The Slack respondents span from S1 to S114, and the Gitter respondents from G1 to G48.

4) Follow-up Interviews

21 respondents expressed their interest in being part of a follow-up interview. The goals of the follow-up interviews are: (i) to clarify the responses of the survey, (ii) to validate the themes obtained from the second phase of the thematic analysis, and (iii) to collect more details from the personal experience of the developers. Our follow-up interviews follow a semi-structured format, i.e., we start with a script of questions but we let the respondents go off the script if he or she wishes to provide more information. We contact the 21 interviewees through their preferred communication channel (*e.g.,* chat, or voice call). The voice/video calls lasted in average 20 minutes, while the chat interviews lasted around 35 minutes. We provide our interview script in Appendix B.4.

5.2.3 Messages Collection from the Chatrooms

We used the Gitter REST API to download archives of the public Gitter rooms. We then use the Gitter REST API to obtain the archives of the 770 joined Gitter rooms. We exclude the 19 Gitter rooms where no conversation has started yet.

In Slack, there is no central browsing page that displays the available public rooms. Instead, each public chatroom has a unique link used to request to join the community. A thorough internet search reveals the links of the top public Slack rooms (*e.g.,* https://slacklist.info/). We were able to join 29 public Slack rooms. To collect the room archives, we used the Slack API which requires a unique token for each room. In some rooms, the token is made available to all the members; while it needs to be requested from the admins in others. In the end, we were able to collect the archives of 11 public Slack rooms.

From the chat data collected, we computed the response time of the individual chatrooms. The response time is computed as the elapsed time between two chat messages. We compute the response time to verify whether the answers from the respondents regarding the activity levels of the chatrooms are in agreement with the actual chat traffic in the chatrooms.

5.2.4 Demographics

We present an exploratory analysis of the demographics data that we collect from the responses of the respondents. Fig. 5.1 shows a summary of the demographics data collected from the Slack and Gitter survey respondents.

Gender: The majority of survey respondents (*i.e.,* 90.7%) in both Slack and Gitter identified as males.

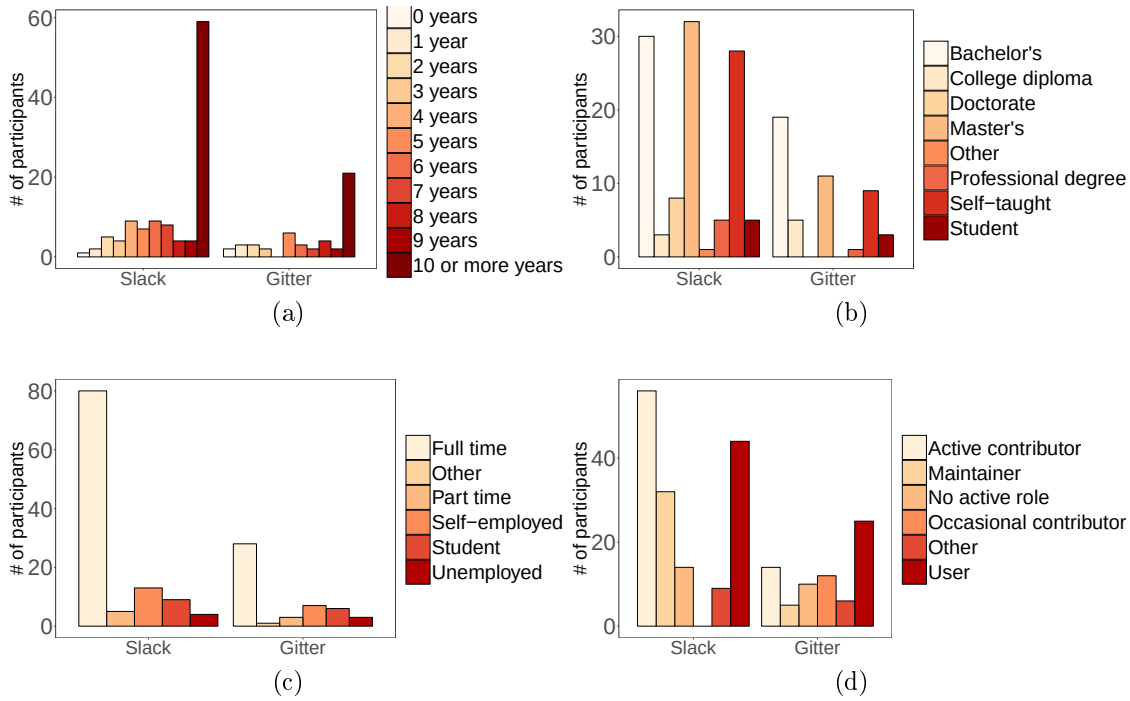


Figure 5.1: Demographics of the Slack and Gitter respondents

Development experience: A shown in Figure 5.1a, almost half of the respondents (59 out of 114 and 20 out of 48 in Slack and Gitter, respectively) reported an experience of 10 or more years.

Education: 61.4% and 62.5% of the respondents have at least a Bachelor degree in Slack and Gitter, respectively. A portion of the respondents (24.6% in Slack and 18.7% in Gitter) disclosed that they are self-taught. Overall, the majority of the developers have received a formal education in software development (Figure 5.1b).

Employment: For the employment status, 70.1% and 58.3% of the Slack and Gitter respondents, respectively, are employed full-time.

Project role: We asked the participants about their roles in the projects associated

to the chatrooms (Figure 5.1c). In Gitter, it was reported that over half of the respondents (*i.e.,* 52.1%) are users of the projects, and do not make contributions to the code base of the projects. The second most common reported role is "contributor", with a distinction between the 29.2% active contributors (*i.e.,* make frequent contributions), and the 25.0% occasional contributors (*i.e.,* making sporadic contributions) (Figure 5.1d). In Slack, the most common reported role is the active contributor role (49.1%), followed by the user role (38.6%). 28.1% and 10.4% of the respondents in Slack and Gitter, respectively, are maintainers of the projects (*i.e.,* have project privileges, such as reviewing code contributions).

5.3 Study Results

In this section, we answer our research questions by reporting the most common themes (shown in **bold**) that emerge from the second phase of the thematic analysis, as well as some of the codes (shown in *italic*) that result from the coding phase of the thematic analysis. Moreover, we compare and contrast the results from Slack and Gitter. We further include quotes from the participants to provide more insights. The Slack and Gitter participants are anonymously referred to as $S\#$ and $G\#$ (respectively), with $\#$ as the numeric identifier of a participant.

RQ 5.1. Why do developers use the chatrooms?

The first research question explores the reasons behind the participation of the developers in the chatrooms. Table 5.2 shows the full list of themes that emerge from the analysis for both Slack and Gitter. The most recurrent themes for RQ 5.1 are **the quality of the help**, **the community building**, and **the response time**.

1) Similarities between Slack and Gitter

Table 5.2: Survey results regarding the motivations of the developers to use the chatrooms.

	Examples of codes	% Times mentioned	
		Slack	Gitter
Motivations for asking questions			
Quality of help	unavailable elsewhere, better solutions	$55.5\%^{(100/180^*)}$	$39.2\%^{(47/120)}$
Response time	interactive, real-time	$26.7\%^{(48/180)}$	$18.3\%^{(22/120)}$
Community	debate, friendly	$15.5\%^{(28/180)}$	$12.5\%^{(15/120)}$
Custom support	personal projects, installation issues	$0.6\%^{(1/180)}$	$16.7\%^{(20/120)}$
Documenta- tion	privileged material, alternative documentation	$1.7\%^{(3/180)}$	$13.3\%^{(16/120)}$
Motivations for answering questions			
Community building	reciprocate help, support community	$59.4\%^{(98/165^*)}$	$48.6\%^{(36/74)}$
Personal gain	promote projects, build reputation	$31.5\%^{(52/165)}$	$29.7\%^{(22/74)}$
Company's goals	mentoring, profits	$9.1\%^{(15/165)}$	$2.7\%^{(2/74)}$
Personal enjoyment	fun, procrastination	$0.0\%^{(0/165)}$	$18.9\%^{(14/74)}$
Uniqueness of the chatrooms			
Response time	real-time, interactive	$37.8\%^{(84/222^*)}$	$48.7\%^{(38/78)}$
Community	low entry barrier, sense of equality	$16.2\%^{(36/222)}$	$21.8\%^{(17/78)}$
App features	integrations, volatile content	$30.2\%^{(67/222)}$	$11.5\%^{(9/78)}$
Quality of help	expert help, specificity	$6.8\%^{(15/222)}$	$17.9\%^{(14/78)}$
Required	internal communication, choice of employer	$9.0\%^{(20/222)}$	$0.0\%^{(0/78)}$
Topics discussed in the chatrooms			
Technical discussions	debugging, coding styles	$40.5\%^{(62/153^*)}$	$61.3\%^{(38/62)}$
Documen- tation	configuration, design decisions	$18.3\%^{(28/153)}$	$11.3\%^{(7/62)}$
Theoretical discussions	pros and cons of OOP, machine learning	$3.3\%^{(5/153)}$	$27.4\%^{(17/62)}$
Tasks	bug reports, features	$21.6\%^{(33/153)}$	$0.0\%^{(0/62)}$
Internal communication	meetings, office politics	$16.3\%^{(25/153)}$	$0.0\%^{(0/62)}$

*The total numbers of resulting codes across the survey parts (*e.g.,* 180 for asking questions and 165 for answering questions) differ because each survey question resulted in a different number of codes, since it is possible to derive one or more codes from each survey response.

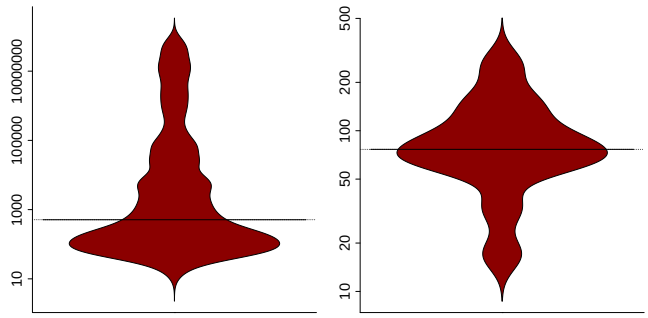


Figure 5.2: Response time distribution (in seconds) of all the chatrooms (left) vs. the chatrooms with the survey responses (right)

Quality of the help. In both Slack and Gitter, the most recurrent theme is the **quality of the help** provided by the chatrooms. The Slack participants report that asking questions on Slack provides *learning* opportunities including the *best practices*, *access to experts*, and *inside knowledge*. *S18* reports that the Slack rooms are "*where the majority of knowledgeable people in my communities are*". The Gitter participants mention that the chatrooms are useful to provide *guidance*, *clarification*, *feedback on new ideas*, and possibly *better solutions*. *G33* stated that he usually asks questions when he knows that "*there has got to be a better way / someone who made this already*". *G16* goes further and says: "*I needed advice on the architecture of my commercial apps and the gitter chat saved me a lot of money which I'd have wasted on servers if I'd gone with something different*".

Response time. The participants mention that the **response time** in chatrooms is an important motivation for asking questions in both Slack and Gitter. The participants describe the conversations in these chatrooms as 'interactive', 'conversational', and 'instant'. We also investigate the chat data of the chatrooms that the participants qualify as fast and we compute the response time (as described in Section 5.2.3). Table 5.3 shows the average response time for the chatrooms that were

perceived as having fast responses. In the chatrooms from which we receive survey responses, the median response time is 77 seconds. However, the median response time of all the chatrooms from the collected data set is 510 seconds. This indicates that our participants come mostly from active and large chatrooms. We show in Fig. 5.2 the distribution of the response time of all the chatrooms, and the distribution of the response time of the chatrooms with survey responses.

We further investigate whether the speed of response has a relationship with the number of participants in a chatroom. We find a *very weak* correlation between the two factors (Spearman's correlation = -0.17 and p-value = 8.56e-07). We conclude that the speed of response is not associated with the number of participants in a room. The number of participants has a *very strong* correlation with the number of messages exchanged in a chatroom, as expected (Spearman's correlation = 0.98 and p-value $<$ 2.2e-16).

Giving back to the community. As far as providing answers in the chatrooms goes, the most recurrent theme is **giving back to the community**. Specifically, the survey participants claim that they are eager to *reciprocate help*. Others believe that answering questions helps to *attract new contributors* to a project and *grow the community*. The communities in the chatrooms provide *low entry barrier* for the new participants, bringing a *sense of equality* among the members. Although most teams in the chatrooms establish community guidelines according to the interviewee *S10*, the moderation is considered moderate compared to other platforms, such as Stack Overflow. *G33* explains that: "*there is no competition of reputation, but rather a nicely balanced space with equal opportunity for everyone to be heard*". *G44* further mentions that there is no need to go through *pre-moderation* or restrictions based on

a *score*.

Personal gain. In addition to altruistic reasons, the survey participants report that there is some **personal gain** from answering questions in the chatrooms. 15.3% (25/163) of the participants claim that answering questions is an opportunity to *learn by teaching*, by "*validating personal assumptions, and having one's personal knowledge and code be "fact checked / sanity checked by more knowledgeable people*" (G42). Moreover, providing help is a means to *build a reputation* among peers. 11.6% (19/163) of the participants report that they answer questions to build reliance and gain respect from others in the same community. Finally, 6.1% (10/163) of the participants explain that they promote their own projects when providing help to others.

The participants mention further themes related to **the features** most appreciated in the chatrooms, such as the *volatile content*. *S11* clarifies that "*the questions on Slack are less public, disappearing eventually from free Slack rooms, such as Vapor*". In Gitter, the most appreciated feature is the seamless *integration* to GitHub, such as the inline markdown and the repository update panel. In terms of **the topics** discussed, technical discussions, such as *custom code review* and *debugging*, are common. *G27* and *G32* explain that debugging questions are usually presented as follows: "*I got unexpected results, what did I do wrong?*", or "*Is it possible to do [task X] using this library?*". According to *S14*, "*the topics span a wide technical spectrum ranging from the architectural level, to the nuts and bolts of specific technical tasks*". The participants also report discussing project documentation issues, such as *the project configuration*. *G5* explains that "*The configuration part would be solved with good*

Table 5.3: Response times (h:mm:ss) in selected Slack and Gitter rooms

Room name (G (Gitter) or S (Slack))	Avg (median) response time	Number of participants
Reaction commerce (G)	1:06:14 (0:01:39)	708
Scala on Android (G)	2:09:00 (0:01:25)	98
Ethereum (G)	0:25:49 (0:01:07)	2021
HelpJavaScript (G)	0:01:01 (0:00:17)	28246
Monogame (G)	1:32:18 (0:01:09)	218
Mithriljs (G)	0:15:00 (0:01:02)	742
Angular (G)	0:02:48 (0:00:33)	7820
Semantic Org (G)	1:23:15 (0:04:17)	1559
Sschmid Entitas CSharp (G)	1:40:33 (0:02:14)	282
Dotnet Orleans (G)	0:18:04 (0:01:03)	455
Cordova (S)	1:07:55 (0:02:38)	270
Clojure (S)	0:13:42 (0:01:25)	711

documentation, but we never found a project with good documentation", thus highlighting existing issues with projects documentation (described by the participants as *voluminous*, *ambiguous*, or *incomplete*).

2) Differences between Slack and Gitter

The sharing of expert knowledge happens for different reasons in each chatroom. For example, 17.5% of the Slack respondents (20 out of 114) mention that providing guidance to the new members over Slack is **required**, and 7.9% (9 out of 114) report that participation is encouraged to **achieve the company's goals**. In this respect, *S52* says the following: "*my senior position requires mentorship over new junior members in our team who require some attention. I want this task to be fulfilled correctly with little distraction. Thus, I am motivated to provide clear and transparent answers as much as possible, to maximize the revenue and achieve the company's goals*". On the other hand, the Gitter respondents share knowledge for **personal enjoyment**. The participants reportedly enjoy chiming in discussions about *corner cases*, or answer questions when they are *bored*, *curious*, or *procrastinating* on their

work. Another example is that Slack respondents are concerned about building their reputations to maintain "*a good employee image*"(*S41*). In Gitter, the respondents are eager to build their reputations to be recognized as experts by their peers.

Slack and Gitter are used for the ***quality of the help*** *received within a short* ***response time****. In return, time is invested by the developers answering the questions to* ***give back to the community****. Sharing knowledge in Slack is sometimes a* ***required*** *activity to discuss the projects'* ***tasks****; while discussions in Gitter tend to happen for* ***personal enjoyment*** *to discuss* ***theory and concepts****.*

RQ 5.2. What is the perceived impact of the chatroom use on the software development process?

The second research question explores the perceived impact of the chatrooms on some aspects of the development process (*e.g.,* resolution of issues). Table 5.4 shows the full list of resulting themes for each aspect, for both Slack and Gitter. The most common reported themes are **the project awareness**, **the best practices**, and **the resolution time** of issues.

1) Similarities between Slack and Gitter

Access to information. The participants claim that the most important impact is **the access to information**, such as *the best coding practices* of a project and *the peripheral knowledge*. For example, it is the opinion of *S19* that "*the chatroom discussions allow for learning the best coding practices in the Vapor project, and for prompting the developers to think of better coding approaches*". Eventually, the developers are able to produce higher quality products, according to *S12*. Furthermore, the developers are able to obtain **up-to-date resources** about a project/technology

Table 5.4: Survey results regarding the perceived impact of the chatrooms on the development process

	Examples of codes	% Times mentioned	
		Slack	Gitter
Impact on the quality of the associated project			
Project awareness	team updates, progress awareness	$51.5\%^{(65/126)}$	$18.5\%^{(10/54)}$
Project tasks	open issues, new pull requests	$27.0\%^{(34/126)}$	$48.1\%^{(26/54)}$
User support	interactive support, maintainer/user communication	$13.5\%^{(17/126)}$	$20.4\%^{(11/54)}$
Project organization	prioritization, issue filtering	$4\%^{(5/126)}$	$5.5\%^{(3/54)}$
No impact	NA	$4\%^{(5/126)}$	$7.4\%^{(4/54)}$
Impact on the quality of the developers' projects			
Access to information	best practices, peripheral knowledge	$47.4\%^{(36/76)}$	$40.0\%^{(14/35)}$
No impact	NA	$28.9\%^{(22/76)}$	$20.0\%^{(7/35)}$
Brainstorm features	planning, new ideas	$13.2\%^{(10/76)}$	$25.7\%^{(9/35)}$
Issue resolution	code reviews, bug fixing	$10.5\%^{(8/76)}$	$14.3\%^{(5/35)}$
Impact on the resolution of the projects' issues			
Resolution time	access to experts, faster resolution	$72.7\%^{(80/110)}$	$51.4\%^{(19/37)}$
Visibility	issue discovery, issue reports	$13.6\%^{(15/110)}$	$24.3\%^{(9/37)}$
No impact	NA	$4.5\%^{(5/110)}$	$21.6\%^{(8/37)}$
Project organization	issue filtering	$9.1\%^{(10/110)}$	$2.7\%^{(1/37)}$

from the chatrooms discussions. *G23* mentions the Angular project as an example of a fast growing technology, where "*the tutorials from even a few months ago may be obsolete*". In addition to the up-to-date resources, *G32* argues that "*the chatrooms' discussions contain quite a bit of 'peripheral' knowledge, such as discovering other technologies, patterns, and news*".

Help in brainstorming features. The design of the projects benefits from the feedback in the chatrooms. Specifically, *S14* explains that it helps shape new features, identify shortcomings of the design, and test pre-release versions. Similarly

to Slack, the Gitter participants report that the use of the chatrooms has helped in brainstorming new ideas and resolving issues, with a reduced effort. In this regard, we find that Slack and Gitter are similar to the mailing lists, which are also used to discuss the activities of the projects [67].

Help with issue resolution. According to *S15*, *S16*, *G7*, and *G14*, the actual **resolution time** of an issue is likely shortened because of the presence of many 'brains and eyes' to help out, and the possibility to tag the concerned developers. *S10* adds that a possible reason is that the discussions in Slack are much faster than in GitHub, where response sometimes takes several days. *S31* goes even further and states that sometimes issues that would have been reported on GitHub are not be reported at all, as they are fixed through an exchange in the chatroom. However, *S13* argues that it does not apply to the resolution time of the more complex issues. Others explain that the impact is not on the resolution time itself, but rather on **the visibility** of the issues because 'louder voices tend to win'. *S17* explains that people pressing him on Slack for an issue fix are probably going to be prioritized, simply because of exposure. *G24* reveals that several issues have been found thanks to Gitter users asking questions, specifying that these issues would have gone undiscovered otherwise. Specifically to Gitter, the panel showing the GitHub repository updates in the associated Gitter room (*e.g.,* a new comment on an issue) is important. *G17* claims witnessing a GitHub issue getting more activity because one person responded to it on GitHub, and others noticed it from the Gitter channel.

Productivity. The productivity of the developers is a much debated topic in the era of socially-enabled software development, and the developers' chatrooms are no exception. The responses from the participants reflect conflicting impact on their own

perceived productivity. 40.5% of the survey participants report a **positive** impact on their productivity, as shown in Table 5.5. The chatroom discussions reduce the 'endless trials and errors' to determine the best way to do a task. G32, a maintainer of a GitHub project, explains that '*the chatrooms offload some of the support to the community. They are especially good at helping newbies. That saves the core team cycles to invest into the project.*' However, a non-negligible portion of the survey participants (15.9%) believe in a **negative** impact on the productivity. *S26* believes that the productivity is slowed down because of the FoMo (Fear of Missing out). *G38* confirms that there is always something going on in the chat, which is an incentive to read and participate. Other survey participants (37.5%) have a more nuanced opinion about the impact of the chatrooms on the productivity (*i.e.,* **mixed or no impact**). The participants find that although productivity might be reduced, the benefits of using the chatrooms even out the damage caused. *S29* explains that the chatrooms help speed up cases where progress is stalled, however, it is sometimes misused to ask questions when an email is more appropriate. Related to that, the participants complain that the inability or difficulty to search for past discussions lowers their productivity. Overall, it seems that the use of the chatrooms requires self-discipline to avoid getting side-tracked, as explained by *S14*. Additionally, *S10* explains that it is possible to make use of the features provided in order to manage (*i.e.,*filter or mute) the chat notifications.

2) Differences between Slack and Gitter

We find that the most reported impacts are the **project awareness** and **guiding the project tasks** (*bug fixes* and *new features*) in Slack and Gitter, respectively. Project awareness is critical for the software development teams, especially when

working remotely. In Slack, the teams are able to *collaborate remotely*, *make decisions faster* in smaller teams, and *share updates*. It is also believed by *S25* that the Slack rooms help minimize the unnecessary talks and meetings for the collocated teams, and hence allow the developers to dedicate more time to the quality of the product. However, *S41* explains that the negative is that decisions are accumulated in the chatroom itself, without being organized or centrally recorded. On the other hand, the discussions in the Gitter chatrooms help the maintainers learn about the problems and wishes people have in an informal setting, so they can improve the existing functionality of the projects and generate ideas for new features (*G11*). This confirms the use of Gitter for user support within the open communities. In addition to user support, the project maintainers utilize the chatroom to quickly 'gut check' an idea before putting together a full pull request on GitHub. It is reported that the discussions on Gitter are helpful in *raising issues* (that are possibly critical) in the project associated to the chatroom. The issues are further *discussed*, and possibly *solved* through code reviews. *G14* explains that discussions on Gitter often lead to new pull requests and issues being opened and closed. However, *G21* states that important issues related to the project are discussed solely on GitHub.

Slack and Gitter reportedly help the developers to have ***access to information****, to* ***brainstorm features****, and to support* ***the issue resolution process****. However, it is perceived that Slack has more impact on improving* ***the projects' awareness****; while Gitter has an impact on* ***guiding the project tasks*** *(*e.g., *new features).*

RQ 5.3. What defines the quality of the chatrooms?

Table 5.5: Reported impact of the chatroom use on the developers' productivity (percentage^{number})

	Positive	**Negative**	**Mixed**	**No impact**	**No answer**
Slack	$39.1\%^{(45)}$	$15.6\%^{(18)}$	$33.9\%^{(39)}$	$6.9\%^{(8)}$	$6.9\%^{(8)}$
Gitter	$43.8\%^{(21)}$	$16.6\%^{(8)}$	$6.3\%^{(3)}$	$22.9\%^{(11)}$	$10.48\%^{(5)}$
Total	$40.5\%^{(66)}$	$15.9\%^{(26)}$	$25.8\%^{(42)}$	$11.7\%^{(19)}$	$7.9\%^{(13)}$

Table 5.6: Survey results regarding the quality determinants of the chatrooms

	Examples of codes	**% Times mentioned**	
		Slack	**Gitter**
Quality determinants of the chatrooms			
Community	expertise, activity level	$56.6\%^{(82/145)}$	$77.0\%^{(57/74)}$
Moderation	on-topic, admins	$37.9\%^{(55/145)}$	$18.9\%^{(14/74)}$
Features	integrations, code highlighting	$5.5\%^{(8/145)}$	$4.1\%^{(3/74)}$
Areas of improvements of chatrooms			
New features	global account, digest, discoverability	$21.9\%^{(28/128)}$	$82.6\%^{(38/46)}$
Improved existing features	bots, history management, media management	$62.5\%^{(80/128)}$	$8.7\%^{(4/46)}$
Performance	responsiveness, memory usage	$15.6\%^{(80/128)}$	$8.7\%^{(4/46)}$

To better inform the design decisions of the chatrooms, we investigate the elements that characterize the quality of the chatrooms in RQ 5.3. Table 5.6 shows the complete list of the themes that emerge from the survey analysis. The survey participants report the following ratings of Slack and Gitter respectively: *Excellent* (33.9% - 41.3%), *Good* (47.2% - 41.3), *Average* (12.3% - 17.4%), *Below average* (3.8% - 0.0%), and *Poor* (2.8% - 0.0%). The most common themes in RQ 5.3 are **the community**, and **the features**.

1) Similarities between Slack and Gitter

Community. The most reported quality determinant in the chatrooms can be summarized as follows: "a friendly, inclusive community of folks with deep technical

knowledge" (*G19*). *S11* argues that in terms of expertise, "*it is better to have a good mix of experienced users and novices*". 55.8% (91/163) of the participants stress on the fact that a welcoming and safe atmosphere is key to encourage contribution and exchange. In addition, the occasional participation of the project leaders or maintainers in the discussions sets apart the good chatrooms from others.

2) Differences between Slack and Gitter

Moderation. Based on the results shown in Table 5.6, moderation of the chat (*e.g.,* staying on topic) is twice more important in Slack than it is in Gitter.

Features. In terms of features, it appears that the Slack participants are concerned about improving the current features; while the Gitter participants request new features, such as *bots* and *history management*. In both chatrooms, it is a common complaint that knowledge is lost. In Slack, the conversations in the chatrooms under the free plans are volatile, and the search features provided are quite basic. In Gitter, the search option is almost useless according to *G23*. Consequently, a non-ephemeral history and a better history management (*e.g.,* advanced search) could be keys for the chatrooms to reach their full potential.

The quality of the ***community****, in terms of friendliness, activity and expertise are key determinants of a chatroom's quality in both Slack and Gitter. However, there is more request for* ***moderation*** *in Slack, compared to Gitter. Several* ***features*** *(*e.g., *bots and history management) could be improved in Slack and included in Gitter.*

5.4 Insights from the Interviews

In this section, we discuss further aspects that emerged from the input of the interviewees. During our interviews, we had discussions with developers who use Slack in a corporate context, and with others who use Slack as part of an open community. All of our Gitter interviewees are part of Gitter rooms pertaining to open communities. Based on the interviews, we observe that the chatrooms are used differently between the open communities and the corporate teams. Therefore, we highlight the main differences observed. Second, we report on two classes of usages that emerge from the interviews' discussions; namely, the developer support and the user support.

Open communities vs. corporate teams. To fully understand the role of the chatrooms in the participatory culture of software development, it is important to get feedback from the members of both the open communities (present in Slack and Gitter) and the corporate teams (mostly in Slack).

Reasons to participate: corporate teams are mostly **required** to provide assistance. Mentorship takes place over the chatrooms, in order to ease the onboarding of the new members. The open communities, on the other hand, display different uses of the chatrooms. First and foremost, developers are drawn to the open communities chatrooms to learn more about the project, and get custom help for specific issues. The developers are happy to reciprocate help, and consider the participation in the conversation a leisurable activity.

Impact on the associated project: when it comes to the impact on the associated project, managing the internal comunication of the team comes on top for the corporate teams. The developers report that 'pinging' a co-worker in the chatroom is less invasive than stopping by their office. Furthermore, the exchange in the chatroom

allows for passive knowledge sharing among the co-workers, an important aspect of maintaining project awareness. In an interview with *S10*, a member in a dozen Slack teams both public and private, he reports that "*company Slack usage tends to be much more structured and proactively administered, by which I mean the use of different user roles and different channel access*". The projects associated to open communities benefit from the chatrooms by attracting new contributors, who consistently participate in the chatrooms.

Developer support vs. user support. As mentioned by the interviewees, the chatrooms exhibit two classes of usage: 1) developer support, and 2) user support. The developer support is about providing guidance to the developers in learning about and potentially contributing to the project or technology associated to the chatroom. The user support happens when technical assistance is provided to the users of a software (*e.g.,* installation issues). As reported by our interviewees, the Gitter open communities are almost exclusively centered around user support. The Slack rooms, on the other hand, exhibit both types of usages even in the open communities. A Slack interviewee (*S21*) informs us that his work team has both public channels to interact with the users of the software, and private channels for internal communication. The Slack open communities also depend on the channel feature of Slack to direct the incoming questions to the right audience (*e.g.,* #team-support, #team-devops). Although 5 of the interviewed developers claim that the developer support is more common in Slack, it is not possible to verify as we do not have access to the data from the private channels.

Should a project use Slack or Gitter? The results of our study reveal that the use of the chatrooms can be valuable to: *1)* encourage participation, by providing an

informal space where incomplete questions and answers are tolerated, and *2)* support project awareness, by allowing passive knowledge sharing among the members of a team. Moreover, we find that the activity level in a chatroom is not associated (*i.e.,* has very low correlation) with the size of a chatroom. Therefore, the chatrooms could be used successfully by both the smaller and larger teams. Our study also shows that Slack and Gitter are suited for different types of audiences. Therefore, a project team should choose carefully between Slack and Gitter, since they have different strengths. Slack could preferably be adopted by project teams looking for a closed and structured communication channel. Through the use of dedicated channels, a Slack team can be organized into a company-like structure (*e.g.,* each department team has a channel). Slack allows the integration of apps that support the goals of the closed corporate teams. For instance, Nikabot is a bot that allows tracking what the team members are doing, by asking everyday: What did you work on today? The bot then gathers the information and reports to the person in charge. Therefore, we believe that Slack is very much suited for corporate teams, especially smaller teams and start-ups which are looking for a central hub to achieve their communication needs. Moreover, the permanent storage and search of discussions is a major reason to choose Slack on a paid plan over Slack on a free plan or Gitter. The strong suit of Gitter is the ease of discoverability and the openness. Therefore, Gitter is likely more suitable for communities looking to increase the adoption of their projects, and support their users. Gitter is particularly interesting for the communities that host a repository on social coding systems, such as GitHub and GitLab. Given the lack of structure in Gitter discussions (*e.g.,* no topic channels and poor search), we recommend Gitter to be used by the communities where the lost content is not an issue. Therefore, we

believe that Gitter is a good alternative for the open source communities looking to provide rapid support to their users.

5.5 Limitations

To better understand the use of the chatrooms by the developers, we opted to use a survey to reach the participants. The survey inclusion criterion in the first deployment phase is the membership in the chatrooms. This suggests a bias towards the developers that favor the use of the chatrooms, over the general population that might have differing views on the chatrooms. To mitigate this bias, we targeted a more general population of developers in the second phase of the survey (active users of GitHub). We observed an agreement between the results of the two deployment phases. In terms of demographics, we observed that half of our participants report an experience of $>$ 10 years in software development, 90.7% are male, and 100% are involved in the more active chatrooms, as measured by the number of participants (median = 709) and the number of exchanged messages within a year (median = 17836). Therefore, our findings may also be biased towards more experienced male developers participating in the active chatrooms. To offset this bias, we asked the participants in the interviews about their experiences with beginner developers, and with the use of the smaller chatrooms in terms of number of participants and messages exchanged. Additionally, almost 70% of the participants are Slack users. It was not our intention to recruit more Slack participants. However, the total number of Slack users is over 8 million; while Gitter has just over 800K users. Therefore, our population of participants reflects the popularity of each chatroom. To offset bias during thematic analysis, we performed the analysis collaboratively, and carried

discussions until consensus was reached.

Additionally, a possible limitation that may result from the design of the user study is the uncertainty about which version of Slack the survey respondents use (*i.e.,* enterprise or free). The answers might differ between the two versions, which is not captured by the current analysis.

Another possible limitation of this study is that the resulting conclusions are closely tied to the two chat services Slack and Gitter. We plan in a future study to define and describe what the ideal chat service for the developers could possibly look like, (based on the insights gained from this study), to inform and improve the design of current and future chat services.

5.6 Summary

With the increasing use of the chatrooms by the software development teams, it is important to study their impact on the development process. We design a survey to assess the developers' motivations when using two widely used developers' chatrooms, namely Slack and Gitter. We find that the chatrooms are used by the developers to exchange timely and expert knowledge, motivated by intrinsic reasons (*e.g.,* support the project community), and extrinsic reasons (*e.g.,* improve the reputations). We further attempt to identify the impact of the use of the chatrooms on the software projects. Our findings show that chatrooms reportedly impact the communication management, the development directions, and the resolution time of issues. The developers' productivity can be positively impacted by the use of chatrooms. We find that the quality determinants of chatrooms are mainly the activity levels and expertise of the community members.

Chapter 6

Learning from the Developers' Experiences Using Communication Channels To Build Software

In this chapter, we collect the developers' experiences about using a set of communication channels to communicate about a set of development activities. We draw from the developers' experiences and from an associated quantitative study a set of recommendations to help the project maintainers make informed decisions, when setting up the communication flow of their projects. We first present the motivation and setup of the study. Then, we present the study results, and synthesize the findings in a discussion section. Finally, we describe the limitations and summary of the study.

6.1 Problem and Motivation

Over the past decade, software development has transitioned from a predominantly solo activity of developing standalone applications, to a highly collaborative activity where boundaries between projects and teams are blurred [125]. This transition has been powered by the increasing popularity of socially-enabled communication and collaboration tools (*e.g.,* the social coding platforms such as GitHub), which enable

many-to-many communication through social networks. For the purpose of developing software, communication does not always imply direct interactions among the developers (*e.g.,* an exchange of e-mails). Instead, communication can occur through documentation [61][125]. For example, when a user files an issue report, developers are notified and become aware of the new issue. Although the issue report is not intended to be a direct communication between the user and a specific developer, the issue report itself *communicates* a problem to the development team. We refer to any tool that is used by developers/users to communicate any aspect of the software project as a ***communication channel***. In general, a ***communication channel*** is used to refer to both the *traditional* channels that were adopted before the advance of social media (*e.g.,* face-to-face interactions, phone), and the channels infused with *social features*, such as user profiles (*e.g.,* Twitter), the ability to *follow* (*e.g.,* GitHub), reputation systems (*e.g.,* StackOverflow), and activity feeds (*e.g.,* GitHub).

With the large variety of software project settings, a plethora of communication channels have been developed by the community. For example, co-located teams require different communication channels than internationally distributed teams [75]. Moreover, the competition in the communication channels market is another driver behind the introduction of alternative communication channels. For example, GitLab [1] is an open source alternative to GitHub. Moreover, the developers are able to collaborate using different types of communication methods each supported by diverse channels, such as chat-based synchronous communication (*e.g.,* Slack, IRC), long-format asynchronous communication (*e.g.,* email, mailing lists), and voice-based communication (Skype, Google Hangout). Given the large variety of communication

[1] https://about.gitlab.com/

channels, researchers have investigated the characteristics and benefits of communication channels. Storey *et al.* [127] show the emergence of communication channels that support networking and the sharing of community knowledge. In a later study by Storey *et al.* [125], the surveyed developers uncover the communication channels that are most essential to their work. However, the developers also report challenges in using the communication channels. For example, the use of too many channels might result in information fragmentation. In addition, face-to-face communication is still not easily mimicked by the available alternative channels. Finally, there exists a lack of awareness of when the use of certain channels are mostly appropriate (*e.g.,* when is best to use synchronous vs. asynchronous channels?) [42].

Although the choice for certain types of communication channels can be a conscious decision in a software team (as performed by the project maintainers of the The Angular JS project [2]), there exists a lack of empirical knowledge about which communication channels should be adopted given the specific necessities of a development team. To study the motivation for choosing certain communication channels, we survey 129 developers who are active in both open source and private repositories. We also derive empirical recommendations about which sets of communication channels are best suited for specific project activities. Our study is guided by the three research questions that follow:

[2]https://github.com/angular/angular.js/blob/master/CONTRIBUTING.md

RQ 6.1 Communication channels: how many is too many?

With the availability of a large variety of communication channels, the quantity and redundant information that is communicated can be overwhelming to the developers (as reported in previous studies [125]). In this research question, we aim to uncover the suitable number of communication channels for the different software development activities, based on the experiences of the developers. We find that the software development activities that require traceability of information (*e.g.,* code review) are best communicated with a fewer number of channels. When the priority is to reach an external audience (*e.g.,* recruiting developers), developers are mostly satisfied with a larger number of channels.

RQ 6.2 Social, digital, or non-digital channels: what's the interplay and is there a winner?

With the increasing adoption of the socially-enabled communication channels (*e.g.,* Slack) and the difficulty of replacing the face-to-face communication, we examine in this research question the interplay between the different categories of communication channels, *i.e.,* socially enabled and digital (*e.g.,* Github), digital (*e.g.,* mailing lists), and non-digital (*e.g.,* face-to-face). Our goal is to uncover how much of the communication should be social, digital, and non-digital to help the developers communicate effectively. Overall, we observe the highest satisfaction when combining socially-enabled (*e.g.,* pull requests) and non-digital (*e.g.,* face-to-face) communication channels.

RQ 6.3 Pick and choose: what are the combination of channels that work best?

With different channels aimed at similar purposes, we examine whether certain combinations of specific channels (*e.g.,* mailing lists + pull requests) have a positive association with the performance of the subject projects. First, we identify the most frequent combinations that are used to perform different development activities, such as bug fixing or release planning. Then, we examine whether there exists a significant association between the frequent combinations of channels and a set of project performance metrics, such the bug fixing time. We find that some combinations of channels (*e.g.,* pull requests and slack) are significantly and positively associated with the performance metrics (*e.g.,* shorter bug fixing time).

6.2 Study Design

In this section, we present the methodology used to answer our research questions. To capture the experience of the developers with communication channels, we distribute a survey to the developers (*i.e.,* our respondents). We use the replies to our survey to answer research questions 6.1 and 6.2. Afterwards, we study associations between the most frequent sets of communication channels and quantitative performance metrics from our projects, such as the time it takes to fix bugs.

6.2.1 Developers' Survey

In this section, we describe the process of 1) designing, 2) distributing, 3) and analyzing our survey.

1) Survey Design

We designed our survey in 8 main parts[3]. In the first part, we inquired about the respondents' **demographics** (*e.g.,* gender, age, and overall development experience) and their roles in their software projects. Studying the role of the respondents is important to analyze the communication channels from different perspectives (*e.g.,* user, contributor or maintainer). In the remaining seven parts, we inquired about the software development activities that require communication and collaboration among developers, as shown in Appendix C.1.

Each part contains questions that are designed to capture: *a)* the different communication channels used for each activity, *b)* the satisfaction of the developers regarding the communication channels, and *c)* 2 open-ended questions for the elaboration and justification of the experiences and satisfaction ratings. The 2 open-ended questions included in the survey were intentionally generic not to steer the respondents in any direction.

2) Survey Distribution

We targeted participants of all levels of expertise and development role to gain insights from different perspectives. For example, the users of a project might be more sensitive to problems in communicating the documentation compared to the maintainers, which are more familiar with the intricacies of the project. Therefore, our criterion to select the respondents was simply the participation in a GitHub repository through code commits.

Prior to the survey distribution, we requested and were granted clearance from the ethics board of Queen's University. We shared with the board the recruitment

[3]https://goo.gl/forms/PgZFyJrE1aGbeeTD3

plan and the risks associated with the recruitment method (*i.e.,* email). We contacted the potential respondents one time by using publicly available information, to comply with the ethical conduct for research involving humans. The potential respondents were informed in the invitation letter (shown in Appendix B.2) about the possibility to contact the ethics board, given any ethical concerns.

We first identified the developers with commit activity on GitHub, while considering all the public repositories hosted on the platform. This initial selection of the respondents resulted in over 4 millions GitHub developers. We then selected a statistically significant random sample (confidence interval = 2 and confidence level = 95%), resulting in 2,400 developers. Finally, we sent the survey request to the selected developers, resulting in 129 responses, *i.e.,* a 5.37% response rate (a comparable response rate to studies such as [40]).

We show in Figure 6.2 a summary of the demographics of our survey respondents. The majority of the respondents are males (*i.e.,* 89%) with over 10 years of software development experience (*i.e.,* 50.3%), who act as maintainers of the projects (*i.e.,* 51.2%). 67.4%$^{87/129}$ of the survey responses are associated to public repositories (*e.g.,* Apache Spark[4] and Magento[5]), 17.8% $^{23/129}$ to private repositories (*e.g.,* Thumbtack [6] and FutureAdvisor [7]), and 14.7% $^{19/129}$ are unknown (*e.g.,* no information about the repository was provided by the respondent). As such, all the conclusions that result from this study are only representative of the study population.

3) Survey Analysis

The survey contains two types of questions: a) multiple-choice questions, and b)

[4]https://github.com/apache/spark
[5]https://github.com/magento/magento2
[6]https://www.thumbtack.com/
[7]https://www.futureadvisor.com/

Figure 6.1: Geographic distribution of the survey respondents. Darker locations indicate higher participation.

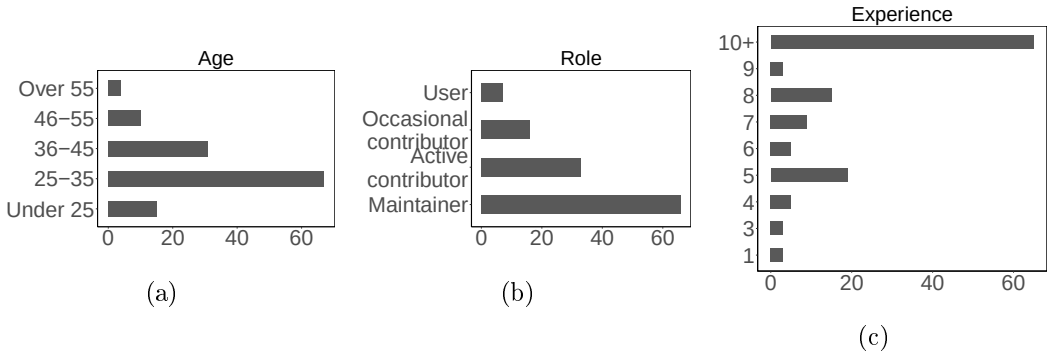


Figure 6.2: Survey respondents age, roles, and experience levels

open-ended questions. The analysis of the multiple choice questions (*e.g., What are the primary communication channels used to communicate about release planning?*) is performed using a mix of techniques, such as frequency and correlation analysis. We detail, under each research question, the approaches used to analyze the multiple choice questions.

Regarding the analysis of the open-ended questions, we use thematic analysis [27]. The thematic analysis consists of extracting *themes* that capture the meaning within

qualitative data. In the first phase of the thematic analysis (*i.e.,* coding), a set of initial *codes* is generated by collapsing the data into a set of labels that capture relevant meaning to the research questions. In our study, a survey response can be collapsed into one or more codes, depending on the complexity and richness of the response. For example, the response "*Issue tracking and pull requests are a good way to plan future additions of features, but the additional communication overhead up to a pull request happens through Skype and E-Mail*" is assigned the codes 'feature planning' and 'communication overhead'. In the second phase of the thematic analysis, the codes are combined into overarching themes that accurately depict the data. For instance, the codes '*low friction*' and '*casual exchange*' can be categorized under the theme '*informal communication*'.

The thematic analysis was performed by Mariam El Mezouar (the writer of the present thesis), and Dr. Daniel Alencar Da Costa (the postdoctoral fellow at the software reengineering lab at the time) for every survey response, until consensus was reached. Saturation is reached after analyzing approximately 50 survey responses. In the remaining of the paper, we refer to the developers who participate in the survey as the respondents. We further refer to the individual respondents using a code of the form R# (where # depicts the unique identifier of a respondent). Whenever discussing our qualitative data (i.e., RQ1 and RQ3), we support our observations by presenting quotes from the respondents and the resulting themes.

6.2.2 Software development activities

1) Definitions

We investigate the communication channels used within the projects, with respect

to the activities involved in the development process. We include in this study 8 software development activities that we believe involve most of the communications happening within a development team. We list below the development activities, and the types of communication associated to each activity:

- **Release planning** involves communicating about aspects such as the product features and the project deadlines. It also serves as a base to monitor progress within the project.
- **Bug fixing** involves discussions about issues, such as new bugs, prioritisation of bugs, bug assignment, and estimation of bug fixing time.
- **Code review** involves developers reviewing each others' source code to ensure code quality.
- **User support** involves communicating with the project's users to assist, troubleshoot, and collect issues reported by the users.
- **Recruiting developers** involves getting in touch with new qualified developers to contribute to the source code of a project. A developer could voluntarily contribute or be invited to contribute by the maintainers.
- **Project promotion** involves increasing the awareness about the project to attract more users.
- **Documentation** involves all the communications related to improving the documentation docs.

2) Performance metrics

To identify the set of communication channels that work well together, we compute a set of performance metrics in the studied software projects. We study the *best* sets of communication channels with respect to several activities, such as release planning, bug fixing, and code review. Out of the 87 public repositories associated to the survey responses, we can access the issue tracking systems, pull requests, and releases data for 58 repositories. The remaining 29 repositories do not use GitHub to track issues, manage pull requests, or create releases. We collect the following metrics:

- **Bug fixing time** captures the duration between the submission of the bug report and when the bug is resolved. Shorter bug fixing time intervals indicate that a development team is more *efficient* when fixing bugs. It is defined as:

$$T_{resolved} - T_{reported}.$$

  where $T_{resolved}$ is the timestamp a given bug is fixed, and $T_{reported}$ is the timestamp the bug is reported.

- **Bug fixing rate** captures the proportion of fixed bugs from the set of reported and closed bugs. Higher bug fixing rates suggest that a development team is more *effective* in fixing bugs. It is computed as follows:

$$\frac{\#fixed\_bugs}{\#reported\_closed\_bugs}$$

- **Merge time** is the interval between the submission of a code change and its merging to the main branch of a project. Shorter merge time values can indicate that a development team is *efficient* when reviewing code changes. We compute this metric in the following way:

$$T_{merged} - T_{submitted}.$$

where T_{merged} is the timestamp a given code contribution is merged to the main branch of a project, and $T_{reported}$ is the timestamp the code contribution is first submitted.

- **Merge rate** captures the proportion of merged code changes in the submitted code changes. A higher merge rate is a possible indicator that a development team is more *effective* when reviewing code changes. It is computed as follows:

$$\frac{\#merged\_changes}{\#submitted\_changes}$$

- **Release frequency** reflects the frequency by which a project rolls out new releases. Higher release frequency possibly suggests the *efficiency* of a development team in preparing and packaging new releases. It is measured as follows:

$$average(\forall j : \Delta(T_{release_{j+1}} - T_{release_j}))$$

6.2.3 Categories of communication channels

The communication channels used in software development can be loosely categorized as 1) non-digital, 2) digital or 3) digital and socially-enabled [127]. Non-digital media refers to the types of communications occurring through traditional channels, such as voice and paper. Digital media is an all encompassing term covering all online communications, including mobile and web. Socially-enabled media is a subset of digital media that is based on platforms where participation is transparent, in terms of the presence of a profile where the user content and interactions are visible. The digital

Category	Communication channels
Socially-enabled and digital	Pull requests Slack Gitter Google Hangouts Twitter Facebook StackOverflow
Digital	Email Mailing lists Internet Relay Chat Skype
Non-digital	Face-to-Face Telephone

Table 6.1: Categorization of the communication channels.

(*e.g.,* mailing list) and non-digital (*e.g.,* phone) channels can be easily distinguished. However, the process of labeling channels as socially-enabled is not as obvious. A communication channel is considered socially-enabled if it includes features that promote participation with transparent user profiles [41]. For instance, traditional issue tracking systems, such as Bugzilla[8], have evolved over time to include user profiles that report user activity (*e.g.,* time of last activity, number of bugs filed, and number of comments made). Another example is Google Hangouts, which we categorize as a socially-enabled channel, under the assumption that each Google user has an associated Google+ profile. The Google+ profile makes the profile, content, and interactions of the associated user transparent. Since April 2019, Google+ has been shutdown, and therefore, Google Hangouts might be re-categorized as a digital channel in future studies. For the rest, we use a similar categorization as Storey *et al.* [127]

[8]https://www.bugzilla.org/

to assign our studied communication channels to the three categories. We show the categorization in Table 6.1 for all the studied communication channels. We exclude the issue tracking systems from this categorization (unless the respondents specify which specific issue tracking system they use) because it is harder to categorize an issue track system without knowing the specificities of the used tool.

6.3 Study Results

In this section, we present our motivation, approach, and findings for each research question.

RQ 6.1. Communication channels: how many is too many?

Motivation. The breadth of communication channels used for communication, collaboration and coordination has dramatically increased over the past decade. Notwithstanding the number of available communication channels, little research has been devoted to studying the experience of developers when using certain types of communication channels. Therefore, based on the experience of developers, we aim in this RQ to identify whether the different development activities (*e.g.,* bug fixing and release planning) require a lower or higher number of communication channels. This investigation is important to better inform team leaders of the number of communication channels that may make the development team more productive.

Approach. In the first part of this research question, we explore the reasons that could possibly impact the choice of the number of communication channels to be used. To do so, we analyze the survey responses using thematic analysis, as described in Section 6.2.1, and we present the resulting themes.

Second, we examine each development activity in terms of the number of channels

used by the different projects. We report statistics on the number of channels used in the studied projects. We also examine whether there exists a significant difference between the different development activities in this regard. To compare how the number of communication channels changes based on development acitivity, we use the Scott-Knott test [79], (with p-value < 0.05). The Scott-Knott test is a statistical multi-comparison procedure based on cluster analysis. The Scott-Knott test sorts the number of communication channels used for the different development activities. Then, it groups the development activities into two different groups that are separated based on their mean values (*i.e.,* the mean value of communication channels used for each development activity). If the two groups are statistically significantly different, then the Scott-Knott test runs recursively to further find new groups; otherwise, the development activities are put in the same group. In the end of this procedure, the Scott-Knott test comes up with groups of development activities that are statistically significantly different in terms of their number of communication channels used.

Moreover, within each development activity, we further compare the open and private repositories, and test the following null hypothesis:

H_a^1*: The open and private repositories use a comparable number of channels to communicate, coordinate and collaborate in the different development activities.*

To test the hypothesis, we use the Mann-Whitney U test [91] and compare the distributions of number of channels used in the open and private repositories for each development activity. To quantify the magnitude of the possible difference, we calculate the effect size using the Cliff's delta [111]. The Cliff's delta is a non-parametric measure, of which the value ranges from -1 (*i.e.,* all values in the first distribution are higher than the second) to +1 (*i.e.,* all values in the first distribution

Table 6.2: Scott-Knott test results when comparing the number of channels used in each development activity, divided into distinct groups that have a statistically significant difference in the mean (p-value < 0.05).

Group	Activity	Number of channels used			
		Mean	Median	Min	Max
G1	Release planning	2.9	3	1	9
G2	Bug fixing	2.4	2	1	7
	User support	2.1	2	0	8
G3	Documentation	1.9	1	1	9
	Recruiting developers	1.8	1	1	9
G4	Project promotion	1.6	1	1	8
	Code review	1.5	1	1	6

are lower than the second). We interpret the effect size eff as small for $0.147 < eff < 0.33$, medium for $0.33 < eff < 0.474$, and large for $eff \geq 0.474$, by following the guidelines of prior work [59][132][37].

Secondly, we examine the satisfaction of developers depending on the number of channels used. As recommended by Sullivan *et al.* [129], we use a stacked barplot (showing the percentages of responses in each satisfaction category) to visualize how the satisfaction of the developers changes. We show how the developer satisfaction changes based on the number of channels used, for each investigated development activity. We also show the overall change of developers' satisfaction, regardless of the development activity. We report our observations on the most suited number of channels for each analyzed scenario. We support and justify our observations using the results from the thematic analysis of the survey responses. With respect to the survey responses, we use quotes from our respondents whenever needed.

Results.

We identify five different factors that could impact the choice of the

Table 6.3: Results of the survey thematic analysis with respect to the number of communication channels used (↗ = higher number of channels, ↘ = lower number of channels)

Themes	Rationale and *examples of codes*	% mentioned*
Reach	↗ number of channels allows to *notify* and *reach* all interested/involved parties from both the developer and non-developer communities, when needed (*e.g.,* to announce a new release)	43%
Standards	The decision on the number of channels can be either a *personal preference*, a *company standard*, or a *customer preference*.	41%
Traceability	↘ number of channels is preferred when *tracking context*, *monitoring progress*, and *identifying accountability* are essential to the task (*e.g.,* who is the owner of this code? or what is the reasoning behind this decision?)	37%
Organization	↗ number of channels is opted for when it is important to *separate artifacts* (*e.g.,* informal discussions about an issue and the formal issue report), and for discussing design decision with *different granularities* (*e.g.,* slower longer term decision on issue trackers vs. fast and small decisions on chatrooms)	19%
Not needed	In some cases, the best form of communication is no communication at all when automated processes are put in place (*e.g.,* when continuous integration and continuous delivery is used).	11%

*The total percentages associated to the themes do not add up to 100%, because a survey response could result in zero or more themes, depending on how rich the response is.

number of channels used in the projects. The thematic analysis performed on the developers' responses to the open ended questions results in the following overarching themes with regard to the number of channels used: 1) reach, 2) standards, 3) traceability, 4) organization, and 5) not needed. We show in Table 6.3 the list of themes, along with the rationale behind each theme, examples of codes, and the number of times a theme appeared in the survey responses.

The development activities require a number of communication channels ranging from 1.5 to 2.9, in average. The maximum reported number of channels used is 9, appearing in release planning, documentation, and recruiting developers. The mininum number of channels used that was reported by our respondents is 0, appearing in user support when no support is provided. The 7 development activities can be divided into 4 statistically significantly different groups, as indicated by the Scott-Knott test. Table 6.2 shows the significantly different groups of development activities, along with descriptive statistics of the number communication channels used for each activity. Therefore, based on the developers' feedback about 129 projects, it appears that different development activities require slightly different number channels for information propagation and collaboration.

We find no significant difference in terms of the number of used channels in the open and private repositories for most of the development activities. The Mann-Whitney U test returns non-significant p-values (*i.e.,* > 0.05) when comparing the distributions of the number of used channels among the different development activities (Figure 6.3). Therefore, we cannot reject the null hypothesis H_a^1. The bug fixing is the only exception where the $p - value$ is equal to 0.015, and

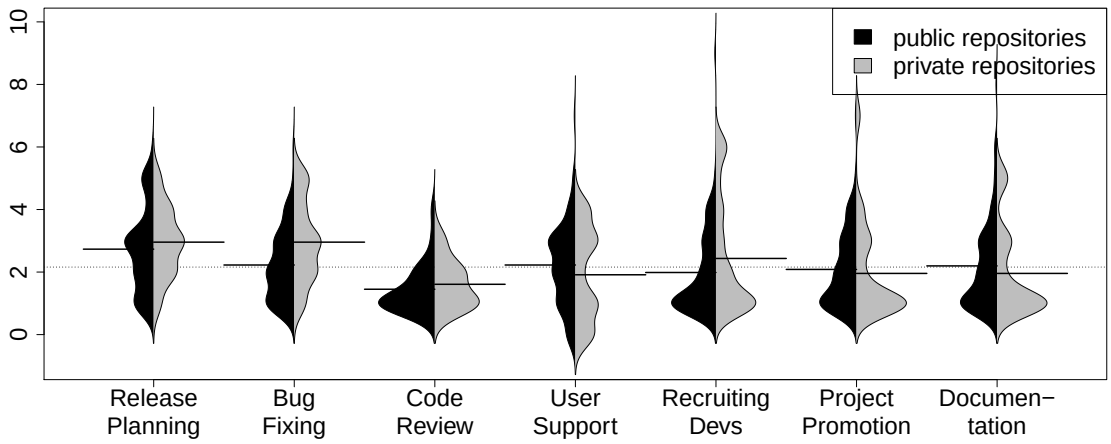


Figure 6.3: The distributions of the number of communication channels used by the projects grouped by the repositories types: public (shown in gray) and private (shown in black).

cliff's delta returns a *small* significant effect size. In the bug fixing activity, the private repositories tend to use slightly more channels ($mean = 2.97$) compared to the open repositories ($mean = 2.21$).

We observe that the number of communication channels associated to a higher satisfaction differs across development activities. We show in Figure 6.4 the percentage of responses in each satisfaction category for every development activity and overall. We can observe in Figure 6.4 that in many cases the higher number of channels are associated to higher satisfaction. The reason behind this observation is that the number of respondents who have reported the use of a higher number of channels (*e.g.,* 9) is much lower (*e.g.,* between 1 and 3), and therefore it is more likely to observe higher satisfaction and agreement among a small number of respondents. In interpreting the results in Figure 6.4, we take the aforementioned

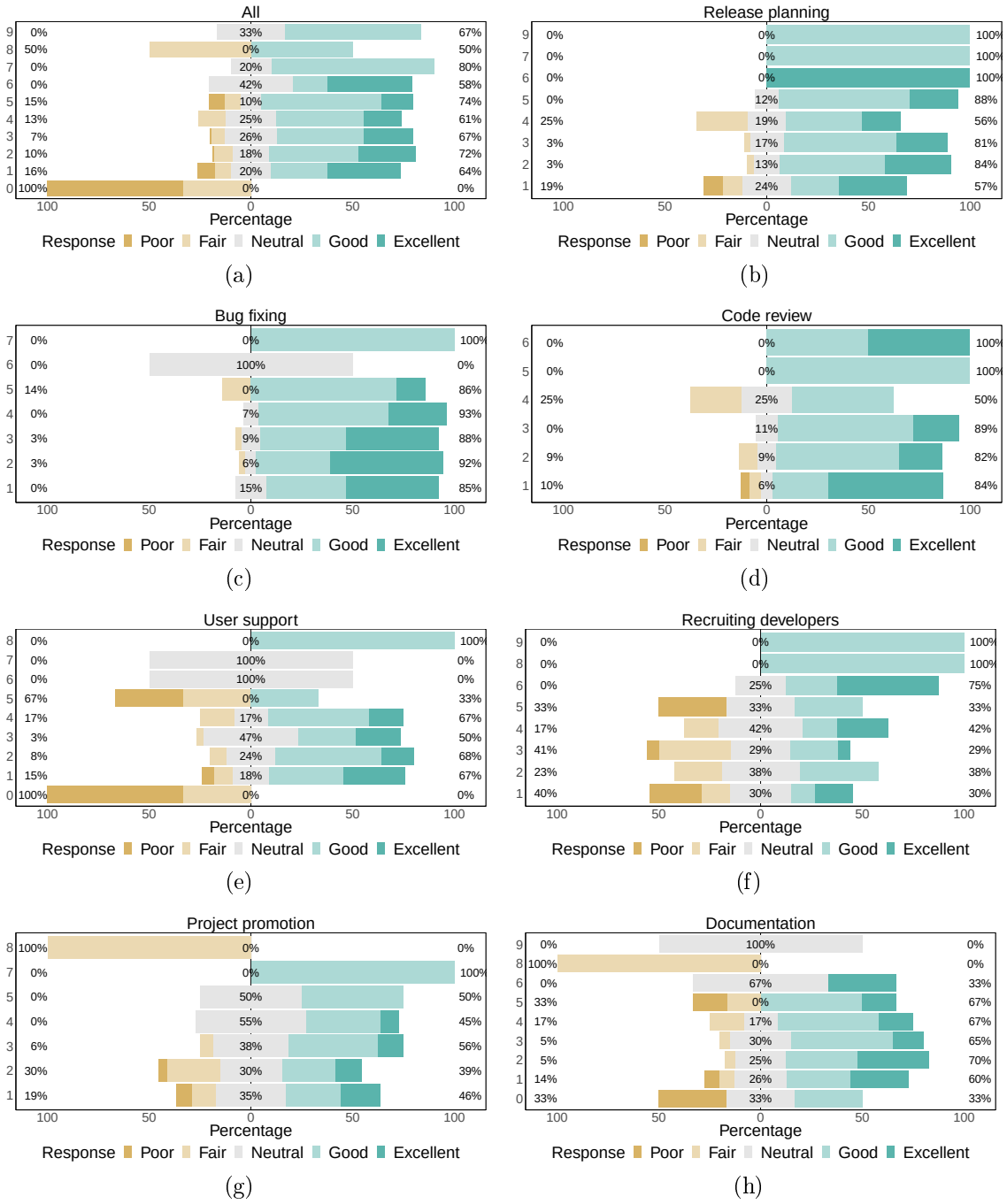
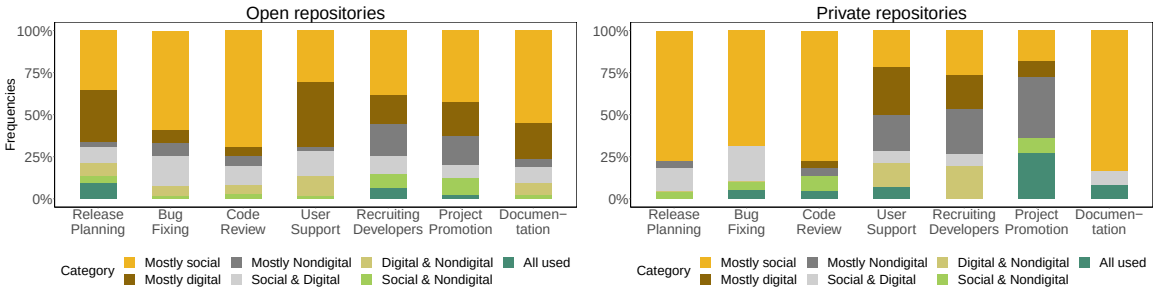


Figure 6.4: Percentages of responses (x-axis) in each satisfaction category with respect to the number of channels used (y-axis)

observation into consideration. When performing tasks, such as recruiting developers, we observe that a higher number of communication channels (*i.e.,* 6) is preferred by developers, as shown in Figure 6.4-f. Although in many cases potential contributors in open source projects are self-selecting, we conjecture that when recruitment is needed, the process requires more aggressive information propagation to reach as many candidates as possible. Indeed, *R11* confirms that "*Finding people who really care about open source (and are available) is hard, so we need wide reach to gather a valid pool*". In terms of release planning, *R75* recommends "*to announce releases in multiple places to hit large chunks of users, who have different preferences for how to receive communications)*". On the other hand, the communication related to activities, such as user support and documentation, can be satisfied with as low as 2 communication channels. In regard to user support, a high number of communication channels may reduce satisfaction. The respondent *R10* elaborates on the usage of 4 channels for the user support activity: "*customer feedback needs to percolate through several communication walls before it reaches developers, which is not very effective.*" With respect to the code review activity, we observe that a lower number of channels is also prefered (*i.e.,* 3). With a lower number of channels, *R7* explains that "*all the discussion can be captured in one place and comments/approvals are added easily*". *R22* further reports that a lower number of channels "*reduces overhead of communication and allows clear traceability of the information*".

Based on the developers' satisfaction, the communication regarding user support and documentation is mostly satisfied with fewer channels. Recruiting developers and release planning likely require a higher number of channels to ensure a higher audience reach.



(a) (b)

Figure 6.5: The frequency of the different categories of communication channels by development activity, in open (*shown on the right*) and private (*shown on the left*) repositories.

RQ 6.2. Social, digital, or non-digital channels: what's the interplay and is there a winner?

Motivation. An increasing number of software development projects have been adopting social features (*e.g.,* user profiles and presence awareness in chat) in their development activities. Nevertheless, digital (*e.g.,* email) and non-digital (*e.g.,* face-to-face) types of communication channels are still important in the collaboration among the developers. Investigating the interplay between social, digital and non-digital communication channels is important towards understanding the best combination of communication channel types. For example, the use of too many social features may be distracting for developers, while relying only on non-digital communication may be almost impossible in several distributed project settings where communication persistence is required.

Approach.

We assign to the studied projects (129 in total from the responses) the counts of social-enabled, digital, and non-digital communication channels for each of the 7

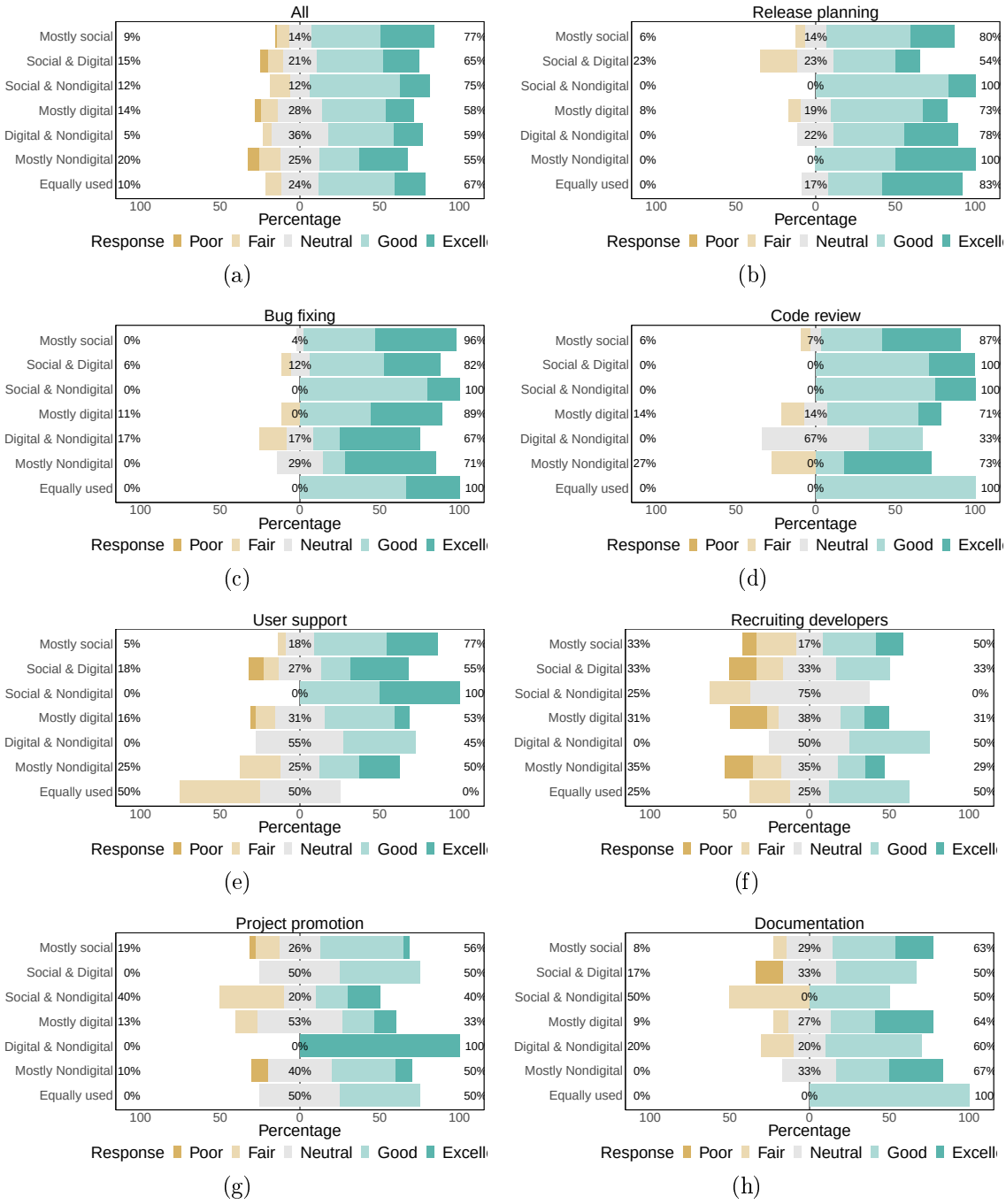


Figure 6.6: Percentages of responses (x-axis) in each satisfaction category with respect to the category of channels used (y-axis)

Table 6.4: Results of the survey thematic analysis with respect to the category of the communication channels used

Category	Associated themes	% times mentioned*
Socially-enabled	Informality	61%
	Fast turnaround	44%
	Wide reach	29%
Digital	Widely-accepted/Simple	76%
	Targeted recipients	57%
	Easy moderation	25%
Non-digital	Optimal	74%
	Conflict-free	45%
	Volatile	22%
	Unfeasible	12%

*The percentages total associated to the themes for each category do not add up to 100, because a survey response could result in zero or more themes, depending on how rich the response is. In some cases, no justification of the channel is provided by the respondents

Table 6.5: Example of the categories assigned to the project *snapcraft.io*. C1: count of socially-enabled, C2: count of digital, and C3: count of non-digital channels

Development activity	C1	C2	C3	Assigned category
Release planning	1	2	1	Mostly digital
Bug fixing	1	1	0	Social and digital
Code review	1	0	0	Mostly Social
User support	1	1	0	Social and digital
Recruiting devs	1	1	1	Equally used
Project promotion	2	1	0	Mostly Social
Documentation	1	3	1	Mostly digital

investigated development activities. We show an example for the project *snapcraft.io*[9] in Table 6.5. To capture the dominating category of communication channel used in a development activity, we further assign a final category to each activity in every project (*e.g.,* mostly social, or equally social and digital). This final assignment is

[9]https://github.com/snapcore/snapcraft

based on the values C1, C2, and C3 shown in Table 6.5. We further report on the themes, obtained from the thematic analysis, which justify the choice of the different categories of channels in Table 6.4.

To investigate the interplay between the categories of communication channels, we examine the frequency of the assigned categories. First, we study the dependence between the development activities (*e.g.,* release planning) and the assigned categories (*e.g.,* mostly social). We test the following null hypothesis:

H_a^2*: The chosen category for a communication channel does not depend on the development activity.*

To test the hypothesis, we use the Chi-squared test of independence [101] between the development activities and the assigned category for the communication channels. The purpose is to investigate whether the different development activities have different requirements in terms of the dominating category of channels used. For instance, does the code review process require more social features to be effective? Indeed, social features have been correlated with a quicker integration of pull requests [154]. As for the user support activity, it may be best handled with a combination of social, digital, and non-digital to accomodate all types of users. For example, some users could be best assisted over the phone, when immediate attention is required.

Finally, we compare the frequencies of the assigned categories between the open and private repositories. We test the following null hypothesis:

H_b^2*: The open and private repositories follow similar strategies when choosing the categories of communication channels.*

To test H_b^2, we use the Wilcoxon signed-rank test [144], a non-parametric statistical test used to perform paired comparisons between samples.

Similarly to RQ1, we further examine the developers' satisfaction with respect to the dominating category of channels used by the projects. We use a stacked barplot to visualize how the satisfaction of the developers changes with the different categories of channels adopted by the developers.

Results.

We find that there is a dependence between the development activities and the types of communication channels used by the projects. Overall, the socially-enabled communication channels are the most used for the development activties, except for the user support. Figure 6.5 shows the frequencies of the different categories, with respect to the development activities and the repository type. The Chi-squared dependence test confirms the dependence between the development activity and the category of channels, for both the open (p-value = 1.10e-06) and private (p-value = 2.85e-05) repositories. Therefore, we reject the null hypothesis H_b^2. The code review process is the most (69.4%) and second most (77.3%) 'social' activity in the open and private repositories, respectively. Indeed, the social features especially enable a transparent identity, which is important when reviewing code as it promotes accountability. The digital channels are the most frequent in the user support process for both types of repositories, as reported by 30.8% and 28.5% of the respondents for public and private respositories, respectively. In the second place comes the socially-enabled channels (for both repository types), and the non-digital channels (for the private repositories only). As conjectured, the user support process needs to accomodate the needs of the users, by adapting the choice of the communication channels to the users' preferences.

The open and private repositories do not significantly differ in terms

of the categories of channels that are used in the development activities. The frequencies of the different categories are shown in Figure 6.5. The Wilcoxon signed-rank test returns an non-significant p-value (0.47), when comparing the two distribusions of frequencies. The open and private repositories follow different governance dynamics. One difference is the incentive behind the contributions, which is voluntary in open repositories and paid in private repositories Another difference is the intellectual property, as the open repositories are open source, while the private repositoies are proprietary. Related to that, privacy concerns also differ across the two types of repositories, with open repositories offering full transparency and private repositories adopting strict access control mechanisms. Nevertheless, our findings suggest that the amount of the communication performed through social, digital, and non-digital channels is actually comparable between the open and private repositories. Therefore, we cannot reject the null hypothesis H_b^2.

The main exception that can be observed in Figure 6.5 is the absence of the category *"mostly digital"* under three development activities in private repositories, including release planning, bug fixing, and documentation. Instead, we observe the prominent presence of socially-enabled channels in the release planning, bug fixing, and documentation activities. Our results suggest that private repositories are more inclined to socially-enabled channels.

The combination of non-digital and socially enabled communication channels occur to be the best in most of the development activities. Although it is most common among our subject projects to use mostly social communication channels, the developers express the highest satisfaction when using a combination of socially-enabled and non-digital channels, as shown in Figure 6.6.

The non-digital channels, specifically face-to-face and the phone, are mostly appreciated because they culminate in *low friction* (*R27*), and *little to no misunderstading* (*R46*). For projects that equally use the socially-enabled and non-digital channels, the socially-enabled channels are the second best alternative to communicating face-to-face, as they support informal communciation, and fast turnaround.

We observe a significant dependence between the development activities and the categories of communication channels. This result highlights the fact that some activities require more social features than others. Moreover, our results reveal that, for the majority of development activities, the most satisfactory combination of communication channels is the equal use of socially-enabled and non-digital communication channels.

RQ 6.3. Pick and choose: what are the combination of channels that work best?

Motivation. In the previous research question, we study the interplay between the categories of the communication channels and identify which are most satisfactory for the developers. In this RQ, we study the combination of specific communication channels (*i.e.,* a set of communication channels). For example, The combination of *pull requests and Slack* may be the best for bug fixing. We study the prevalence of certain sets of communication channels, and examine how the use of the sets correlates with several performance metrics, such as the bug fixing time. Our goal is to provide recommendations of specific sets of channels that may be the most effective for a particular development activity.

Approach.

To identify the most frequent combinations of channels, we use the Apriori algorithm [5], which performs frequent item set mining from a set of channels used together by a project P, to communicate about a given development activity. The Apriori algorithm accepts two arguments: the support and confidence.

The support reflects how popular a set X is, as measured by the proportion of transactions t in which X appears. A transaction t refers to the set of channels used together by a project in a development activity. The support is computed as follows:

$$supp(X) = \frac{t \in T; X \subseteq t}{|T|}$$

The confidence measures how likely a channel Y is used when a channel X is used, expressed as $X -> Y$. Confidence is defined as follows:

$$conf(X -> Y) = \frac{supp(XUY)}{supp(X)}$$

Using the Apriori algorithm, we identify the maximally frequent sets of channels for each development activity. A frequent set is defined as a set with a support and confidence greater than the user-specified minimum values. A maximally frequent set is a frequent set which is not contained in another frequent itemset.

To extract the frequent sets of channels, we set the support to 0.1 (*i.e.,* a set has to appear in at least 10% of the survey responses), and the confidence to 0.8 (*i.e.,* channels X and Y from a frequent set are used together 80% of the time). We use the results of the thematic analysis (performed on the answers to the survey open questions) to clarify the interplay between the combination of channels used. Therefore, we study the themes that emerge from the analysis, and show them in bolded text. We also examine the number of times that the themes were mentioned by the respondents.

To better understand the resulting frequent sets from the Apriori algorithm, we show the resulting themes (in bolded text) from the thematic analysis to explain why certain channels are used. We further report in Table 6.7 a summary of the themes associated to each of the communication channels considered in this study. The themes reflect the reasons for (not) choosing the channels.

To identify the most suited frequent channel sets in the release planning, bug fixing, and code review activities, we compute the performance metrics listed in Section 6.2.2. We then examine the association between the frequent channel sets and the performance metrics, to test the following null hypothesis:

H_a^3: *The presence or absence of a set of communication channels in a project has no association with the performance metrics of the projects.*

To test the hypothesis, we use the Kruskal-Wallis test [84] to compare the distributions of the performance metrics, with respect to the presence or absence of the frequent set of channels. To assess the magnitude of the possible difference, we calculate the effect size using Cliff's delta [111].

Results.

In release planning, bug fixing, and documentation, there exists a recurrent communication channel that appears in the most frequent sets. In the case of *release planning*, email appears to be a central channel to communicate releases (as shown in Table 6.6). In combination with emails, the projects communicate face-to-face, with pull requests, with Slack, or with issue tracking systems. As revealed by our respondents, issue tracking and pull requests are a good way to **plan future additions** of features, and to record the decision process. However, the additional **communication overhead** up to a pull request occurs through email and

Development activity	Frequent sets	Support
Release planning	1- {**Email**,Face to face}	0.15
	2- {**Email**,Pull requests}	0.15
	3- {**Email**,Slack}	0.14
	4- {**Email**,Issue tracking systems}	0.14
	5- {Face to face,Slack}	0.12
	6- {Google Hangouts,Slack}	0.12
	7- {Face to face,Issue tracking systems}	0.12
	8- {Issue tracking systems,Mailing lists}	0.11
	9- {Face to face,Pull requests}	0.11
	10- {Issue tracking systems,Pull requests,Slack}	0.10
Bug fixing	1- {**Issue tracking systems**,Pull requests}	0.34
	2- {**Issue tracking systems**,Slack}	0.16
	3- {**Issue tracking systems**,email}	0.15
	4- {Pull requests,Slack}	0.14
	5- {Email,Pull requests}	0.12
	6- {Email,Face to face}	0.12
	7- {Face to face,Issue tracking systems}	0.12
Code review	1- {Face to face}	0.22
	2- {Slack}	0.13
	3- {Issue tracking systems,Pull requests}	0.12
	4- {Email}	0.12
User support	1- {Email,Issue tracking systems}	0.21
	2- {Face to face}	0.13
	3- {Mailing lists}	0.12
	4- {Telephone}	0.11
	5- {Email,Slack}	0.10
Recruiting developers	1- {Email,Face to face}	0.12
	2- {Issue tracking systems,Pull requests}	0.10
Project promotion	1- {Twitter}	0.24
	2- {Face to face}	0.20
	3- {Email}	0.18
	4- {Mailing lists}	0.18
	5- {Facebook}	0.11
Documentation	1- {**Issue tracking systems**,Pull requests}	0.21
	2- {**Issue tracking systems**,Slack}	0.12
	3- {**Issue tracking systems**,Email}	0.10

Table 6.6: Frequent sets of communication channels for every development activity

Table 6.7: Results of the survey thematic analysis for the studied communication channels. For each theme, we show the percentage* of times that the theme appeared in an answer mentioning the associated channel.

	Communication channels	**Associated themes**
Socially-enabled	Pull requests	Recording decision process[55%] Planning future additions[45%] Providing context to bug fixing[30%] Slow progress[20%]
	Slack/Gitter	Synchronous communication[35%] Addressing delays[30%] Lost knowledge[23%] Prioritization[17%] Informal requirement development[10%]
	Google hangouts	Video call meetings[36%]
	Twitter/Facebook	Wide reach[33%] User support[22%] Lightweight[21%] Volatile[18%]
	StackOverflow	User support[16%] Documentation resource[29%]
Digital	Email/Mailing lists	Long-format design issues[56%] Specific recipients[46%] Addressing delays[37%] Widely accepted[33%]
	IRC	Widely accepted[50%]
Non-digital	Face-to-face/Phone	Optimal decision making[74%] Conflict-free[45%]

*The percentages total associated to the themes for each communication channel do not add up to 100, because a survey response could be used to derive zero or more themes, depending on how rich the response is. In some cases, no justification of the channel is provided

face-to-face. In release planning, Slack (or similar) is suited for **announcements**, and for quick exchanges to solve **ephemeral issues** that do not need to be revisited later. In both *bug fixing* and *documentation*, the issue tracking system is the

Table 6.8: Effect size and direction of the significant frequent channel sets on the performance metrics. The sets vary across the development activities and are numbered in the order they appear in Table 6.6 (n.s: not significant).

Release planning	Set 1	Set 2	Set 3	Set 4	Set 6
Release frequency	large (↗)	medium (↗)	medium (↗)	large (↗)	small (↘)
Bug fixing	Set 1	Set 2	Set 3		
Fixing time	large (↘)	large (↘)	large (↘)		
Fixing rate	small (↘)	medium (↘)	n.s		
Code review	Set 1	Set 2	Set 3		
Merge time	large (↘)	medium (↘)	medium (↘)		
Merge rate	n.s	n.s	n.s		

central medium of communication, and is frequently paired with pull requests, Slack or email. In regards to these possible combinations, our survey respondents explain that although issue tracking systems are effective for **bugs collection**, they can be inadequate for **prioritization**. *R58* explains the former point as follows: "*there are lots of bugs reported more than a year ago, with no record of why they have never been addressed.*" Therefore, Slack (or similar) is used as a **synchronous**, **fast**, and **non-invasive** form of communication for back and forth discussions around the bugs, and for obtaining further information on how to address a bug. Email is best suited for **long-format** discussions of **design aspects**, when the right approach for fixing a bug is not obvious. In some cases, the bugs are informally **reported** by email or on Slack, before the formal creation of the bug report. In bug fixing, pull requests are linked to

the bugs and are associated to tests that confirm the bugs. The linking allows to keep track of the **history of a bug fix**, with confidence and with **context**. In terms of *developers' recruitment*, we observe two distinct types of channels combination that reflect *a)* a **traditional** recruitment strategy (*i.e.,* email and face-to-face), and *b)* **contribution-based** recruitment strategy which considers prior contributions of a developer for recruitment purposes.

In code review, user support, and project promotion, it is less likely to observe frequently-used combinations of channels across the projects. In the code review, user support, and project promotion activities, the maximally frequent sets generated by the Apriori algorithm are mostly composed of a single channel only (as shown in Table 6.6). In *code review*, for instance, the most common combination of channels is composed by the issue tracking systems and the pull requests, for a **formal** and **structured** code review process. In the issue tracking systems and pull requests combination, the projects further use face-to-face, Slack, and email communication, with no combination occurring frequently enough (*i.e.,* with a support > 0.1 and confidence > 0.8). Although face-to-face remains the most **optimal** form of communication in the process of **decision making**, it is not always achievable. Therefore, Slack and email are especially useful when there is a serious **delay** on a given pull request to be merged by the assigned reviewer. Regarding *user support*, it is common across the projects to use email paired with the issue tracking system, to address the issues faced by the users. This combination of channels is common because it allows a **separation** between the *help* questions broadcasted using email, and the 'actionable' technical issues posted on the issue tracking system. However, email can be misused when the users submit their problems, often without

sufficient details and explanation. In such cases, the use of Slack or the phone could be preferred because it allows for a **back and forth exchange**, to clarify the issues, although both channels **lack context** and require the participants to reiterate the issue details. Nevertheless, in some cases, the developers **do not have full control** of how the information is sourced, as the users could reach out in a variety of ways. Finally, it common in *project promotion* to employ the **less technical** and **lightweight** communication channels (*e.g.,*Twitter) to reach the largest possible audience. In other cases, preference is given to mailing lists (or blogs) because they allow for the **inclusion of more details**, and can **'survive' longer**, contrary to social media where content is **volatile**.

The use of certain channel sets is positively associated with the performance metrics. In release planning, for instance, the frequency of the releases has a significant association (p-value = 1.78e-07) with the use of the set {email, face-to-face} (*i.e., Set 1* as numbered in Table 6.6), with a large effect size according to Cliff's delta. Specifically, projects that use *Set 1* have more frequent releases compared to the projects that do not use *Set 1*. Other sets (not included in Table 6.8), such as *Set 8*, do not show a significant association with the release frequency. Another example with an opposite effect is *Set 6* ({Google hangout, Slack}), which has a small effect size on lowering the release frequency. We show in Table 6.8 the channel sets that exhibit a significant association with the release frequency of the subject projects (along with the effect size and direction). In bug fixing, the successful sets of channels (*i.e.,* associated with a lower bug fixing time and a higher bug fixing rate) are Sets 1, 2, and 3, with large effect size. Therefore, the pairing of structured tools, such as issue tracking systems, with less structured tools, such as Slack, appears to be effective for

the bug fixing process. In terms of code review, face-to-face (*i.e., Set 1* in code review) is associated with a lowered merge time with a large effect size, thus confirming the efficiency of face-to-face interaction in the decision making process. Slack (*i.e.,* Set 2) has a similar association with a medium effect. Based on the obtained results, we can reject the null hypothesis H_a^3, and conclude that there exists an association between the communication channels used and some of the performance metrics of the subject projects.

*The frequent sets of communication channels that we identify highlight the communication needs of the developers, with respect to every development activity. We further uncover significant associations between some of the channel sets (*e.g., *pull request and Slack) and some performance indicators of the projects (*e.g., *bug fixing time).*

6.4 Discussion

In this section, we synthesize our findings to provide a set of recommendations to the project maintainers, to help in setting up an efficient communication flow.

R1 – It is best to tailor the communication channels used to each development activity. The developers' feedback reveals a strong affinity with the socially-enabled channels for most development activities, as discussed in RQ2. However, not all development activities are equally 'social'. For instance, the code review process, which by nature involves extensive discussions, is the most (69.4%) and second most (77.3%) 'social' activity in the open and private repositories. Other activities, such as user support and recruiting developers, are still likely to use other categories of channels (*i.e.,* digital and non-digital). Therefore, it is best to tailor the choice of channels

to the intended purpose (*e.g.,* how much discussion is involved in the process?), and audience (*e.g.,* where are the project users most active?).

R2 – For every development activity, it is important to put in place both formal and informal communication mechanisms. When setting up the communication channels to be used, it is also important to consider including both formal (*e.g.,* issue tracking systems) and informal (*e.g.,* chat) mechanisms. The formal mechanisms are used to **maintain permanent record** and for **long-format content**; whereas the informal mechanisms are considered by our respondents as '**low friction**' and allowing a **quick feedback loop**. In this regard, R86 states "*Email / lists are the most 'widely accepted', but Twitter hits people fastest, and then Issues / PRs are usually where those invested more heavily in specific things would care to read*". R16 further clarifies that "*Slack ends up being the backbone for communication, with the details of the process being captured in trackers and PRs. I barely check my email more than once a day, whereas I have Slack open all the time and tabs open in my browser for our tracker and Github*". Two respondents also reveal that the review time of pull requests is "by far the bottleneck of the development process". Therefore, it is essential to use synchronous and informal channels, such as Slack, to draw the attention of the parties involved when an action is required.

R3 – Opting for less channels is best, when accountability and traceability are of the essence. Having multiple options to reach the team members or the project users is valuable. However, information, such as rationale behind decisions, can drown in the midst of incessant streams of conversations. Therefore, using fewer channels is key to keeping track of the collective thought process in an organized and trackable manner, when needed. Similar to traceability, accountability can be lost

if too many channels are used to communicate about some activities, such as code review. R110 explains that "*some issues are sometimes raised, discussed, and fixed in Slack*". This becomes problematic when such issues need to be revisited in the future, with no obvious trace of the involved developers.

R4 – Complementing (instead of replacing) the traditional communication tools is a possible winner. The feedback from our respondents has made it clear that non-digital channels, such as face-to-face or phone, are invaluable and lead to little to no misunderstandings. The analysis of the survey results indicates that an equal use of socially-enabled and non-digital communication channels is possibly the most satisfying combination. As such, it is best to find a middle ground between going 'social' and remaining 'traditional' when possible, to get the best of both worlds. Indeed, R63 complained about "*institutional inertia*", as their associated company is unable to adjust to remote work processes, and are more comfortable conducting code reviews in person. On the other hand, R12 and R78 explain that although the adopted tools are adequate, face to face is best to "*exchange on complex issues*", and to "*recruit the right developers*".

R5 – The use of social features should be accompanied by clearly outlined protocols of use. It comes as no surprise that going overly 'social' can be disruptive to the development process. 10 of our respondents mentioned that the socially-enabled tools can open the way for unwanted, overwhelming, and irrelevant communications. It is particularly applicable for teams that do not restrict the access to their communication channels (such as Gitter chatrooms). Therefore, it is recommended to outline a set of rules and broadcast it among the participants, to

limit unwanted and intrusive behaviour. Examples of rules are the use of proper language, no solicitation or spamming, participating with a unique username, and not monopolizing the conversation.

6.5 Limitations

To study the experience of developers in using the communication channels, we distribute a survey to developers. The survey inclusion criterion is activity in the GitHub social coding platform. This possibly suggests a bias towards the developers that favor the use of the socially-enabled platforms (*e.g.,* GitHub) in the open repositories. Fortunately, 17.8% of our respondents are most active in private repositories, thus giving us a glimpse of the workings of the private projects that either limit the access of their repositories to the members, or are corporate entities that do not adopt social coding platforms. In terms of demographics, our respondents are mostly males (89%), with over 10 years of software development experiences (50.3%), who maintain the projects (51.2%). As a result, our findings may be biased in terms of showing the perceptions of the more experienced male developers that act as maintainers.

In our survey, we ask the developers which communication channels they use for a given activity, given a list of channels to choose from. As such, the data collected about the use of the communication channels is binary, *i.e.,* a team uses a channel X (or not) for a given activity. However, the data collected does not reflect the amount of communication occurring over a channel X. Indeed, different teams may be using the same channels, but with varying frequencies. To address part of this concern, we specified in our survey questions that we are interested in the *primary* communication channels used.

The low response rate to our survey (5.37%) is a limitation of our results, as the sample might not be representative of the entire population. There are two elements that mitigate part of this risk. First, the responses are associated to 129 different projects, thus giving us insights about a large pool of projects. Second, despite collecting feedback from one respondent about each of the 129 projects, the respondents are confident about their knowledge of the development activities within their projects. Specifically, 88% of the respondents reported that they are either *familiar* or *very familiar* with the development activities included in the survey.

To measure the perceived satisfaction of the respondents, we use Likert scale ratings (*e.g.,* numeric scales from 1 to 5). Despite the popularity of the Likert scale ratings, it is associated with issues such as the assumption of an even distance between the various points (*e.g.,* distance between *good* and *excellent* is similar to the distance between *neutral* and *good*). Another challenge associated to the Likert scale is that, in case respondents are confused by a question, or wish to respond in a way that is not available on the scale itself, they would respond by giving a midline response. To attempt to offset these challenges, we refrain from using statistical tests to evaluate the scales as normally distributed, parametric data. Instead, we opt for a visualization of the ratings using a stacked barplots to identify the cases that exhibit the highest satisfaction. To mitigate the subjectivity inherent to the Likert scale, we ask the respondents follow-up open ended questions to justify or clarify their ratings.

Regarding the quantitative analysis performed in RQ3, we observe significant associations between specific combinations of channels and a set of performance metrics, based on 58 repositories. The performance of projects is a result of a complex mix of

factors, both technical and social. Therefore, we do not claim that the choice of communication channels can fully explain the performance of projects, and only report the observed significant associations to possibly highlight successful combinations of channels.

6.6 Summary

Software development is an increasingly social, monitored, and distributed activity. Therefore, the communication ecology formed by the developers is increasingly complex, with great amounts of information exchanged among the participants. In this study, we aim to capture the experience of developers in using a set of communication channels, to communicate about specific development activities. First, we examine the feedback from the survey respondents from two perspectives: *a)* the complexity of the communication in terms of the number of channels used, and *b)* the nature of the channels used. We find that the different development activities call for personalized communication flows, in terms of both the number of channels used and the nature of the channels. For instance, the user support activity appears to be best achieved with a fewer number of channels (to reduce the confusion of the users in contacting the developers). Most development activities show a higher satisfaction with a mix of socially-enabled and non-digital channels. Second, we identify the most frequent combinations of tools used by the developers, and assess their impact on a set of performance measures. Our analysis reveals significant associations between some of the frequent sets of channels (*e.g.,* pull request and Slack) and specific performance measures of the projects (*e.g.,* shorter bug fixing time). Overall, it appears that setting up the communication flow of the projects should not be an ad-hoc activity (*e.g.,*

choosing a tool that is trending), but rather a thought-out process that considers the particular needs of the project and all the stockholders involved.

Chapter 7

Conclusions and Future Work

In this chapter, we summarize our work and present the potential opportunities for future work. The social knowledge surrounding the process of developing software provides a unique window into the main actors involved in this process, *i.e.,* the developers. In many project settings, the collaboration among the developers and among the developers and their users is mediated using the social and communication channels. As such, developing a deep understanding of the affordances and impacts of said channels is crucial to build a fuller picture of the software development process.

In this dissertation, we conducted a set of empirical studies investigating three types of social knowledge: 1) crowd-sourced feedback, 2) organizational structure, 3) wisdom of the developers. The studies revealed interesting insights to guide the developers and the project maintainers towards better use of the social and communication channels. We reiterate below the main contributions that resulted from the studies conducted.

7.1 Contributions

The research work described in this dissertation makes the following overarching contributions:

Demonstrating the value of crowd-sourced feedback

By building a tool to map bug-reporting messages on Twitter, *i.e.,* an important venue for crowd-sourced feedback, to formal bug reports, we are able to demonstrate that the tweets can reveal the types of bugs most critical to a large user-base after a new software release. For instance, the end-users are most likely to complain about problems relating to performance, compared to problem in the appearance of the software. More importantly, our study reveals that monitoring the end-users tweets can potentially speed up the bug fixing process, thanks to an earlier discovery of the bugs.

Identifying winner team structures in social coding

We systematically define, identify, and characterize the team structures used in the context of social coding. Our study reveals that the team structure of a project can partly explain the performance of the developers in processing incoming code submissions. Specifically, team structures, where developers are well connected (*i.e.,* developers collaborate with many (if not most) of their teammates), are centralized around key contributors, and possibly take roles as both integrators and contributors of code submissions, are associated to higher performance. Finally, we provide recommendations on warning signs that a team is evolving towards a structure associated to lower performance.

Revealing the affordances of channels used for informal communication

We collect the developers' experiences on the use of an open source (*i.e.,* Gitter) and a proprietary (*i.e.,* Slack) chat services. We obtain insights about the uses and perceived impact of chatrooms by the open communities (in both Slack and Gitter) and the corporate teams (mostly in Slack). We find that the chatrooms are used by the developers to exchange timely and expert knowledge, motivated by intrinsic reasons (*e.g.,* support the project community), and extrinsic reasons (*e.g.,* improve the reputations). We further reveal that the chatrooms reportedly impact the communication management, the development directions, and the resolution time of issues. We finally provide recommendations that highlight when it is best to use either chatroom, depending on the needs of the projects

Providing recommendations on setting up the communication flow of projects

We capture the experience of developers in using a set of communication channels, to communicate about specific development activities. We provide recommendations on the most suitable channels to use, in terms of number and nature, for each investigated development activities. For instance, the user support activity appears to be best achieved with a fewer number of channels (to lower the confusion of the users in contacting the developers). We also identify the most frequent combinations of tools used by the developers, and assess their impact on a set of performance measures. Our analysis reveals significant associations between some of the frequent sets of channels (*e.g.,* pull request and Slack) and specific performance measures of the projects (*e.g.,* lowered bug fixing time).

7.2 Future Work

The research work presented in this thesis highlights the need to account for, examine, and monitor the social aspects of the development process. Below, we propose some potential research opportunities that may benefit this line of work in the future.

Extracting knowledge from social interactions

The use of socially-enabled collaboration channels in software development is a double-edged sword. From one hand, this type of channels affords synchronous and transparent communication. From the other hand, communication using socially-enabled channels may result in the loss of valuable knowledge. As developers communicate to discuss issues, to debate design decisions, or to provide user support, much expert knowledge is exchanged but is not properly documented or saved for future reference. The volatility of the socially-enabled channels can be detrimental in the long run in terms of preserving the knowledge surrounding the building and maintenance of a software system. Therefore, we believe that it is valuable in the future to build approaches and tools aimed at extracting the knowledge present in such channels. We list below examples of types of knowledge that can possibly be extracted from the developers' interactions.

- In the communication channels where user support is provided, such as chatrooms, it would be beneficial to analyze the chat and capture the most Frequently Asked Questions (FAQs) for two reasons. First, the FAQs would highlight documentation problems, and therefore help the developers improve the software documentation and avoid repetitive inquiries. Second, the FAQs could

also capture weak parts about the software systems, and point the developers towards possible improvement in their code.

- In the channels where developers discuss project matters, such as bugs and features, it would be valuable to capture the expertise and interests of the developers at a granular level. For instance, a certain developer might exhibit more affinity towards bugs and features related to improving the performance of the system. With such knowledge captured, concerned developers could be notified when a topic of interest is raised in the discussion; thus making the discussion more efficient.

- In the channels where the developers explicitly collaborate by reviewing each others code, such as in pull requests, we propose to monitor the success of collaborations in terms of the developers involved, the time, and the quality of the produced code. This knowledge can be used by project maintainers to encourage successful collaborations, and keep an eye out for risky collaborations.

Building automated tools to better monitor social interactions

In the classic problem of predicting defects in software projects, a long line of work proposed a multitude of approaches and metrics to predict the existence and criticality of the bugs (*e.g.,* [73][95][97]). Given the importance of social interactions on the performance of building and maintaining software systems, we believe it would be important to design and extract metrics capturing the 'health' of social interactions in the software development process. Prior work (including our own) has included few social metrics, such as prior connection of a developer to a project in [134], or the reciprocity of collaborations among the developers [49] to predict performance

indicators of projects. However, more effort could be invested in designing metrics to fully capture different social aspects of the software development process. The resulting metrics could be used to build more comprehensive models to study the associations between project metrics (both social and technical), and the projects' performance indicators. The resulting metrics could also be used to build automated tools aimed at assessing how successful a team is in their communication. We show below examples of social metrics and their possible uses:

- ***Quality of interaction metrics***: it could be valuable to collect, from the developers interactions, metrics that reflect the quality of a given interaction. Examples of such metrics are the tone (*e.g.,* how friendly was the exchange), the efficiency (*e.g.,* how long was the interaction?), and the outcome (*e.g.,* did the interaction result in the desired outcome?). Project maintainers could monitor how successful interactions are between the developers, to address situations where the developers are not able to collaborate successfully.

- ***Quality of channel metrics***: As far as the use of channels is concerned, we reveal in Chapter 6 the importance of a thought-out communication flow. We now conjecture that the decision about which channels to use should not be static, but rather evolutionary based on the projects' needs. Therefore, we propose to design metrics and tools to capture the success of a given channel. For instance, the metrics could reflect the time spent on the channel to complete a task, the satisfaction of the developers in using the channel, the measured impact of the channel on the productivity of the developers. This type of metrics could capture the overall value of a communication channel, and guide the decision of the project maintainers in adjusting their communication flow.

Bibliography

[1] Gitter - explore. `https://gitter.im/explore/`. Accessed: 2018-09-30.

[2] Slack - about us. `https://slack.com/about`. Accessed: 2018-09-30.

[3] Twitter: number of active users 2010-2018.

[4] 14-year-old's facetime bug discovery could rattle apple, Feb 2019.

[5] Rakesh Agrawal, Ramakrishnan Srikant, et al. Fast algorithms for mining association rules. In *Proc. 20th int. conf. very large data bases, VLDB*, volume 1215, pages 487–499, 1994.

[6] Navid Ahmadi, Mehdi Jazayeri, Francesco Lelli, and Sasa Nesic. A survey of social software engineering. In *ASE Workshops*, pages 1–12. IEEE, 2008.

[7] Miltiadis Allamanis and Charles Sutton. Why, when, and what: Analyzing stack overflow questions by topic, type, and code. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 53–56, Piscataway, NJ, USA, 2013. IEEE Press.

[8] Brigham S Anderson, Carter Butts, and Kathleen Carley. The interaction of size and density with graph-level indices. *Social Networks*, 21(3):239–267, 1999.

[9] R. Arun, V. Suresh, C. E. Veni Madhavan, and M. N. Narasimha Murthy. On finding the natural number of topics with latent dirichlet allocation: Some observations. In *Proceedings of the 14th Pacific-Asia Conference on Advances in Knowledge Discovery and Data Mining - Volume Part I*, PAKDD'10, pages 391–402, Berlin, Heidelberg, 2010. Springer-Verlag.

[10] Alberto Bacchelli, Tommaso Dal Sasso, Marco D'Ambros, and Michele Lanza. Content classification of development emails. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 375–385. IEEE, 2012.

[11] Pierre F. Baldi, Cristina V. Lopes, Erik J. Linstead, and Sushil K. Bajracharya. A theory of aspects as latent topics. *SIGPLAN Not.*, 43(10):543–562, October 2008.

[12] Earl T Barr, Christian Bird, Peter C Rigby, Abram Hindle, Daniel M German, and Premkumar Devanbu. Cohesive and isolated development with branches. In *Fundamental Approaches to Software Engineering*, pages 316–331. Springer, 2012.

[13] Ohad Barzilay, Christoph Treude, and Alexey 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.

[14] Olga Baysal, Reid Holmes, and Michael W Godfrey. No issue left behind: Reducing information overload in issue tracking. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 666–677. ACM, 2014.

[15] Andrew Begel, Jan Bosch, and Margaret-Anne Storey. Social networking meets software development: Perspectives from GitHub, MSDN, Stack Exchange, and TopCoder. *IEEE Softw.*, 30(1):52–66, January 2013.

[16] Yoav Benjamini and Yosef Hochberg. Controlling the false discovery rate: a practical and powerful approach to multiple testing. *Journal of the royal statistical society. Series B (Methodological)*, pages 289–300, 1995.

[17] Yoav Benjamini and Daniel Yekutieli. The control of the false discovery rate in multiple testing under dependency. *Annals of statistics*, pages 1165–1188, 2001.

[18] Francesco S. Bersani, Daniel Lindqvist, Synthia H. Mellon, Elissa S. Epel, Rachel Yehuda, Janine Flory, Clare Henn-Hasse, Linda M. Bierer, Iouri Makotkine, Duna Abu-Amara, Michelle Coy, Victor I. Reus, Jue Lin, Elizabeth H. Blackburn, Charles Marmar, and Owen M. Wolkowitz. Association of dimensional psychological health measures with telomere length in male war veterans. *Journal of Affective Disorders*, 190:537 – 542, 2016.

[19] Nicolas Bettenburg and Ahmed E Hassan. Studying the impact of social structures on software quality. In *Program Comprehension (ICPC), 2010 IEEE 18th International Conference on*, pages 124–133. IEEE, 2010.

[20] Ann Bies, Mark Ferguson, Karen Katz, Robert MacIntyre, Victoria Tredinnick, Grace Kim, Mary Ann Marcinkiewicz, and Britta Schasberger. Bracketing guidelines for treebank ii style penn treebank project. *University of Pennsylvania*, 97:100, 1995.

[21] Christian Bird, Alex Gourley, Prem Devanbu, Anand Swaminathan, and Greta Hsu. Open borders? immigration in open source projects. In *Proceedings of the Fourth International Workshop on Mining Software Repositories*, MSR '07, pages 6–, Washington, DC, USA, 2007. IEEE Computer Society.

[22] Steven Bird. Nltk: The natural language toolkit. In *Proceedings of the COLING/ACL on Interactive Presentation Sessions*, COLING-ACL '06, pages 69–72. Association for Computational Linguistics, 2006.

[23] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. Latent dirichlet allocation. *J. Mach. Learn. Res.*, 3:993–1022, March 2003.

[24] Kelly Blincoe, Jyoti Sheoran, Sean Goggins, Eva Petakovic, and Daniela Damian. Understanding the popular users: Following, affiliation influence and leadership on github. *Information and Software Technology*, 70:30 – 39, 2016.

[25] Amiangshu Bosu, Christopher S. Corley, Dustin Heaton, Debarshi Chatterji, Jeffrey C. Carver, and Nicholas A. Kraft. Building reputation in stackoverflow: An empirical investigation. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 89–92, Piscataway, NJ, USA, 2013. IEEE Press.

[26] Gargi Bougie, Jamie Starke, Margaret-Anne Storey, and Daniel M. German. Towards understanding twitter use in software engineering: Preliminary findings, ongoing challenges and future questions. In *Proceedings of the 2Nd International Workshop on Web 2.0 for Software Engineering*, Web2SE '11, pages 31–36. ACM, 2011.

[27] Virginia Braun and Victoria Clarke. Using thematic analysis in psychology. *Qualitative research in psychology*, 3(2):77–101, 2006.

[28] Frederick P Brooks Jr. *The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition, 2/E*. Pearson Education India, 1995.

[29] Chris Buckley and Ellen M. Voorhees. Evaluating evaluation measure stability. In *Proceedings of the 23rd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '00, pages 33–40, New York, NY, USA, 2000. ACM.

[30] Jacques Bughin, Richard Dobbs, Charles Roxburgh, Hugo Sarrazin, Geoffrey Sands, and Magdalena Westergren. The social economy: Unlocking value and productivity through social technologies. *McKinsey. com*, 2012.

[31] Carter T Butts et al. Social network analysis with sna. *Journal of Statistical Software*, 24(6):1–51, 2008.

[32] Juan Cao, Tian Xia, Jintao Li, Yongdong Zhang, and Sheng Tang. A density-based method for adaptive lda model selection. *Neurocomput.*, 72(7-9):1775–1781, March 2009.

[33] Casey Casalnuovo, Bogdan Vasilescu, Premkumar Devanbu, and Vladimir Filkov. Developer onboarding in github: the role of prior social links and language experience. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 817–828. ACM, 2015.

[34] Huseyin Cavusoglu, Zhuolun Li, and Ke-Wei Huang. Can gamification motivate voluntary contributions?: The case of stackoverflow q&a community. In

Proceedings of the 18th ACM Conference Companion on Computer Supported Cooperative Work & Social Computing, CSCW'15 Companion, pages 171–174, New York, NY, USA, 2015. ACM.

[35] Hyunyoung Choi and Hal Varian. Predicting the present with google trends. *Economic Record*, 88:2–9, 2012.

[36] Norman Cliff. Dominance statistics: Ordinal analyses to answer ordinal questions. *Psychological Bulletin*, 114(3):494–509, November 1993.

[37] Jailton Coelho and Marco Tulio Valente. Why modern open source projects fail. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 186–196, New York, NY, USA, 2017. ACM.

[38] Kevin Crowston and James Howison. The social structure of free and open source software development. *First Monday*, 10(2), 2005.

[39] Bill Curtis, Herb Krasner, and Neil Iscoe. A field study of the software design process for large systems. *Communications of the ACM*, 31(11):1268–1287, 1988.

[40] Daniel Alencar da Costa, Shane McIntosh, Christoph Treude, Uirá Kulesza, and Ahmed E Hassan. The impact of rapid release cycles on the integration delay of fixed issues. *Empirical Software Engineering*, pages 1–70, 2018.

[41] Laura Dabbish, Colleen Stuart, Jason Tsay, and Jim Herbsleb. Social coding in github: Transparency and collaboration in an open software repository. In *Proceedings of the ACM 2012 Conference on Computer Supported Cooperative Work*, CSCW '12, pages 1277–1286, New York, NY, USA, 2012. ACM.

[42] Richard L Daft and Robert H Lengel. Organizational information requirements, media richness and structural design. *Management science*, 32(5):554–571, 1986.

[43] Marcel A. de Reus and Martijn P. van den Heuvel. The parcellation-based connectome: Limitations and extensions. *NeuroImage*, 80:397 – 404, 2013. Mapping the Connectome.

[44] Sebastian Deterding. Gamification: designing for motivation. *interactions*, 19(4):14–17, 2012.

[45] Sebastian Deterding, Miguel Sicart, Lennart Nacke, Kenton O'Hara, and Dan Dixon. Gamification. using game-design elements in non-gaming contexts. In *CHI'11 extended abstracts on human factors in computing systems*, pages 2425–2428. ACM, 2011.

[46] Romain Deveaud, Eric SanJuan, and Patrice Bellot. Accurate and effective latent concept modeling for ad hoc information retrieval. *Document numérique*, 17:61–84, 2014. DOI : 10.3166/DN.17.1.61-84.

[47] Nicolas Ducheneaut. Socialization in an open source software community: A socio-technical analysis. *Computer Supported Cooperative Work (CSCW)*, 14(4):323–368, 2005.

[48] Kate Ehrlich and Marcelo Cataldo. All-for-one and one-for-all?: A multi-level analysis of communication patterns and individual performance in geographically distributed software development. In *Proceedings of the ACM 2012 Conference on Computer Supported Cooperative Work*, CSCW '12, pages 945–954, New York, NY, USA, 2012. ACM.

[49] Mariam El Mezouar, Feng Zhang, and Ying Zou. An empirical study on the teams structures in social coding using github projects. *Empirical Software Engineering*, May 2019.

[50] Joseph L Fleiss. Measuring nominal scale agreement among many raters. *Psychological bulletin*, 76(5):378, 1971.

[51] Linton C Freeman. A set of measures of centrality based on betweenness. *Sociometry*, pages 35–41, 1977.

[52] Linton C Freeman. Centrality in social networks conceptual clarification. *Social networks*, 1(3):215–239, 1978.

[53] Cristina Gacek and Budi Arief. The many meanings of open source. *IEEE software*, 21(1):34–40, 2004.

[54] Diego Garlaschelli and Maria I Loffredo. Patterns of link reciprocity in directed networks. *Physical review letters*, 93(26):268701, 2004.

[55] Mohammad Gharehyazie, Daryl Posnett, Bogdan Vasilescu, and Vladimir Filkov. Developer initiation and social interactions in oss: A case study of the apache software foundation. *Empirical Software Engineering*, 20(5):1318–1353, 2015.

[56] Rosalba Giuffrida and Yvonne Dittrich. A conceptual framework to study the role of communication through social software for coordination in globally-distributed software teams. *Information and Software Technology*, 63:11–30, 2015.

[57] Alec Go, Lei Huang, and Richa Bhayani. Twitter sentiment analysis. *Entropy*, 17, 2009.

[58] Max Goldman, Greg Little, and Robert C Miller. Real-time collaborative coding in a web ide. In *Proceedings of the 24th annual ACM symposium on User interface software and technology*, pages 155–164. ACM, 2011.

[59] Keith J Goulden. Effect sizes for research: a broad practical approach. *Journal of Developmental & Behavioral Pediatrics*, 27(5):419–420, 2006.

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

[61] Georgios Gousios, Martin Pinzger, and Arie van Deursen. An exploratory study of the pull-based software development model. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 345–355, New York, NY, USA, 2014. ACM.

[62] Georgios Gousios, Margaret-Anne Storey, and Alberto Bacchelli. Work practices and challenges in pull-based development: The contributor's perspective. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 285–296, New York, NY, USA, 2016. ACM.

[63] Georgios Gousios, Andy Zaidman, Margaret-Anne Storey, and Arie van Deursen. Work practices and challenges in pull-based development: The integrator's perspective. In *Proceedings of the 37th International Conference on*

Software Engineering - Volume 1, ICSE '15, pages 358–368, Piscataway, NJ, USA, 2015. IEEE Press.

[64] Thomas L. Griffiths and Mark Steyvers. Finding scientific topics. *Proceedings of the National Academy of Sciences*, 101(suppl 1):5228–5235, 2004.

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

[66] Carl Gutwin, Reagan Penner, and Kevin Schneider. Group awareness in distributed software development. In *Proceedings of the 2004 ACM Conference on Computer Supported Cooperative Work*, CSCW '04, pages 72–81, New York, NY, USA, 2004. ACM.

[67] Carl Gutwin, Reagan Penner, and Kevin Schneider. Group awareness in distributed software development. In *Proceedings of the 2004 ACM conference on Computer supported cooperative work*, pages 72–81. ACM, 2004.

[68] A. Guzzi, A. Bacchelli, M. Lanza, M. Pinzger, and A. van Deursen. Communication in open source software development mailing lists. In *2013 10th Working Conference on Mining Software Repositories (MSR)*, pages 277–286, May 2013.

[69] Anja Guzzi, Alberto Bacchelli, Michele Lanza, Martin Pinzger, and Arie van Deursen. Communication in open source software development mailing lists. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 277–286. IEEE Press, 2013.

[70] Jungpil Hahn, Jae Yun Moon, and Chen Zhang. Emergence of new project teams from open source software developer networks: Impact of prior collaboration ties. *Information Systems Research*, 19(3):369–391, 2008.

[71] Mark S Handcock, David R Hunter, Carter T Butts, Steven M Goodreau, and Martina Morris. statnet: Software tools for the representation, visualization, analysis and simulation of network data. *Journal of statistical software*, 24(1):1548, 2008.

[72] Mark Handel and James D Herbsleb. What is chat doing in the workplace? In *Proceedings of the 2002 ACM conference on Computer Supported Cooperative Work*, pages 1–10. ACM, 2002.

[73] Ahmed E. Hassan. Predicting faults using the complexity of code changes. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 78–88, Washington, DC, USA, 2009. IEEE Computer Society.

[74] James D. Herbsleb and Audris Mockus. An empirical study of speed and communication in globally distributed software development. *IEEE Transactions on Software Engineering*, 29(6):481–494, 2003.

[75] James D Herbsleb and Deependra Moitra. Global software development. *IEEE software*, 18(2):16–20, 2001.

[76] Felix Hill, Roi Reichart, and Anna Korhonen. Simlex-999: Evaluating semantic models with (genuine) similarity estimation. *Computational Linguistics*, 2016.

[77] Qiaona Hong, Sunghun Kim, S. C. Cheung, and Christian Bird. Understanding a developer social network and its evolution. In *Proceedings of the 2011 27th*

IEEE International Conference on Software Maintenance, ICSM '11, pages 323–332, Washington, DC, USA, 2011. IEEE Computer Society.

[78] James Howison, Keisuke Inoue, and Kevin Crowston. Social dynamics of free and open source team communications. In *IFIP International Conference on Open Source Systems*, pages 319–330. Springer, 2006.

[79] Enio G Jelihovschi, José Cláudio Faria, and Ivan Bezerra Allaman. Scottknott: a package for performing the scott-knott clustering algorithm in r. *TEMA (São Carlos)*, 15(1):3–17, 2014.

[80] Yujuan Jiang, Bram Adams, and Daniel M. German. Will my patch make it? and how fast?: Case study on the linux kernel. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 101–110, Piscataway, NJ, USA, 2013. IEEE Press.

[81] Mitchell Joblin, Sven Apel, Claus Hunsen, and Wolfgang Mauerer. Classifying developers into core and peripheral: An empirical study on count and network metrics. In *Software Engineering (ICSE), 2017 IEEE/ACM 39th International Conference on*, pages 164–174. IEEE, 2017.

[82] Efthymios Kouloumpis, Theresa Wilson, and Johanna D. Moore. *Twitter Sentiment Analysis: The Good the Bad and the OMG!*, pages 538–541. AAAI Press, 2011.

[83] David Krackhardt. Graph theoretical dimensions of informal organizations. *Computational organization theory*, 89(112):123–140, 1994.

[84] William H Kruskal and W Allen Wallis. Use of ranks in one-criterion variance analysis. *Journal of the American statistical Association*, 47(260):583–621, 1952.

[85] J Richard Landis and Gary G Koch. The measurement of observer agreement for categorical data. *biometrics*, pages 159–174, 1977.

[86] Bin Lin, Alexey Zagalsky, Margaret-Anne Storey, and Alexander Serebrenik. Why developers are slacking off: Understanding how software teams use slack. In *Proceedings of the 19th ACM Conference on Computer Supported Cooperative Work and Social Computing Companion*, pages 333–336. ACM, 2016.

[87] E. Linstead and P. Baldi. Mining the coherence of gnome bug reports with statistical topic models. In *6th IEEE International Working Conference on Mining Software Repositories. MSR '09.*, pages 99–102, May 2009.

[88] Bing Liu and Lei Zhang. A survey of opinion mining and sentiment analysis. In *Mining text data*, pages 415–463. Springer, 2012.

[89] Matt MacMahon, Brian Stankiewicz, and Benjamin Kuipers. Walk the talk: Connecting language, knowledge, and action in route instructions. *Def*, 2(6):4, 2006.

[90] Lena Mamykina, Bella Manoim, Manas Mittal, George Hripcsak, and Björn Hartmann. Design lessons from the fastest q&a site in the west. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 2857–2866. ACM, 2011.

[91] Henry B Mann and Donald R Whitney. On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics*, pages 50–60, 1947.

[92] Jennifer Marlow, Laura Dabbish, and Jim Herbsleb. Impression formation in online peer production: Activity traces and personal profiles in github. In *Proceedings of the 2013 Conference on Computer Supported Cooperative Work*, CSCW '13, pages 117–128, New York, NY, USA, 2013. ACM.

[93] Andrew Meneely, Laurie Williams, Will Snipes, and Jason Osborne. Predicting failures with developer networks and social network analysis. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 13–23. ACM, 2008.

[94] Audris Mockus, Roy T. Fielding, and James D. Herbsleb. Two case studies of open source software development: Apache and mozilla. *ACM Trans. Softw. Eng. Methodol.*, 11(3):309–346, July 2002.

[95] Raimund Moser, Witold Pedrycz, and Giancarlo Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 181–190, New York, NY, USA, 2008. ACM.

[96] Nachiappan Nagappan and Thomas Ball. Using software dependencies and churn metrics to predict field failures: An empirical case study. In *Proceedings of the First International Symposium on Empirical Software Engineering and Measurement*, ESEM '07, pages 364–373, Washington, DC, USA, 2007. IEEE Computer Society.

[97] Nachiappan Nagappan, Thomas Ball, and Andreas Zeller. Mining metrics to predict component failures. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 452–461, New York, NY, USA, 2006. ACM.

[98] Brendan O'Connor, Ramnath Balasubramanyan, Bryan R Routledge, and Noah A Smith. From tweets to polls: Linking text sentiment to public opinion time series. *ICWSM*, 11(122-129):1–2, 2010.

[99] Alexander Pak and Patrick Paroubek. Twitter as a corpus for sentiment analysis and opinion mining. In *LREC*, volume 10, pages 1320–1326, 2010.

[100] Chris Parnin, Christoph Treude, Lars Grammel, and Margaret-Anne Storey. Crowd documentation: Exploring the coverage and the dynamics of api discussions on stack overflow. *Georgia Institute of Technology, Tech. Rep*, 2012.

[101] Karl Pearson. X. on the criterion that a given system of deviations from the probable in the case of a correlated system of variables is such that it can be reasonably supposed to have arisen from random sampling. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 50(302):157–175, 1900.

[102] X. Peng, M. Ali Babar, and C. Ebert. Collaborative software development platforms for crowdsourcing. *IEEE Software*, 31(2):30–36, Mar 2014.

[103] Charlie Picorini. Lucene-java wiki: Powered by. `https://wiki.apache.org/lucene-java/PoweredBy`, sep 2015. [Online; accessed 19-December-2016].

[104] Heather A Piwowar. Who shares? who doesn't? factors associated with openly archiving raw research data. *PloS one*, 6(7):e18657, 2011.

[105] Martin F Porter. An algorithm for suffix stripping. *Program*, 14(3):130–137, 1980.

[106] Rick. View long-running pull requests, Jan 2013.

[107] P. C. Rigby and M. Storey. Understanding broadcast based peer review on open source software projects. In *2011 33rd International Conference on Software Engineering (ICSE)*, pages 541–550, May 2011.

[108] Peter C Rigby, Earl T Barr, Christian Bird, Prem Devanbu, and Daniel M German. What effect does distributed version control have on oss project organization? In *Release Engineering (RELENG), 2013 1st International Workshop on*, pages 29–32. IEEE, 2013.

[109] Peter C. Rigby and Margaret-Anne Storey. Understanding broadcast based peer review on open source software projects. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 541–550, New York, NY, USA, 2011. ACM.

[110] Jeffrey Robertsa, Il-Horn Hann, and Sandra Slaughter. Communication networks in an open source software project. In *IFIP International Conference on Open Source Systems*, pages 297–306. Springer, 2006.

[111] J. Romano, J.D. Kromrey, J. Coraggio, and J. Skowronek. Appropriate statistics for ordinal level data: Should we really be using t-test and Cohen'sd for

evaluating group differences on the NSSE and other surveys? In *annual meeting of the Florida Association of Institutional Research*, pages 1–3, 2006.

[112] Steve Rowe. Lucene-java wiki: Lucene faq. `https://wiki.apache.org/lucene-java/PoweredBy`, dec 2013. [Online; accessed 19-December-2016].

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

[114] Daniel Schall. Who to follow recommendation in large-scale online development communities. *Information and Software Technology*, 56(12):1543–1555, 2014.

[115] Barry Schwartz. A new click through rate study for google organic results, oct 2014.

[116] Abhishek Sharma, Yuan Tian, and David Lo. What's hot in software engineering twitter space? In *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*, pages 541–545. IEEE, 2015.

[117] David J. Sheskin. *Handbook of Parametric and Nonparametric Statistical Procedures*. Chapman & Hall/CRC, 4 edition, 2007.

[118] David J. Sheskin. *Handbook of Parametric and Nonparametric Statistical Procedures*. Chapman & Hall/CRC, 4 edition, 2007.

[119] Sidney Siegel. *Nonparametric statistics for the behavioral sciences*. McGraw-hill, 1956.

[120] Leif Singer, Fernando Figueira Filho, Brendan Cleary, Christoph Treude, Margaret-Anne Storey, and Kurt Schneider. Mutual assessment in the social programmer ecosystem: An empirical investigation of developer profile aggregators. In *Proceedings of the 2013 Conference on Computer Supported Cooperative Work*, CSCW '13, pages 103–116, New York, NY, USA, 2013. ACM.

[121] Leif Singer, Fernando Figueira Filho, and Margaret-Anne Storey. Software engineering at the speed of light: How developers stay current using twitter. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 211–221, New York, NY, USA, 2014. ACM.

[122] Richard Socher, John Bauer, Christopher D Manning, and Andrew Y Ng. Parsing with compositional vector grammars. In *ACL (1)*, pages 455–465, 2013.

[123] Kalyanasundaram Somasundaram and Gail C. Murphy. Automatic categorization of bug reports using latent dirichlet allocation. In *Proceedings of the 5th India Software Engineering Conference*, ISEC '12, pages 125–130, 2012.

[124] Robert George Douglas Steel and James Hiram Torrie. *Principles and procedures of statistics: with special reference to the biological sciences*. McGraw-Hill, 1960.

[125] M. Storey, A. Zagalsky, F. F. Filho, L. Singer, and D. M. German. How social and communication channels shape and challenge a participatory culture in software development. *IEEE Transactions on Software Engineering*, 43(2):185–204, Feb 2017.

[126] M. Storey, A. Zagalsky, F. F. Filho, L. Singer, and D. M. German. How social and communication channels shape and challenge a participatory culture in software development. *IEEE Transactions on Software Engineering*, 43(2):185–204, Feb 2017.

[127] Margaret-Anne Storey, Leif Singer, Brendan Cleary, Fernando Figueira Filho, and Alexey Zagalsky. The (r) evolution of social media in software engineering. In *Proceedings of the Conference on the Future of Software Engineering*, FOSE 2014, pages 100–116, New York, NY, USA, 2014. ACM.

[128] Margaret-Anne Storey, Christoph Treude, Arie van Deursen, and Li-Te Cheng. The impact of social media on software engineering practices and tools. In *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*, FoSER '10, pages 359–364, New York, NY, USA, 2010. ACM.

[129] Gail M Sullivan and Anthony R Artino Jr. Analyzing and interpreting data from likert-type scales. *Journal of graduate medical education*, 5(4):541–542, 2013.

[130] Didi Surian, Nian Liu, David Lo, Hanghang Tong, Ee-Peng Lim, and Christos Faloutsos. Recommending people in developers' collaboration network. In *2011 18th Working Conference on Reverse Engineering*, pages 379–388. IEEE, 2011.

[131] Y. Tian and D. Lo. An exploratory study on software microblogger behaviors. In *2014 IEEE 4th Workshop on Mining Unstructured Data*, pages 1–5, Sept 2014.

[132] Yuan Tian, Meiyappan Nagappan, David Lo, and Ahmed E Hassan. What are the characteristics of high-rated apps? a case study on free android applications. In *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*, pages 301–310. IEEE, 2015.

[133] Christoph Treude, Ohad Barzilay, and Margaret-Anne Storey. How do programmers ask and answer questions on the web? (nier track). In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 804–807, New York, NY, USA, 2011. ACM.

[134] Jason Tsay, Laura Dabbish, and James Herbsleb. Influence of social and technical factors for evaluating contribution in github. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 356–366, New York, NY, USA, 2014. ACM.

[135] Jason Tsay, Laura Dabbish, and James Herbsleb. Let's talk about it: Evaluating contributions through discussion in github. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 144–154, New York, NY, USA, 2014. ACM.

[136] Bogdan Vasilescu, Vladimir Filkov, and Alexander 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, Washington, DC, USA, 2013. IEEE Computer Society.

[137] Rahul Venkataramani, Atul Gupta, Allahbaksh Asadullah, Basavaraju Muddu, and Vasudev Bhat. Discovery of technical expertise from open source code

repositories. In *Proceedings of the 22nd International Conference on World Wide Web*, WWW '13 Companion, pages 97–98, Republic and Canton of Geneva, Switzerland, 2013. International World Wide Web Conferences Steering Committee.

[138] Lorenzo Villarroel, Gabriele Bavota, Barbara Russo, Rocco Oliveto, and Massimiliano Di Penta. Release planning of mobile apps based on user reviews. In *Proceedings of the 38th International Conference on Software Engineering*, pages 14–24. ACM, 2016.

[139] Georg von Krogh, Sebastian Spaeth, and Karim R Lakhani. Community, joining, and specialization in open source software innovation: a case study. *Research Policy*, 32(7):1217 – 1241, 2003. Open Source Software Development.

[140] Shaowei Wang, David Lo, and Lingxiao Jiang. An empirical study on developer interactions in StackOverflow. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, pages 1019–1024. ACM, 2013.

[141] Stanley Wasserman and Katherine Faust. *Social network analysis: Methods and applications*, volume 8. Cambridge university press, 1994.

[142] Cathrin Weiss, Rahul Premraj, Thomas Zimmermann, and Andreas Zeller. How long will it take to fix this bug? In *Proceedings of the Fourth International Workshop on Mining Software Repositories*, MSR '07, pages 1–, Washington, DC, USA, 2007. IEEE Computer Society.

[143] Chris Welch. Apple pulls ios 8.0.1 after users report major problems with update, sep 2014.

[144] Frank Wilcoxon. Individual comparisons by ranking methods. *Biometrics bulletin*, 1(6):80–83, 1945.

[145] Timo Wolf, Adrian Schroter, Daniela Damian, and Thanh Nguyen. Predicting build failures using social network analysis on developer communication. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 1–11, Washington, DC, USA, 2009. IEEE Computer Society.

[146] Xuchen Yao and Benjamin Van Durme. Information extraction over structured data: Question answering with freebase. In *ACL (1)*, pages 956–966. Citeseer, 2014.

[147] Robert K. Yin. *Case Study Research: Design and Methods - Third Edition*. SAGE Publications, 3 edition, 2002.

[148] Y. Yu, H. Wang, V. Filkov, P. Devanbu, and B. Vasilescu. Wait for it: Determinants of pull request evaluation latency on github. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 367–371, May 2015.

[149] Y. Yu, H. Wang, G. Yin, and C. X. Ling. Who should review this pull-request: Reviewer recommendation to expedite crowd collaboration. In *2014 21st Asia-Pacific Software Engineering Conference*, volume 1, pages 335–342, Dec 2014.

[150] Yue Yu, Gang Yin, Huaimin Wang, and Tao Wang. Exploring the patterns of social behavior in github. In *Proceedings of the 1st International Workshop on Crowd-based Software Development Methods and Technologies*, CrowdSoft 2014, pages 31–36, New York, NY, USA, 2014. ACM.

[151] Marcelo Serrano Zanetti, Ingo Scholtes, Claudio Juan Tessone, and Frank Schweitzer. Categorizing bugs with social networks: A case study on four open source software communities. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 1032–1041, Piscataway, NJ, USA, 2013. IEEE Press.

[152] Jerrold H. Zar. *Spearman Rank Correlation*. John Wiley & Sons, Ltd, 2005.

[153] Feng Zhang, F. Khomh, Ying Zou, and A.E. Hassan. An empirical study on factors impacting bug fixing time. In *Reverse Engineering (WCRE), 2012 19th Working Conference on*, pages 225–234, Oct 2012.

[154] Guoliang Zhao, Daniel Alencar da Costa, and Ying Zou. Improving the pull requests review process using learning-to-rank algorithms. *Empirical Software Engineering*, pages 1–31, 2019.

Appendix A

Chapter 3 Subject Systems

In Chapter 3, the subjects systems used are Firefox and Chrome. We include in Table A.1 more details about the subject systems, including the dates of the releases used in the study, the number of bugs associated to each release, and the number of mapped tweets using our approach.

Table A.1: Descriptive statistics of the studied releases of Firefox and Chrome in Chapter 3.

Firefox				Chrome			
Version	**Date**	**# Bugs**	**# Tweets**	**Version**	**Date**	**# Bugs**	**# Tweets**
5.0	2011-06-21	389	2890	15.0	2011-10-25	86	629
6.0	2011-08-16	345	1744	16.0	2011-12-13	205	623
7.0	2011-09-27	73	2126	17.0	2012-02-08	200	590
8.0	2011-11-08	282	1531	18.0	2012-03-28	197	556
9.0	2011-12-20	265	1364	19.0	2012-05-15	78	527
10.0	2012-01-31	353	817	20.0	2012-06-26	72	780
11.0	2012-03-13	236	956	21.0	2012-07-31	71	878
12.0	2012-04-24	284	953	22.0	2012-09-25	58	759
13.0	2012-06-05	199	788	23.0	2012-11-06	92	902
14.0	2012-06-26	243	3205	24.0	2013-01-10	69	658
15.0	2012-08-28	233	1483	25.0	2013-02-21	169	815
16.0	2012-10-09	238	1263	26.0	2013-03-26	914	1020
17.0	2012-11-20	262	1842	27.0	2013-05-21	703	925
18.0	2013-01-08	240	1049	28.0	2013-06-17	746	798
19.0	2013-02-19	350	1123	29.0	2013-08-20	711	1174
20.0	2013-04-02	329	2406	30.0	2013-09-18	640	1201
21.0	2013-05-14	217	1567	31.0	2013-11-12	830	1310
22.0	2013-06-25	245	1835	32.0	2014-01-14	564	1231
23.0	2013-08-06	287	2362	33.0	2014-02-18	834	1195
24.0	2013-09-17	198	1725	34.0	2014-04-02	679	1154
25.0	2013-10-29	219	1549	35.0	2014-05-20	807	1336
26.0	2013-12-10	278	1381	36.0	2014-07-15	579	910
27.0	2014-02-04	230	1706	37.0	2014-08-26	687	1354
28.0	2014-03-18	226	1929	38.0	2014-10-07	617	958
29.0	2014-04-29	402	3102	39.0	2014-11-12	763	1530
30.0	2014-06-10	284	1100	40.0	2015-01-20	579	1073
31.0	2014-07-22	209	1122	41.0	2015-03-03	656	1055
32.0	2014-09-02	236	958	42.0	2015-04-14	452	1103
33.0	2014-10-14	325	3423	43.0	2015-05-19	708	1538
34.0	2014-12-01	248	1705	44.0	2015-07-21	477	1037
35.0	2015-01-13	329	1878	45.0	2015-09-01	436	1188
36.0	2015-02-24	303	1830	46.0	2015-10-13	422	1078
37.0	2015-03-31	381	1628	47.0	2015-12-01	310	1250
38.0	2015-05-12	369	2144	48.0	2016-01-20	272	891
39.0	2015-07-02	231	1649				
40.0	2015-08-11	378	1294				
41.0	2015-09-22	249	858				

Appendix B

Chapter 5 Survey-Related Material

B.1 Survey Questions

The survey questions are listed in Table B.1.

B.2 Invitation Letter

Dear [Participant],

My name is Mariam El Mezouar. I am a PhD student at Queen's University, Canada supervised by Dr. Ying Zou (ying.zou@queensu.ca). I am inviting you to participate in a survey about developers' chatroom because you are an active GitHub contributor. We apologize in advance if our email is unwanted or a waste of your time.

This study will help us understand how tools like Gitter or Slack support the software development process. There are no mandatory questions in our survey and it will likely take only 15 mins of your time. To participate, please click on this link https://goo.gl/forms/oX4UqWUDRcykBP372.

Please note that it is possible to withdraw your survey submission within 3 weeks,

Table B.1: Survey Questions

#	Question	Answer choices
Section 1: Demographics		
Q1	Do you participate in ...?	[Slack, Gitter, both, none, other]
Q2	What is your gender?	[female, male, other]
Q3	What is the highest computer science degree you have completed?	[student, college diploma, bachelor's degree, master's degree, professional degree, doctorate, self-taught, other]
Q4	What is your current employment status?	[employed full time, employed part time, unemployed, student, self-employe, retired, other]
Q5	What is your overall software development experience?	[1, 2, 3, 4, 5, 6, 7, 8, 9, 10+]
Q13	What is your role in the project or the technology associated to the chatroom team?	[user, maintainer, contributor, no active role, other]
Section 2: Motivations		
Q6	What are your motivations for asking questions on the chatrom?	open-ended question
Q7	Can you tell us about specific cases when the chatroom helped you to solve issues?	open-ended question
Q8	What are your motivations for answering questions on the chatroom?	open-ended question
Q9	Can you tell us about specific cases when you helped to solve issues on the chatroom?	open-ended question
Q10	What types of issues/topics do you mostly discuss on the chatroom?	open-ended question
Q11	When would you use the chatroom rather than other developer platforms?	open-ended question
Section 3: Impacts		
Q12	What is the chatroom team you are most active in? (you can answer the following questions with the selected team in mind)	open-ended question
Q14	If it applies, do you think that the chatroom has an impact on the quality of the associated project or technology? How?	open-ended question
Q15	Do you also think that the chatroom has an impact on the quality of your own projects? How?	open-ended question
Q16	Do you think that the chatroom has an impact on your productivity? How?	open-ended question
Q17	Do you think that the chatroom has an impact on the retention of developers that use the project or the technology?	open-ended question
Q18	If it applies, do you think that the chatroom has an impact on the resolution time of issues in the associated project or technology? How?	open-ended question
Section 4: Quality determinants		
Q19	In your opinion, what makes a good chatroom?	open-ended question
Q20	In your opinion, how can the chatrooom be improved?	open-ended question
Q21	Overall, how would you rate your experience using the chatroom?	[1, 2, 3, 4, 5] (1 being poor to 5 being excellent)

by simply emailing the researchers. The data collected from the survey will be securely stored on a single computer machine at the research lab of the researchers within Queen's university.

To compensate you for your time, you may win a $10 Amazon gift card if you complete the full questionnaire. We are offering these gift cards to 30% randomly drawn participants who complete the questionnaire. We would also be happy to share with you a summary of the survey results by email, if you are interested.

If you have any questions about this survey, or difficulty in accessing the site or completing the survey, please contact Mariam El Mezouar at mariam.el.mezouar@queensu.ca or Daniel Alencar da Costa at daniel.alencar@queensu.ca. Thank you in advance for your time and for providing this important feedback!

Any ethical concerns about the study may be directed to the Chair of the General Research Ethics Board at chair.GREB@queensu.ca or 1-844-535-2988 (Toll free in North America).

DISCLAIMER: We have no affiliation with any business or commercial entities. This is a one time request for your participation in an academic survey. You will not receive any further unsolicited emails from us.

Best Regards,

Mariam El Mezouar

PhD Candidate

Queen's University

B.3 Sample of the Thematic Analysis

Table B.2: Samples of the thematic analysis of the survey responses

Phase 1: Coding	
Survey response	**Extracted codes**
I'm a remote developer and enjoy the social community aspect of the groups and also value the immediate nature of the discussions when trying to solve a problem that generally aren't as fluid in other forums like StackOverflow	Socializing, Immediacy
To make sure people get answers to difficult / frustrating problems when using my software and as a feedback loop for what is and isn't working well in the software.	Help users, Issue Discovery
Sometimes docs are incomplete, ambiguous or too voluminous to read in it's entirety just to get started.	Inadequate documentation
Contributors communicate in real-time about issues/ideas and it's a quick way to gut-check an idea before putting together a full PR	Interactive, Brainstorm features
Yes. I trust in the Angular gitter community to guide me to the right approach in creating my application. This is extremely helpful in a fast growing language/technology where tutorials and resources from even a few months may already be outdated.	Custom feedback, Outdated documentation
Phase 2: Generating themes	
Codes	**Theme**
Socializing	Community
Immediacy, Interactive	Response time
Brainstorm features, Issue discovery	Project Tasks
Outdated documentation, Inadequate documentation	Documentation
Help users, Custom feedback	User support

B.4 Excerpt of Interview Script

Thank you for participating in our survey about the use of chatrooms by developers! Do we have your permission to record the content of this interview for future use? Based on your survey answers, you are a {Slack/Gitter} user and you identify as a {maintainer/contributor/user} of the project associated to the chatroom. I will ask you a series of questions with the goal of clarifying your survey answers, and possibly validating or rejecting other possible answers. Please feel free to go off on

any tangents that you judge relevant to the topic.

(Part 1)

Can you tell us about your motivations for contributing to the chatrooms when asking questions?

Prompts:

Response time?

Quality of the help?

The community of developers?

Can you tell us about your motivations for contributing to the chatrooms when answering questions?

Prompts:

Community building?

Personal gain?

Personal enjoyment?

(Part 2)

Do you believe that {Slack/Gitter} has a noticeable impact on the associated projects in terms of the quality of the code produced, the issue resolution, the productivity of the developers?

Prompts:

Support project awareness?

Prioritization/Visibility of issues and features?

User support?

Improved productivity?

(Part 3)

What are the most important elements that define the quality of a good chatroom, and what could be improved?

Prompts:

Community activity?

Moderation?

Features?

(Examples of tangents)

In your opinion, what are the differences in the usage of the chatrooms by the open communities vs. the corporate teams?

Do you believe that the chatrooms (Slack or Gitter) drive project selection or vise versa? For example, could developers adopt certain projects because the community is already settled and active in Slack? or is it the opposite?

What are the elements to consider by a project team when selecting (or not) a chatroom as a central hub of communication?

Appendix C

Chapter 6 Survey-Related Material

C.1 Survey Questions

The survey questions are listed in Table C.1.

C.2 Invitation Letter

Dear [Participant],

My name is Mariam El Mezouar. I am a PhD student at Queen's University, Canada supervised by Dr. Ying Zou (ying.zou@queensu.ca). I am inviting you to participate in a survey about developers' chatroom because you are an active GitHub contributor. We apologize in advance if our email is unwanted or a waste of your time.

This study will help us understand how channels likes chatrooms or mailing lists support the software development process. The expected outcome of this study is to provide a set of recommendations to the practitioners (i.e. the developers) regarding the most appropriate tools to use to successfully conduct the software projects. There

#	Question	Answer choices
Section 1: Demographics		
Q_1^1	What is your gender?	[female, male, other]
Q_2^1	Where are you from?	open-ended question
Q_3^1	How old are you?	[Under 25, 25-35, 36-45, 46-55, Over 55]
Q_4^1	What is your overall software development experience?	[1, 2, 3, 4, 5, 6, 7, 8, 9, 10+]
Q_5^1	What is the name of the GitHub project you are most active in if applicable?	open-ended question
Q_6^1	What is your role in the project or the technology associated to the chatroom team?	[user, maintainer, active contributor, occasional contributor, *other*]
Sections 2 to 8: Development activities		
Q_1^{2-8}	How familiar are you with the <activity name>process within the project?	[1, 2, 3, 4, 5] (1 being not familiar to 5 being expert)
Q_2^{2-8}	What are the primary communication channels used to communicate about <activity name>?	[Face to face, Email, Mailing lists, Telephone, IRC, Slack, Gitter, Google Hangouts, Skype, Google groups, Stack Overflow, Issue tracking systems, Pull requests, Twitter, Facebook, *Other*]
Q_3^{2-8}	How well are the channels working for this purpose?	[1, 2, 3, 4, 5] (1 being poor to 5 being excellent)
Q_4^{2-8}	Would you like to justify your rating?	open-ended question
Q_5^{2-8}	What are the reasons behind using these communication channels for this purpose?	open-ended question

Table C.1: Survey questions

are no known risks to this study. There are no mandatory questions in our survey and it will likely take only 15 mins of your time.

To compensate you for your time, you may win a $10 Amazon gift card if you complete the full questionnaire. We are offering these gift cards to 30complete the questionnaire. We would also be happy to share with you the results of the survey, if you are interested.

If you have any questions about this survey, or difficulty in accessing the site or completing the survey, please contact Mariam El Mezouar at mariam.el.mezouar@queensu.ca or Daniel Alencar da Costa at daniel.alencar@queensu.ca. Thank you in advance for your time and for providing this important feedback!

Any ethical concerns about the study may be directed to the Chair of the General Research Ethics Board at chair.GREB@queensu.ca or 1-844-535-2988 (Toll free in North America).

DISCLAIMER: We have no affiliation with any business or commercial entities. This is a one time request for your participation in an academic survey. You will not receive any further unsolicited emails from us.

Best Regards,

Mariam El Mezouar

PhD Candidate

Queen's University