UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

EDUARDO WITTER DOS SANTOS

Understanding the Impact of Teams in Modern Code Review

Thesis presented in partial fulfillment of the requirements for the degree of Master of Computer Science

Advisor: Prof. Dr. Ingrid Oliveira de Nunes

Porto Alegre
July 2021

CIP — CATALOGING-IN-PUBLICATION

Witter dos Santos, Eduardo

Understanding the Impact of Teams in Modern Code Review / Eduardo Witter dos Santos. – Porto Alegre: PPGC da UFRGS, 2021.

114 f.: il.

Thesis (Master) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2021. Advisor: Prof. Dr. Ingrid Oliveira de Nunes .

1. Modern code review, recommender systems. I. , Prof. Dr. Ingrid Oliveira de Nunes. II. Título.

In the memory of my mother, Maria Eliza, an authentic, strong, and loving woman who taught me the most valuable lessons in life.

ACKNOWLEDGEMENTS

First, I would like to thank my family for understanding the long hours of absence. They always supported me in this journey, knowing this is something that means a lot to me. Miguel and Raquel, you gave me both inspiration and motivation for this work.

I would also like to thank my advisor Ingrid Nunes for her guidance through the development of this dissertation. During challenging times in my personal life, her empathy and encouraging words were as important as her excellent scientific guidance.

I would like to acknowledge Daniel Pigatto and Ricardo Pianta, who supported this work, believing that academia and industry can mutually benefit when working closer. Also, I must acknowledge that this research would not have been possible without the infrastructure provided by the Federal University of Rio Grande do Sul.

ABSTRACT

Modern code review (MCR) is a practice in which code reviews are performed in a tool-supported, asynchronous, and lightweight way and is widely adopted in the software industry. MCR provides improved code quality and fewer bugs, and it can also positively influence the attitude of developers by creating a sense of collective code ownership. To be effective, it depends on many technical and non-technical factors, such as the size of the code change being reviewed and the reviewer's experience. Moreover, finding good reviewers is critical for the process because code reviews are essentially a collaborative task, which relies mainly on reviewers' contributions. Several studies were already performed in order to understand how code review is affected by different factors, such as the reviewer's experience and properties of the code being reviewed. However, there is limited work considering aspects related to how teams are organized and geographically distributed. Not only there is a lack of studies in this direction, but also this kind of information has not been explored in reviewer recommenders. Instead, existing recommenders are typically based on data obtained from version control systems, which usually do not provide information about team organization, locations, and time zones. In this dissertation, we aim to understand how to improve modern code review by exploiting the team structure within an organization developing software with geographically distributed teams. To achieve this, we investigate code review from different perspectives in three separate studies. In the first study, we mined data from repositories, code review databases, and team structure to investigate how the effectiveness of code review is influenced by team-related factors, such as how teams are organized and geographically distributed. To complement this study, we surveyed developers to investigate how code reviews should ideally happen, what motivates developers to engage in code reviews, and how they interact. Finally, based on the outcomes and findings from these first two studies, we propose, implement and evaluate two code review prediction models that consider team-related information. Our studies show that team-related aspects have a significant impact on code reviews. For instance, code reviews with more teams and locations generally have more contributions from reviewers, but usually take more time to be completed. Moreover, the two proposed code review prediction models are able to largely improve baselines when predicting reviewer participation and amount of provided feedback.

Keywords: Modern code review, recommender systems.

Entendento o Impacto de Times na Revisão de Código Moderna

RESUMO

Na revisão de código moderna (MCR), o código produzido é revisado de forma assíncrona por outros desenvolvedores, com a ajuda de ferramentas, com baixa rigidez na sua execução. Amplamente adotada na indústria, traz benefícios para a qualidade do código, dentre outros diversos benefícios. Para ser efetiva, depende de fatores técnicos e não-técnicos, como a experiência dos revisores e a quantidade de linhas de código alteradas. Além disso, a revisão de código dependente fortemente dos revisores, e portanto um revisor adequado é crucial. Diversos estudos já foram realizados para entender como diferentes fatores afetam a revisão de código. Entretanto, poucos trabalhos consideram a influência da organização dos times e sua distribuição geográfica. Além destas limitações, as técnicas existentes para recomendar revisores não são focadas em cenários com times geograficamente distribuídos. Em vez disso, a maioria destas técnicas é inteiramente baseada em dados obtidos de sistema de controle de versão, que usualmente não fornecem informação sobre times, cidades e fusos horários. Nesta dissertação, nosso objetivo é entender como melhorar a revisão de código moderna considerando a informação sobre a estrutura dos times e sua distribuição. Para isto, nós analisamos atividade de revisão de código de diferentes pontos de vista em três estudos separados. No primeiro estudo, nós mineramos dados de repositórios de código-fonte, bases de dado de revisão de código e estrutura de times para investigar como a efetividade da revisão de código é influenciada por fatores relacionados a times, tais como sua organização e distribuição geográfica. Para complementar este primeiro estudo, entrevistamos desenvolvedores para investigar como a revisão de código deveria acontecer idealmente, o que motiva os desenvolvedores a participar de revisões de código, e como ocorre a interação entre as pessoas neste processo. Por fim, baseado nos resultados destes dois primeiros estudos, nós implementamos e avaliamos dois preditores aplicados a revisão de código, levando em consideração os dados relacionados a times. Nossos estudos mostram que aspectos relacionados a times tem impacto significativo nas revisões de código. Por exemplo, revisões de código com mais times e cidades envolvidos tem em geral mais contribuições por parte dos revisores, porém levam mais tempo para serem concluídas. Além disso, os preditores propostos tem melhoria significativa de performance quando usam informação relacionada a times.

Palavras-chave: revisão de código moderna, sistemas de recomendação.

LIST OF ABBREVIATIONS AND ACRONYMS

MCR	Modern Code Review
DSD	Distributed Software Development
FLOSS	Free, Libre, and Open Source Software

LIST OF FIGURES

LIST OF TABLES

CONTENTS

1 INTRODUCTION

Code review is a common practice adopted in software development to improve software quality based on static code analysis by peers. There are studies that provide evidence that it reduces the number of defects detected in production, mainly when it has adequate code coverage as well as engagement and participation of reviewers (MCINTOSH et al., 2014). Moreover, code review is a recognized way to foster knowledge sharing that benefits authors and reviewers (HUNDHAUSEN; AGRAWAL; AGARWAL, 2013). It also improves team collaboration because it creates collective ownership of the source code, which results from collaborative rather than individual work (BACCHELLI; BIRD, 2013; THONGTANUNAM et al., 2016b). Nowadays, code review is less formal than in earlier decades of software development. In the past, it was typically in the form of code inspections (FAGAN, 1986), which required formal meetings and checklists (KOLLANUS; KOSKINEN, 2009). Today, such a practice is typically more informal, being referred to as *Modern Code Review* (MCR) (BACCHELLI; BIRD, 2013; DAVILA; NUNES, 2021).

MCR is a practice in which developers other than the author of a code change provide feedback before this change is accepted into a project's repository. This is done—with tool support—in an asynchronous and lightweight way while preserving key benefits of code review. As a result, this practice has been used in open-source software (OSS) projects (KONONENKO; BAYSAL; GODFREY, 2016; LI et al., 2017) and in companies such as Microsoft (BACCHELLI; BIRD, 2013) and Google (SADOWSKI et al., 2018).

The effectiveness of code review depends on different factors, and when it cannot provide expected benefits, it becomes a costly and time-consuming task (CZERWONKA; GREILER; TILFORD, 2015; THONGTANUNAM et al., 2016a). For example, if there is a time gap between completing a change and its review by a peer, the author may have its work partially blocked, possibly affecting the whole software release (THONGTANUNAM et al., 2015). This lack of dynamism in the code review activity increases the work in progress for teams, as they start other tasks while waiting for the pending reviews. Furthermore, the context switching between coding tasks and reviews may also have a negative impact on developers' work.

In the context of DSD (distributed software development), which is increasingly common in FLOSS (Free, Libre, and Open Source Software) and proprietary software, code review faces additional challenges. For instance, authors and reviewers might work

on different teams and might have different priorities and be less aware of each other urgency for a pending code review. These and other drawbacks of distribution affect people who work in the same building or on the same floor if they cannot reach each other with a short walk of 30 meters (OLSON; OLSON, 2000). Furthermore, communication during code review can be adversely affected as the geographic distance is usually associated with cultural, political, temporal, language, and organizational differences, as people can work from virtually anywhere and cooperate from their homes or even from other companies.

Although there are studies investigating the factors that positively and negatively affect the effectiveness of code review, this investigation happened only to a limited extent with respect to the influence of team-related aspects. For instance, reviewers are expected to be one of the most impacting factors. Nevertheless, most of the techniques to find suitable reviewers use data obtained from version control systems, basing their recommendations on the commit messages, file names, and authorship of recent changes in the same files. Other techniques use data from code review databases, suggesting reviewers that recently provided more comments in the changed files, for instance. As can be seen, team-related aspects are not considered by these techniques. In this work, we thus propose a technique to recommend suitable reviewers in the context of software developed within organizations with geographically distributed teams. This technique considers the findings, insights, and lessons learned from two studies we conducted to explore how both technical and non-technical factors influence the code review activity.

In the following sections, we detail the problem to be addressed by this research and the gaps of existing work in Section 1.1, and in Section 1.2 we propose a solution. Finally, Section 1.3 presents an outline of the remainder of this dissertation.

1.1 Problem Statement and Limitations of Existing Work

Considering the scenario described above, we derive the following research question: *How to improve modern code review exploiting the team structure within a software organization?* Despite the significant amount of work that has been done in the context of code review, existing studies and proposed techniques have limitations, discussed as follows.

Although several studies were already performed to understand how a set of influence factors affects a set of code review outcomes (THONGTANUNAM et al., 2016b;

BOSU; GREILER; BIRD, 2015; BAYSAL et al., 2016, 2016; BELLER et al., 2014; THONGTANUNAM et al., 2016b; BOSU; GREILER; BIRD, 2015), there is little investigation of the impact of teams and their geographic distribution on code review. Some of the analyzed influence factors are related to the characteristics of authors and reviewers, such as their experience. In contrast, others are related to properties of the change being reviewed, like the number of changed lines and files.

Even though several of these studies have analyzed MCR targeting FLOSS projects, such as OpenStack, Qt, and LibreOffice, which present DSD characteristics, most of them did not investigate the impact of distribution: factors associated with distribution were random variables rather than independent variables. For instance, companies with both co-located and distributed teams were analyzed indistinctly.

Besides these limitations in the studies investigating the effects of team-related aspects, the existing techniques to recommend suitable reviewers—which are important because reviewers are critical for the effectiveness of code review—are not tailored to scenarios with multiple teams, which might be geographically distributed. Instead, most of them do not consider this type of information, as they are entirely based on data from version control systems, which usually do not provide information about team organization, locations, and time zones.

Furthermore, these techniques to find suitable reviewers are not based on studies about what positively or negatively influences code review, as they evaluate whether a few heuristics are capable of recommending the same reviewers that authors would have selected. For instance, there are studies that base their recommendations on previous reviews of files with similar file paths or with similar commit messages (THONGTANUNAM et al., 2014; XIA et al., 2015), which are properties obtained from version control systems. Similarly, other studies use data from code review databases, such as who has provided more comments in previous reviews in the same file, or even based on who has participated in more reviews in the same files (ZANJANI; KAGDI; BIRD, 2016).

In summary, neither are there studies investigating how team-related aspects affect code review nor have these aspects been taken into account to recommend suitable reviewers in distributed scenarios.

1.2 Proposed Solution and Overview of Contributions

In this work, we aim to improve MCR by considering team-related information in the context of software developed within organizations, where teams might be in different locations. As mentioned in the previous section, there is limited work evaluating how these aspects influence code review, and no technique considers team-related data to recommend reviewers.

Thus, we first conducted two studies to have a better understanding of this matter. We focused on the influence of team-related factors, such as the number of different teams and cities involved in code reviews. However, factors like patch size (LOC) are also be evaluated because several previous studies demonstrated their relevance, suggesting that their effects need to be compared with those of team-related factors. Then, with the findings, knowledge, and lessons learned from these studies, we implemented and evaluated code review predictors to help developers find a reviewer for a given code change. In Figure 1.1 we present the development workflow of the proposed solution, which is detailed next.

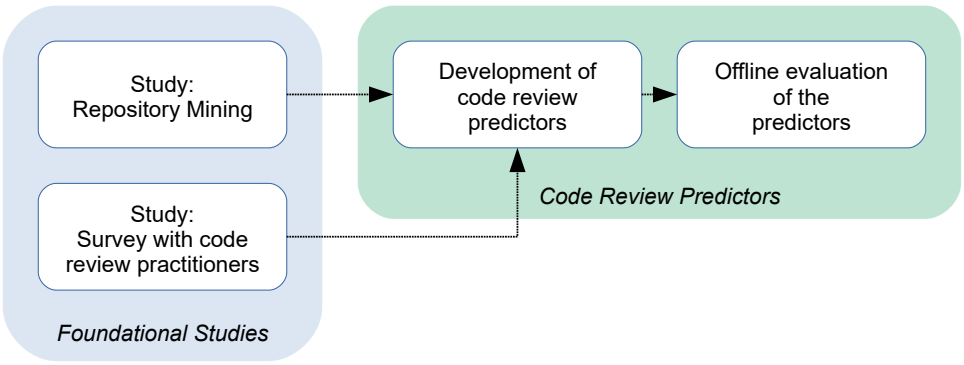


Figure 1.1 – Workflow of the proposed solution

The first study is observational and based on data mined from software repositories, code review databases, and detailed information about developers and managers (their teams, locations, and hierarchy) of a large project with DSD characteristics. It investigates how a set of code review outcomes is affected by a set of influence factors. More specifically, this investigation consists of analyzing the influence of the number of teams and locations and the influence of the patch size (LOC), and the number of active reviewers to allow us to compare their impacts in the same context. Four code review outcomes are analyzed: duration (days), the participation of reviewers (proportion of invited reviewers that actually participate), comment density (number of comments considering

the number of changed lines), and comment density by each active reviewer. The analyzed code review outcomes are indicators of code review effectiveness, as code reviews are expected to have a reasonable duration to allow a proper analysis of the change by a significant proportion of the invited reviewers with a fair amount of interaction with the authors.

The second study consists of a survey that allows us to compare the results of the first study with a subjective perception from software developers with relevant experience in code review, both as authors and reviewers. Moreover, the survey also aims to understand the perceived benefits and drawbacks of adopting code review, how people interact in this process, and what motivates the reviewers.

Based on the findings from these studies, we implemented and evaluated two code review predictors. These predictors consider data from version control systems, code review databases, and a database with administrative information (manager, location, and team for each project member). The first predictor aims to respond to whether a given developer allowed to review the code change will actually engage in the review, considering any existing access control and permissions associated with the software repository. For developers predicted to participate in the code review, a second predictor estimates how much feedback will be provided during the code review. We performed an offline evaluation of these predictors with separated training and test datasets.

1.3 Outline

The remainder of this dissertation is organized as follows. In Chapter 2, we present a background on code review, including the various forms in which it can be adopted in software projects. In Chapter 3, we examine the existing work investigating how technical and non-technical factors affect MCR and the existing approaches to recommend reviewers. In Chapters 4 and 5, we present two foundational studies that allowed us to understand how MCR is affected by team-related factors in real projects. Based on these two studies, in Chapter 6 we present and evaluate two code review prediction models. Finally, in Chapter 7, we conclude and detail future work.

2 BACKGROUND

Different practices have been adopted in the software industry to perform a manual inspection of source code to achieve better software quality and other benefits. Some practices are more structured and formal than others and thus require more preparation and orchestration. This chapter introduces code review practices and compares them considering the list of prescribed roles, the flexibility of roles, the need for a preparation phase, and participants to be available simultaneously. We start by presenting formal code inspection, which was the first practice widely adopted in the industry. Then, we present other typical code review practices, namely code walkthrough, pair programming, and modern code review, ordered according to the formality of the inspection process.

2.1 Formal Code Inspection

Formal code inspection is a highly structured practice to manually review source code, usually with rigid roles and sequences of phases and was first described in the pioneering work of Fagan (1976). The inspection process must examine all produced source code and should happen regularly. Its major goal is to find defects.

Fagan's method starts with a *planning* phase, when all artifacts to be reviewed are checked for minimum requirements and people's agenda are verified for availability. It is followed by an *overview* phase, when the roles are assigned to the inspection team members: author, tester, reader, and moderator, and when all artifacts are presented to them. Every team member receives specific training for a given role prior to the whole process. Next, in the *preparation* phase, the group carefully studies the artifacts to be inspected, so they can finally find defects in the *inspection* phase. The author will fix them during *Rework* phase and present the final code to the moderator in the *follow- up* phase. This approach started at IBM and then became widely adopted in the industry (FAGAN, 1986), receiving significant attention from the scientific community.

Further variations of the original Fagan's method emphasized or even eliminated specific phases or roles. For instance, the *active design review* method (PARNAS; WEISS, 1985) uses smaller review batches, each one with limited scope and more specialized reviewers with just three phases (overview, preparation, and inspection), rather than the six phases described earlier. Bisant and Lyle (1989) suggested that small organizations or teams could be more productive using a two-person inspection method, in which only

author and reviewer roles are present—but still preserving the same phases of the original method. Positive results were presented by an n-fold inspection method (MARTIN; TSAI, 1990) in which multiple smaller teams work in parallel to inspect the same source code, detecting more defects when compared to the original Fagan's method. However, it is justifiable only for the early phases of mission-critical projects due to its much higher cost.

According to a survey of software inspection research (KOLLANUS; KOSKINEN, 2009), formal code inspections received less attention from researchers after 1990. A more recent experience report (MEYER, 2008), however, described a project with structured inspection phases that resemble formal code inspections.

2.2 Code Walkthrough

Formal inspection is this process composed of many phases described above. A form of code inspection that reduces the number of required phases is the so-called *code walkthrough* (IEEE-1028, 2008), which prescribes three roles: an author, a recorder, and a walkthrough leader. During the inspection process, the author (a developer) presents his work in a meeting conducted by a walkthrough leader. At the same time, a designated participant records the defects, flaws, decisions, and action items. The author is allowed to be the walkthrough leader. Usually, a code walkthrough aims at defect hunting, audit, and knowledge sharing to its audience.

A code walkthrough usually happens on demand to examine specific parts of the source code (FAGAN, 1986), and thus are considered less rigid than formal code inspections. However, it also clearly defines the roles of each participant in the process. A practice named Structured Walkthrough (YOURDON, 1979) is similar to Fagan's method for formal code inspection (AURUM; PETERSSON; WOHLIN, 2002), requiring preparation, walkthrough, and rework phases.

As with formal code inspections, this practice received less attention from researchers after 1990 (KOLLANUS; KOSKINEN, 2009). However, it is still part of IEEE Std. 1028, an IEEE Standard for Software Reviews and Audit, updated in 2008.

2.3 Pair Programming

Requiring fewer roles than Code Walkthrough, *pair programming* involves only two developers—a driver and a navigator—who work together on the same machine, designing, implementing, and testing the same piece of software. The driver and navigator roles are regularly swapped. The navigator provides the specification on what code is to be written and should explain its decisions, and reviews the code typed by the driver, who should pay attention to the specifications and offer alternative specifications. Whenever a disagreement occurs, the solution proposed by the navigator should prevail and be implemented to test its correctness. Besides the mutual technical surveillance, peers should also ensure that both are focused and with a high attention level. Pair programming is part of Extreme Programming (BECK, 2000), where Beck first coined this term. As in other software inspection practices, there is evidence that this practice can improve the overall quality of produced software, from design to implementation, with known benefits for knowledge sharing and mutual trust among team members and long-term maintainability (RADERMACHER; WALIA, 2011). Given that distributed development has been increasingly adopted, there is a tool-assisted variation of pair programming (namely, *Distributed Pair Programming*). It uses regular screen sharing tools or specific distributed IDE that can enforce the practice and allow concurrent editing (SCHENK; PRECHELT; SALINGER, 2014).

2.4 Modern Code Review

The last form of code inspection we present, which is the focus of this work, is *Modern Code Review*. It can be defined as an informal, tool-based approach to make source code inspection part of software development (BACCHELLI; BIRD, 2013). It is based on authors receiving feedback from reviewers using specific tools usually integrated with version control systems. This activity, for authors and reviewers, occurs asynchronously, typically interleaved with other activities.

It is a well-established practice (BACCHELLI; BIRD, 2013) in several open source and proprietary projects (BALACHANDRAN, 2013; RAHMAN et al., 2016a). It is a lightweight, ad-hoc approach to allow feedback from reviewers to authors, defect identification, and knowledge sharing, hopefully providing most of the benefits of formal inspections at a more reasonable cost when compared to formal inspections. For ex-

ample, formal meetings and structured reviews with checklists are known to be costly without improving the number of reported defects (MILLER; WOOD; ROPER, 1998; SABALIAUSKAITE; KUSUMOTO; INOUE, 2004), so these elements are typically not adopted in MCR. Typically, code review tools are integrated with version control systems (GITHUB, 2017; GOOGLE, 2017a) and can even enforce the practice by blocking integrating a piece of code to a repository without being reviewed. In order to reduce the human effort on code review, MCR is often aided by automated reviewers to provide feedback based on configurable quality criteria, like coding standards, static analysis, and unit testing (PANICHELLA et al., 2015; BALACHANDRAN, 2013). The lack of a formal process can be seen at the same time as a drawback as reviewers adopt different quality criteria. Moreover, code review happens in parallel with other development activities without a designated timebox, so reviewers are not always available to get involved and execute a comprehensive analysis of the source code, often paying more attention to non-functional issues (MÄNTYLÄ; LASSENIUS, 2009). These and other adverse effects are more or less visible depending on several factors, which are the object of recent studies detailed next.

2.5 Final Remarks

This chapter presented four existing approaches for code review, which are compared and summarized in Table 2.1. The following comparison criteria are adopted: (i) *synchronous*, which indicates if all participants must be available at the same time during some phase; (ii) *requires preparation*, which indicates if a preparation phase is mandatory; (iii) *roles*, which lists the prescribed roles; and (iv) *flexible roles*, which indicates if roles can be changed or accumulated. Approaches with a preparation phase and with more roles can be considered more formal than the others. Having presented a background on relevant code inspection practices, in the next chapter, we examine the research work that has been done and is related to this dissertation.

Table 2.1 – Comparison of code review approaches.

Approach	**Synchronous**	**Requires Preparation**	**Roles**	**Flexible Roles**
Formal inspection	Yes	Yes	4: author, tester, reader, moderator	No
Code walkthrough	Yes	Yes	3: author, recorder, leader	Yes
Pair programming	Yes	No	2: driving, navigator	Yes
Modern code review	No	No	2: author, reviewer	Yes

3 RELATED WORK

Our work aims to improve code review in the context of geographically distributed teams developing software within organizations. In this chapter, we introduced previous research work that is related to our research problem. We first discuss related work that explores how technical and non-technical factors influence code review in Section 3.1. Given these factors are usually considered to implement reviewer recommenders, we present in Section 3.2 existing approaches to find suitable reviewers for a given code change.

3.1 Investigation of Factors Influencing Code Review

It is fundamental to understand the factors that influence the effectiveness of code review so that existing code review practices and reviewer recommenders can be improved. Therefore, many studies focus on providing a deeper understanding of code review and its influence factors (e.g., the number of changed lines of code and experience of individuals) and outcomes (e.g., duration and discussion among reviewers). To ease that analysis, we first discuss those that evaluated the influence of technical factors and then present existing work about the influence of non-technical factors. Finally, we compare these studies.

3.1.1 Investigation of Technical Factors

Considering technical factors, the relationship between different factors and outcomes has been studied. A study conducted by Thongtanunam et al. (2015) provided evidence that reviewers are less rigorous and find fewer defects on files with a high incidence of defects in the past, focusing on superficial aspects, such as coding standards rather than on functional aspects. In a more recent study (THONGTANUNAM et al., 2016a), the same authors identified that bug fixes typically receive the first feedback faster than implementations of new features. Moreover, they reported that changes with detailed and explanatory commit messages have lower stale rates, while those poorly described receive less attention from reviewers. Focusing on the code review duration (in working days), a few influence factors were investigated. Bosu, Greiler and Bird (2015) concluded that

the patch size influences the duration in most of the analyzed cases, while task priority in the release plan and the affected software components have only occasionally influenced some of the projects analyzed by Baysal et al. (2016).

3.1.2 Investigation of Non-technical Factors

Non-technical factors also received attention recently. As stated by Czerwonka, Greiler and Tilford (2015), the social network that naturally emerges inside the companies or projects should be considered as well as the specific reviewers' skills and their availability and willingness to review. An analysis of the social network of three open-source projects (YANG, 2014) revealed that the most active reviewers have central roles in the social network of those projects and are frequently some of the most significant contributors. Bosu, Greiler and Bird (2015) observed, in a particular organization, that 75% of the code review feedback comes from members of the author's team but is slightly less useful than those from other teams. Baysal et al. (2016), in turn, pointed out that when multiple organizations contribute to the same project, the code review can take more time to be completed and have higher rejection rates depending on which organization is authoring or reviewing a patch.

The experience of the author has also been pointed out as relevant in code review. Senior members of the company and those with recognized expertise usually receive more priority, faster, and more detailed feedback, enabling a faster code review with better results for the quality (BAYSAL et al., 2016; RAHMAN et al., 2016a). The experience of the reviewers is important as well, based on results of the investigation of large company (BOSU; GREILER; BIRD, 2015)—the quality of provided feedback increased during the first year in the company and then stabilized in a *plateau*.

Thongtanunam et al. (2016a), in their study involving three large open-source projects, also investigated non-technical factors, focusing on how the code review was affected by prior events on the files under review. Their conclusions are (1) files that received slow initial feedback in the past will also likely slow initial feedback in the future; (2) files with more authors and reviewers in the past receive more attention; and (3) the number of changed files, directories and the length of the commit message is also important.

3.1.3 Summary of Investigated Factors

Given that many factors that influence code review have been investigated, we summarize what previous studies analyzed in Table 3.1. Rows in this table consist of the examined influence factors, while columns represent the analyzed outcomes associated with code review. In cells, we list the studies that focused on the relationship between a given influence factor and outcome.

As depicted in Table 3.1, most works analyze a single code review outcome or a single influence factor. A notable exception is a work conducted by Thongtanunam et al. (2016a), which analyzed six influence factors and three code review outcomes. The variety of influence factors and code review outcomes suggests that code reviews are an inherently complex activity, affected by both technical and non-technical factors. Similarly, the number of analyzed outcomes suggests that the characteristics of a good code review are not a consensus.

Another relevant conclusion is that, although many studies have analyzed projects with distributed software development (DSD), none of them analyzed DSD-related influence factors or outcomes. For instance, McIntosh et al. (2014) presented a case study of the QT, VTK, and ITK projects, which have distributed teams and several participating companies. The authors of that study evaluated how code review coverage and participation affect the number of defects detected after the software release, without considering how these projects were affected by their distribution of teams, locations, and participating companies. Similarly, a comprehensive analysis was performed by Baysal et al. (2016) for WebKit and Blink, both distributed projects with contributions from several organizations. The authors of that study evaluated how technical and non-technical factors can influence the acceptance rate and total life cycle time of a given patch, from submission to integration. Again, no DSD-related factors were analyzed, and the authors highlighted that more work remains to be done to understand how the geographic distribution of developers affects code review.

3.2 Code Reviewer Recommenders

In peer code review, the collaboration of invited reviewers is critical for its effectiveness, making the selection of suitable reviewers an essential part of the process (BACCHELLI; BIRD, 2013; RIGBY; BIRD, 2013). When a suitable reviewer is not found, 4%

Table 3.1 – Related work: Summary of analyzed influence factors, code review outcomes and how the first are related to the latter.

Influence Factors	Absence of Discussion	Comment Usefulness	Feedback Delay	Particip. in Review	Post-release Defects	Review Duration	Review Iterations	Review Quality
Amount of Previous Defects	♠		♠	♣				♣
Affected Modules					♢			
Author's Experience				♠	♢			
Author's Company					♢			
Bug Fix or New Feature			♠			†		
Commit Message Size	♠							
External Reviewers		♡						
Length of Prior Discussions	♠			△				
Number of Authors	♠		♠					
Number of Reviewers	♠							
Patch Size (Files)	♠	♡						
Patch Size (LOC)					♢	†		
Prior Feedback Delay			♠					
Priority					♢			
Review Coverage				△ ⋆				
Review Speed				△				
Reviewer Experience		♡			♢			
Source Code Type		♡						

♣: (THONGTANUNAM et al., 2015)
♠: (THONGTANUNAM et al., 2016a)
♢: (BAYSAL et al., 2016)
♡: (BOSU; GREILER; BIRD, 2015)
†: (BELLER et al., 2014)
△: (MCINTOSH et al., 2014)
⋆: (SHIMAGAKI et al., 2016)

to 30% of the changes are subject to excessive delays, as demonstrated by Thongtanunam et al. (2015). Several studies evaluated different techniques to recommend reviewers in the context of Modern Code Review, with a wide range of features considered in the recommendation.

In the following sections, we present techniques to recommend suitable reviewers, split into four groups based on the criteria used to link reviewers and the code to be reviewed. The *code change experience* group considers the experience as an author of changes on the same code or similar code; thus, someone who authored several changes in a given file is supposed to be a good reviewer when someone else changes it. The *reviewing experience* group considers the previous experience as a reviewer for the same code or similar code; thus, someone who frequently provided comments for changes in a given file is supposed to be a good reviewer when a new change needs to be reviewed. The *collaboration history* group considers that the number of previous interactions between a given pair of developers indicates that they are likely to collaborate well during a code review. Finally, the *other approaches* group includes approaches that explore alternative information to recommend reviewers.

3.2.1 Approaches Based on Code Change Experience

Recommenders based on code change experience consider that the most relevant reviewers are those that *recently* changed the code to be reviewed, i.e., they consider the experience as an author to suggest reviewers. Balachandran (2013) proposed two techniques, RevHistRECO and ReviewBot. They both use information about previous code reviews to recommend reviewers. For each possible reviewer, the experience as author or reviewer is equally valued (code change and reviewing experience, respectively). We next examine both techniques.

RevHistRECO (BALACHANDRAN, 2013) considers the list of changed files, and for each file, it considers the last closed code review in which this file has been changed. Thus, a list of code reviews is derived from the list of changed files. As each code review has an author and a set of reviewers, there is also a list of users associated with the changed files, which are considered as the possible reviewers. Then, the possible reviewers are ranked considering the number of contributions (as author or reviewer) in the closed code reviews of the changed files. Finally, the top three reviewers are recommended.

According to the authors of both techniques, *ReviewBot* (BALACHANDRAN,

2013) has significantly improved the accuracy of *RevHistRECO*—in the same scenario and conditions—by considering the line change history for the lines in the patch to be reviewed, instead of the file change history. As in RevHistRECO, a list of users is obtained from all closed code reviews associated with the changed lines. Again, both authors and reviewers of previous code reviews are considered, and source files are considered more relevant than resource files. However, ReviewBot's algorithm used to rank reviewers considers a decaying factor that gives more value to recent code reviews.

RevHistRECO and ReviewBot did not consider to which file a given comment was provided. Thus, if a given comment is related to a specific file, it is considered for all modified files in the same change. Moreover, as the analyzed change history considers only the file or its lines, both techniques fail to provide recommendations for newly created files. These limitations are tackled by several techniques presented in the next sections.

3.2.2 Approaches Based on Reviewing Experience

Modern Code Review is usually assisted by specific tools that are integrated with version control systems. These tools provide us with a database with code reviews, comments, and other information, allowing a deeper understanding of how code review takes place. In this section, we present techniques that use this information to obtain the experience of candidate reviewers.

Thongtanunam et al. (2014) proposed *RevFinder*, a technique that considers that files with similar locations and names are usually reviewed by the same developers. For each changed file in a given code review, this technique uses a File Path Similarity (FPS) metric to find closed code reviews for similar files and get the names of the reviewers that contributed with comments. Reviewers are then ranked, considering the number of previous reviews of files with similar paths, with a decaying factor to value more recent experiences; the degree of similarity between file locations is also considered for ranking reviewers. RevFinder is a notable technique, as it was the comparison baseline for several other techniques, which are detailed next.

WRC and *ProfileBased*, proposed by Hannebauer et al. (2016) and Fejzer, Przymus and Stencel (2018), respectively, are similar to RevFinder but use different metrics to compute the file path similarity. As RevFinder, WRC and ProfileBased consider only the file names to compute a similarity value; they perform better for projects with more strict

naming rules, such as the Linux Kernel and Android Open Source Project.

An improved similarity function is used by TIE (XIA et al., 2015). This technique considers a modified FPS alongside the similarity of change description (known as commit title and message) to find similar changes and then rank the potential reviewers. An independent prediction model handles each characteristic, and each model suggests a set of recommended reviewers. Then, an empirical coefficient is determined to weigh both models' suggestions and create a single ranked list of suggested reviewers.

A technique named *cHRev* was proposed by Zanjani, Kagdi and Bird (2016). Given a code change to be reviewed, it considers the code review history for the specific files that were changed to provide recommendations. The suggested reviewers are ranked considering their experience with each changed files, which is computed using three factors: a) the total number of comments to each file, b) the total number of different workdays on which the comments were provided to each file, and c) the most recent workday on which a comment was provided to each file. These factors can be seen as indicators of review quantity, frequency, and recency, respectively. cHRev used RevFinder as its baseline, but while RevFinder considers the number of *reviews* of similar files, cHRev uses the number of *comments* provided for the same files in previous reviews.

While several techniques mentioned before use file paths or commit messages to find similar code reviews, a technique named *CORRECT* (RAHMAN et al., 2016a) suggests reviewers based on their experience as reviewers of other changes that involve the same *technologies* or use the same *external libraries*. The dataset used to determine the reviewer's experience includes reviews in other projects. This technique was implemented as a browser plugin that communicates with a server that runs the recommendation algorithm and accesses the code review database to obtain the necessary information.

A technique named *PR+CF* was proposed by Xia et al. (2017), and it recommends reviewers using a hybrid model. Initially, it considers a matrix of all reviewers (columns) and all completed reviews (rows). Each cell contains the number of comments the reviewer provided for that specific review, with a decaying factor to value the recency of comments. As that matrix is very sparse, collaborative filtering techniques are used to fill the void cells, based on data from changes with similar file paths. Then, the matrix is used to create a hybrid model that uses a latent factor model (KALMAN, 1996) and a neighborhood model (KOREN, 2008), which are machine learning algorithms.

The techniques presented in this section provided a substantial improvement over those presented in Section 3.2.1, as they used more sophisticated metrics to model the

reviewers' experience. However, none of them consider the emerging social network among developers or how much a specific pair of developers have interacted in the past, which is important as code review is essentially a collaborative task. In the next section, we present techniques that consider the collaboration history to suggest suitable reviewers.

3.2.3 Approaches Based on Collaboration History

Developers regularly collaborate during code reviews, providing comments and feedback, creating an emerging social network. Some code review platforms such as Github provide even built-in social network mechanisms, like following other developers, being followed by them, participating in multiple teams and organizations. Explicit or emergent, social networks influence code review (CZERWONKA; GREILER; TILFORD, 2015). In this section, we present techniques that take into account this information.

Wang, Yin and Ling (2014) proposed *CN*, a technique that considers the previous collaboration among developers to suggest reviewers. In this context, the comments provided or received during previous code reviews are considered a form of collaboration. The authors use a decaying factor to value the recency of collaboration, assuming that recent collaboration is more relevant. Collaboration diversity is also considered; for instance, five comments provided in the same review are less valuable than one comment in five different reviews. Collaboration data is used to create a weighted graph of developers, representing the comment network (CN) among them. Then, candidate reviewers that are more related to the author receive a better score.

RevRec was proposed by (OUNI; KULA; INOUE, 2016) and is based on reviewers' experience and collaboration obtained from past reviews. The experience considers the number of comments (the total number of review comments) and their recency (date of the last comment) for all files with similar paths. Collaboration, in turn, is measured by the number of comments a developer provided to another developer during a code review. A genetic algorithm is used to combine both experience and collaboration and provide a list of recommended reviewers.

YING et al. proposed in 2016 a technique named *EARec*, which provides recommendations based on authority and experience of the reviewers. Reviewers' experience is based on the number of comments provided in the past in reviews with similar commit messages and changes in files with similar paths. The authority is determined by creat-

ing a graph that contains the possible reviewers; the edge between two nodes exists if the corresponding reviewers have provided comments in the same change (a co-reviewed change), and the weight of each edge is the number of co-reviewed changes. The degree of centrality of each possible reviewer is a representation of her authority.

CoreDevRec was proposed by Jiang, He and Chen (2015) to find a specific type of reviewer: a core member developer. This role is usually played by an experienced developer who has the authority to approve or reject changes proposed by contributors. Many features are considered to recommend a core member, like the changed file paths and the social relations among contributors and core members (following/followed). Also, the total number of prior changes from the same author that were evaluated and submitted by a given core developer is considered as an indication that this specific core developer likes to evaluate contributions from that author. The activeness of core developers is also considered, being derived from several metrics, such as the number of recently reviewed changes and the average delay for the first feedback on recently reviewed changes. These features are combined to create a model that predicts the probability of a given reviewer to actually contribute to a specific code review; reviewers are ranked according to this probability.

The techniques presented in this section take into account metrics supposed to represent how developers, as authors and reviewers, collaborate. Except for CoreDevRec, which is based on native social network mechanisms of Github, all techniques consider the number of exchanged comments or the number of co-reviewed changes as metrics.

3.2.4 Other Approaches

The previously presented approaches presented information in three particular directions: code change experience, reviewing experience, and collaboration history. Jeong et al. (2009), instead, used a broader range of features, which are considered to create a prediction model. This model is used to predict the probability of a given reviewer to contribute to a given change. Candidates are then sorted, and those with higher probabilities are ranked first. This technique considers the author's name as one of its features, alongside several features related to the patch content (such as the number of changed lines of code, the number of changed files, the names of the changed files, and the complexity of the change). Moreover, when an issue tracker entry can be associated with the code under review, its priority and severity are extracted and considered as features. The prediction

model provides an accuracy similar to that of recommender systems used to assign bugs to developers, which is a different problem, as the authors of that work remarked.

3.2.5 Discussion

Given that we introduced the main techniques to recommend suitable reviewers in an MCR context, we now proceed to their comparison. Table 3.2 contains a summary of the presented techniques, organized in chronological order of publication, with their adopted recommendation criteria. The main recommendation criteria are code change experience, reviewing experience, and collaboration history. Some techniques take into account the exact content that was added, changed, or removed, such as the number of statements (such as *if*, *else*, and *for*), so we add a fourth criterion, names *patch content*. Complementary, Table 3.3 provides detailed information about the features considered by each technique. Features related to the patch's content under review, such as its complexity and used APIs and libraries, are considered only by Bayesian Network and CORRECT. Most techniques use features related to the reviewing experience and collaboration history and often adopt a decaying factor to value more recent information. Features related to code change experience are used only by RevHistRECO (BALACHANDRAN, 2013) and ReviewBot (BALACHANDRAN, 2013).

Table 3.2 – Recommender systems: Chronology and recommendation criteria.

Technique Mnemonic	Work	Recommendation criteria				
Bayesian Network †	Jeong et al. (2009)					P
RevHistRECO	Balachandran (2013)	M	R			
ReviewBot	Balachandran (2013)	M	R			
CN	Wang, Yin and Ling (2014)		R	C		
RevFinder	Thongtanunam et al. (2015)		R			
TIE	Xia et al. (2015)		R			
CoreDevRec	Jiang, He and Chen (2015)		R	C		
cHRev	Zanjani, Kagdi and Bird (2016)		R			
CORRECT	Rahman et al. (2016a)		R			P
RevRec	Ouni, Kula and Inoue (2016)		R	C		
EARec	Ying et al. (2016)		R	C		
WRC	Hannebauer et al. (2016)		R			
PR+CF	Xia et al. (2017)		R			
ProfileBased †	Fejzer, Przymus and Stencel (2018)		R			

†: mnemonics were not defined by the authors of the corresponding work.

Recommendation criteria: M: code change experience; R: reviewing experience; C: collaboration history; P: patch content;

In terms of their implementations, from the presented techniques, all but four (JIANG; HE; CHEN, 2015; RAHMAN et al., 2016a; WANG; YIN; LING, 2014; JEONG et al., 2009) follow a three-step approach. First, they identify a set of similar code reviews that have already been completed; the similarity is often computed using features such as the file names and commit messages. Second, candidate reviewers are ranked using a set of metrics supposed to represent the likelihood of being an adequate reviewer. Third, the top-k ranked reviewers are recommended.

Regarding the evaluation of these techniques, Table 3.4 presents for each technique: the adopted comparison baseline, the set of metrics used in the evaluation, and the number of evaluated projects. Most techniques were evaluated using open source projects, and only two (ZANJANI; KAGDI; BIRD, 2016; RAHMAN et al., 2016a) were evaluated exclusively on closed source projects. Moreover, the metrics considered in the evaluation of each technique are not uniform. However, *precision* and *recall* are present

Table 3.3 – Related work: Recommender systems and their features.

Features	Techniques													
Name	**Bayesian Network**	**RevHistRECO**	**Review Bot**	**ProfileBased**	**RevFinder**	**TIE**	**CoreDevRec**	**cHRev**	**CORRECT**	**RevRec**	**EARec**	**WRC**	**PR+CF**	**CN**
Same files		●	●					●						
Similar file paths	●			●	●	●	●			●	●	●	●	
Similar commit message						●					●			
N°of changed files	●													
Complexity of changes	●													
Used technologies and libs									●					
Severity/Priority of task	●													
N°of reviews as author	●	●	●^{t}											
N°of reviews as reviewer		●	●^{t}		●^{t}		●^{t}				●	●^{t}		
N°of comments								●		●			●^{t}	
N°of comments in co-review							●			●	●			●^{t}
N°of comments to author							●							●^{t}
N°of comments from author														●^{t}
Frequency of comments								●						
Recency of comments				●				●		●				
Delay of initial feedback							●^{t}							
Social network with author							●							

t: technique considers a decaying factor to value more recent data

Table 3.4 – Recommender systems evaluation: Baselines, metrics and projects.

Technique	Evaluation baselines	Evaluation metrics	Evaluated projects
Bayesian Network	None	K	2
RevHistRECO	None	K	2 ♣
ReviewBot	RevHistRECO	K	2 ♣
CN	Most active developer	P R	6
RevFinder	ReviewBot	K M	4
TIE	RevFinder	K M	4
CoreDevRec	RevFinder	K M	5
cHRev	RevFinder, xFinder	P R M F	5 †
CORRECT	RevFinder	K P R	16 †
RevRec	cHRev, RevFinder, ReviewBot	P R M	3
EARec	Most active developer	P R	9
WRC	RevFinder	K	4
PR+CF	RevFinder, TIE, CN, CodeDevRev	P R	5
ProfileBased	RevFinder, ReviewBot	P R M F	4

Evaluation metrics:
K: top-k accuracy
P: precision
R: recall
M: Mean Reciprocal Rank
F: F1 score

Evaluated projects:
†: mixed open source and closed source projects
♣: closed source projects

in most recent studies, and half of them use *mean reciprocal rank* (CRASWELL, 2009).

All presented techniques used offline evaluations, where code reviews inside a given time frame are considered as the training dataset, and code reviews inside a subsequent time frame are considered as the test dataset. Thus, a suggested reviewer is considered correct if the recommended reviewer reviewed the code. This type of evaluation has limitations, as it hinders the understanding of how the recommendations affect the code review process and the experience of the users of the recommender systems, as pointed out by (KOVALENKO et al., 2018).

3.3 Final Remarks

In this chapter, we presented studies regarding how technical and non-technical factors influence code review. We evidenced the existing gaps in the understanding of how team-related factors influence code review effectiveness. Naturally, reviewers are critical to the code review process and are likely to have a strong influence on its result—that is, effective code reviews are not possible without adequate reviewers. Finding a suitable

reviewer for a given change is a relevant challenge, as code review is a collaborative task, thus mixing technical skills, availability, and willingness to collaborate. In this context, we presented several existing techniques to find suitable reviewers, which are mostly based on code change experience, reviewing experience, and collaboration history. Again, team-related information, such as how developers are grouped in teams that can or cannot be in the same location, are not considered. Based on the analyzed related work, we conclude that there are opportunities to understand how team-related factors affect code review. Similarly, there is space to improve the recommenders of reviewers by taking into account team-related features.

4 QUANTITATIVE STUDY: MINING CODE REVIEW DATABASES

In this chapter, we present a foundational study that explores how both technical and non-technical factors influence the effectiveness of code review in the context of multiple teams that are geographically distributed. We investigated the relationship between four influence factors — namely number of changed lines of code, involved teams, involved locations, and active reviewers—and four review outcomes that can be seen as an indication of the review effectiveness, such as its duration and number of comments. We next provide details of our target project in Section 4.1, describing its code review process. Next, we describe our study settings in Section 4.2. We present and analyze obtained results in Section 4.3, which is followed by a discussion in Section 4.4.

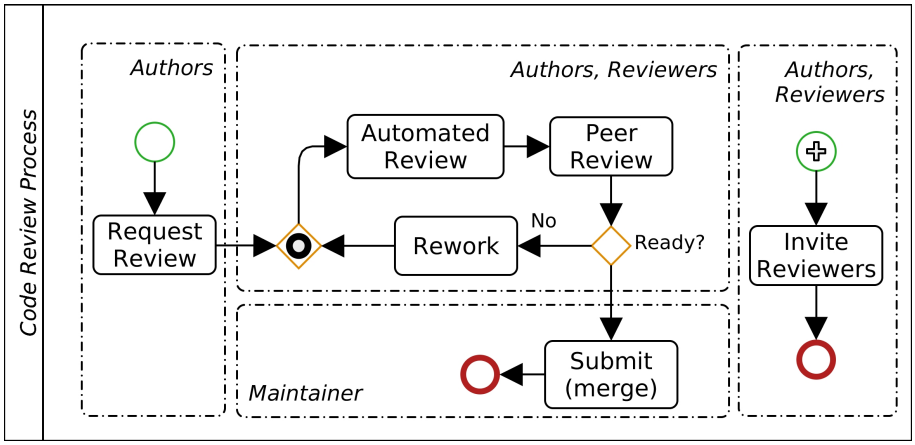
4.1 Study Subject

This is study is based on the analysis of data collected from a single (commercial) software project. Due to its size, we were able to collect a large amount of information regarding its code review. We next describe the code review process of the project and provide details about the collected data. No further information can be given due to a confidentiality agreement

4.1.1 Code Review Process

We overview the code review process followed in the target project in Figure 4.1. First, authors send a piece of code to be reviewed. Anyone can, at any point in time, invite reviewers or add itself as a reviewer, what would allow any (interested) developer to contribute. Moreover, in our target project, Gerrit is configured with the *reviewers-by-blame* plugin (GOOGLE, 2017b), which automatically adds reviewers based on the last changes made on the files to be reviewed, as proposed by Balachandran (2013). Immediately after the code is sent, it is analyzed by automated reviewers that check several quality criteria, such as compilation, cyclomatic complexity, lack of documentation, failed unit tests, among other static analysis and runtime verifications. This automated verification usually takes less than 15 minutes to execute and rejects the change if any critical test fails, so that the author can fix the reported issues. Human reviewers and authors can discuss,

Figure 4.1 – Overview of the code review process.



Source: the authors

ask and provide suggestions for each line of code. Moreover, each reviewer can vote to summarize its feedback using one of the following values.

Veto The reviewer considers that the change cannot be integrated without fixing the reported issues or answering questions made. This will *block* the change, thus it cannot be merged.

Rejection The reviewer *recommends* fixes before the change is merged.

Neutral The reviewer typically *asks* questions easy to be answered.

Acceptance The reviewer considers that the change is adequate and can be merged, but considers that reviews from other reviewers are needed.

Approval Only maintainers of the module have this kind of vote, as the ultimate responsible for the quality of the module. Maintainers can perform technical reviews, but they must also verify that relevant developers are not missing in the list of invited reviewers and that the overall state of the code review is adequate.

It is important to note that all invited reviewers, but the maintainer, are not obliged to provide feedback. Before approving the change, the maintainer of each module should consider if the most important reviewers already reviewed the code. In the end, the piece of reviewed code is considered *submittable* if all the following conditions are satisfied: (i) there is no rejection from automated reviewers; (ii) there is no veto; and (iii) the maintainer has approved the change. If all these conditions hold, the maintainer is able to merge the change into the destination branch.

4.1.2 Analyzed Data

The project target of this study involves the development of an operating system for embedded systems of routers and switches, using the C, C++ and Yang ((IETF), 2010) languages. This project has a total of 269 repositories, from which 63 are dedicated to test automation, using Python, Vagrant and Ansible. We consider that the operating system code and its tests are part of the same project, as the developers implement both firmware and tests for each task. All repositories are configured to reject the integration (merge) of code without being reviewed.

The mined data referes to a period of 72 weeks, starting in October, 2014. In the collected data, we had a total of 11,109 code reviews. After filtering these data as described above, we obtained 8,329 code reviews associated with 39,237 comments. Such code reviews are associated with: (i) 201 experienced developers; (ii) 4 development sites in 4 different cities; (ii) and 21 different teams. All teams are organized as *feature teams* and use Scrum with three-week sprints to release new software versions every three months.

4.2 Study Settings

After discussing our target project, we now proceed to detailing our study. We first state our goal and research questions, then describe collected metrics and finally our study procedure.

4.2.1 Goal and Research Questions

To design our study, we followed the goal-question-metric (GQM) paradigm (BASILI; SELBY; HUTCHENS, 1986). Therefore, we first specify our goal using the GQM template and derived research questions. Our goal is detailed next.

To understand the factors that influence code review in the context of distributed software development, characterize and evaluate the relationship between different influence factors and code review effectiveness from the perspective of the researcher as code review is performed by software developers in a single project study.

Based on this goal, we derived a set research questions, each associated with one of the influence factors investigated in our study. There are both both technical and non-technical factors. As said, although some have been investigated in the past, it is our goal to analyze them in the context of distributed software development. For short, we refer to our investigated scenario as *distributed code review*. Our research questions are listed as follows.

RQ1: Does the number *lines of code to be reviewed* influence the effectiveness of distributed code review?

RQ2: Does the number of involved *teams* influence the effectiveness of distributed code review?

RQ3: Does the number of involved *development sites* influence the effectiveness of distributed code review?

RQ4: Does the number of *active reviewers* influence the effectiveness of distributed code review?

4.2.2 Influence Factors and Outcomes

Each research question is associated with an influence factor to be investigated, with respect to their impact on the effectiveness of distributed code review. However, there is no unique metric to measure review effectiveness. Therefore, we consider a set of outcomes of code review, which are measured. They can be used as indicators of the review effectiveness. Before detailing these outcomes, we next further specify our influence factors, which are listed following the order of our research questions.

Patch Size (LOC) The patch size (LOC) is used to refer to the number of lines of code added or modified in a commit and thus need to be reviewed. This lines of code considered are those present in the final version of code, after going through the reviewing process.

Teams Teams refer to the number of distinct teams associated with with the author and invited reviewers. If the author and all reviewers belong to the same team, the value associated with this influence factor is 1.

Locations Locations refer to the number of distinct geographically distributed development sites associated with the author and invited reviewers. If the author and all reviewers work in the same development site, the value associated with this influence factor is 1.

Active Reviewers Actives reviewers are those that actually participate in the reviewing process—with comments or votes—from those invited. Although this can be seen as an outcome of the review, given that there is no control of how many of the invited reviewers will actually participate, we aim to explore if the number of active reviewers influence other outcomes, such as duration. Therefore, active reviewers is investigated as an influence factor, consisting of the number of reviewers that contributed to the review.

Now we focus on describing the analyzed code review outcomes. As discussed in the related work section, different outcomes can be investigated. For example, Bosu et al. (BOSU; GREILER; BIRD, 2015) created a model to evaluate whether the comments of a code review are useful based on the text of the given comments. This measurement, however, may not be completely precise. In our work, we focus on measurements that are more objective and easily obtainable from code review tools. We describe our four selected outcomes below. We show next to the outcome name an acronym that is used to refer to it in later sections.

Duration (DUR) Duration counts how many days the code review process lasted, from the day that the source code is available to be reviewed to the day that it received the last approval of a reviewer.

Participation (PART) Participation consists of the fraction of invited reviewers that are active, ranging from 0% (no invited reviewer participates) to 100% (all invited reviewers participate). Automated reviewers are not taken into account.

Comment Density ($\mathbf{CD}_G$) Instead of simply counting the number of review comments, we take into account the amount of code to be reviewed. Therefore, comment density refers to the number of review comments, which can be any form of interaction such as approval, rejection, question, idea or other sort of comments made by any reviewer, divided by the number of groups of 100 LOC under review. Comment

density thus gives the number of review comments for each 100 LOC. The multiplying factor of 100 is used to avoid very small fractioned numbers, which are harder to compare and less intuitive. Comments from automated reviewers are also not taken into account, as this type of feedback is a constant, regardless of human interactions.

Comment Density by Reviewer ($\mathbf{CD}_R$) We not only analyze the overall comment density, but also the comment density (as above) by reviewer. Therefore, this outcome is obtained by dividing the comment density by the number of active reviews (without taking into account automated reviewers). We aim to verify whether any of our analyzed influence factors makes reviewers to be more engaged in the review.

4.2.3 Procedure

In short, our study procedure consists of extracting information from a database that stores information about the code review process and analyze the relationship between influence factors and outcomes. In this section, we detail how we operationalized this.

Our data is extracted from Gerrit[1], a tool that provides the management of Git repositories with fine-grained control over the permissions for users and groups. It also provides a native mechanism to implement code review, with its associated approvals, allowing votes, comments and edition of the source code. Every interaction among authors and reviewers is recorded, including the comments and votes of the *review bots*, which are automated reviews. It also provides a sophisticated query mechanism to get information about all open and closed reviews. We next describe the steps taken to obtain, process and filter the data for this study using Gerrit.

4.2.3.1 Getting raw data from the code review database

First, we fixed a time frame in the past so we can get data completed code reviews (details of our target project and collected data are given in the next section). Gerrit provides a query mechanism (GOOGLE, 2017a) that can be used to get structured[2] infor-

[1] <https://www.gerritcodereview.com/>

[2] For details about how Gerrit data is structured, please refer to <https://gerrit-review.googlesource.com/Documentation/cmd-query.html#_schema>

mation about code reviews in JSON format. One query for each week had to be made due to the limitation of obtaining at most 500 results per query.

4.2.3.2 Parsing and filtering code review information

The retrieved files contain raw data as obtained from Gerrit, in JSON format. Some of our required information was directly available in these files, such as the name of the commit author. However, in most of the cases, we needed to parse the files to extract information that is not explicitly provided, such as the name of the reviewers and the patch size. This required us to iterate over the raw data that is structured according to the internal database model of Gerrit. After this initial processing, we filtered the commits by removing code reviews from specific types of modules that would distort the results due to their nature. We discarded the following categories of code reviews.

1. *Code reviews from documentation repositories.* Some repositories are used exclusively for internal documentation of the project (processes and products) and they have a different workflow and time constraints as well as are not tied to any software release.

2. *Code reviews from external repositories.* Some repositories are entirely maintained by open source communities, e.g. the Linux kernel. Gerrit has local internal copies of these repositories due to traceability and performance issues—in this way, the repository does not need to be downloaded multiple times. This was not taken into account as reviewers do not review projects in these external repositories.

3. *Code reviews from third-party repositories.* Some repositories only contain code from external providers of components and modules. Usually, the providers release new versions of these components and modules periodically, and local copies are saved in Gerrit. Similarly to above, these are also not reviewed and not taken into account.

4. *Code review of repositories with particular artifacts.* Some repositories store particular types of artifacts, usually binary files, such as images and libraries. These files are not reviewed and, if considered, would increase the patch size. Therefore, they were not taken into account.

4.2.3.3 Representing and analysing data

Given that we have four research questions with four associated influence factors as well as four outcomes, there is a large amount of data to be analysed. Our data consists essentially of continuous or discrete positive numbers, in vary different scales and ranges. For example, there are only four involved locations while the patch size can be up to approximately 4K LOC. To deal with these discrepancies, we adopted an approach similar to that of Baysal et al. (BAYSAL et al., 2016). We clustered data in groups, representing the variance of outcomes in each group using box plots. Additionally, we performed statistical tests to identify groups that are significantly different from each other.

4.3 Results and Analysis

Having described our study procedure and our target project from which we collected the data needed for you study, we proceed to the presentation of obtained results. They are presented according to our research questions, and in each of them we discuss results associated with each of our investigated outcomes. As we are ultimately interested in the effectiveness of code review, we next describe what we consider an effective code review based on the outcomes considered in the study.

Too short or too long code review There are studies (KEMERER; PAULK, 2009; FERREIRA et al., 2010) that suggest time constraints for code review activities, limitation on the number of lines reviewed per hour and also the total amount of hours spent doing code review in a single day. Such limitations are imposed because the code review may become error-prone or even consume more time and resources to be finished due to tiredness. Moreover, if the review takes too long (i.e. high duration) to be completed, developers may be prevented to continue their work and also work does not get done. Therefore, shorter code reviews are preferred. However, if such review is too short, it may also mean that reviewers have not properly analyzed the change.

Low reviewer participation When reviewers are invited to participate in the review, it is expected that they contribute. However, not all participate. Therefore, the higher the participation of reviewers, the better. Nevertheless, we do not expect that participation is 100%, given that there are developers that are invited automatically and may not be relevant reviewers anymore.

Few contributions from reviewers Reviewers may contribute in different ways,

ranging from a simple vote to long discussions. We assume that the higher the number of comments made by reviewers, the more fruitful the discussion and consequently the more effective the review. However, as explained, we do not consider the absolute number of comments, but its density considering the amount of code to be reviewed. Moreover, we consider the amount of contribution generally (CR_G) and by reviewer (CR_R). For both, the higher, the better.

4.3.1 RQ1: Patch Size (LOC)

The first influence factor we analyze is the patch size in terms of LOC. We slit our data into 13 groups according to this factor, listed in the first column of Table 4.1. This table shows our obtained results regarding this influence factor—mean (M) and standard deviation (SD)—for each group, considering each review outcome. We also show the number of code reviews in each group as well as values associated with all reviews, in order to be able to compare overall values with values of each group. For better visualizing the results, they are also presented in Figure 4.2.

We analyze the influence of patch size in review outcomes by comparing the means across the different groups. Given that our data does not have a normal distribution, we use the Kruskal-Wallis test (non-parametric) to verify whether there is statistically significant difference among the means. This is also the case for the other analyzed influence factors and outcomes. When there are significant differences, we use Dunn's test for post hoc tests, because the compared groups have different sizes.

Our results show the there is a significant difference across the different groups ($H=1761.32,\ p < 0.05$). More specifically, larger patches take longer to have their review completed—which is expected. Kemerer and Paulk (2009) and Ferreira et al. (2010) pointed out that there must be a limit of LOC reviewed per hour by a single reviewer and a maximum number of hours of code review per day, in order to achieve good coverage and final quality. However, as can be seen in Figure 4.2, the relationship between patch size and duration is not linear. For example, the average time to review patches of 601–800 LOC is two times greater than the time to review patches of 61–80 LOC, which are ten times smaller. Among some groups, e.g. 21–40 LOC and 41-60 LOC, there is no statistically significant difference. Due to space restrictions, we do not report all significant differences among groups. They can be seen elsewhere.[3]

[3] Available at <http://inf.ufrgs.br/prosoft/distMCR/sbes2017>.

Table 4.1 – Outcomes by patch size (LOC).

LOC	#Rev	DUR		PART		CD_G		CD_R	
		M	SD	M	SD	M	SD	M	SD
0–20	3836	1.8	6.6	86.1	22.3	99.3	139.6	47.3	60.6
21–40	716	3.9	13.0	81.7	22.7	15.3	15.4	5.7	4.1
41–60	475	4.4	10.2	78.4	24.0	9.4	8.4	3.5	2.2
61–80	342	5.0	10.3	79.5	22.4	7.8	6.8	2.7	1.6
81–100	273	5.8	16.0	74.9	24.0	5.4	4.3	2.0	1.2
101–200	762	6.0	12.9	75.2	23.8	4.3	3.7	1.5	0.9
201–400	688	8.0	16.8	73.7	23.7	2.6	2.5	0.9	0.7
401–600	371	9.4	16.0	71.9	22.3	1.5	1.5	0.5	0.3
601–800	203	10.4	16.7	71.2	23.1	1.3	1.5	0.4	0.3
801–1000	158	10.0	15.7	69.4	24.6	1.0	0.9	0.3	0.2
1001–2000	282	13.4	19.1	66.9	23.2	0.8	0.8	0.2	0.2
2001–3000	78	15.1	18.9	62.0	24.7	0.4	0.3	0.1	0.1
>3000	145	12.7	17.0	67.1	24.5	0.2	0.2	0.1	0.1
Total	8329	4.7	12.1	80.1	23.8	48.8	105.8	22.9	46.9

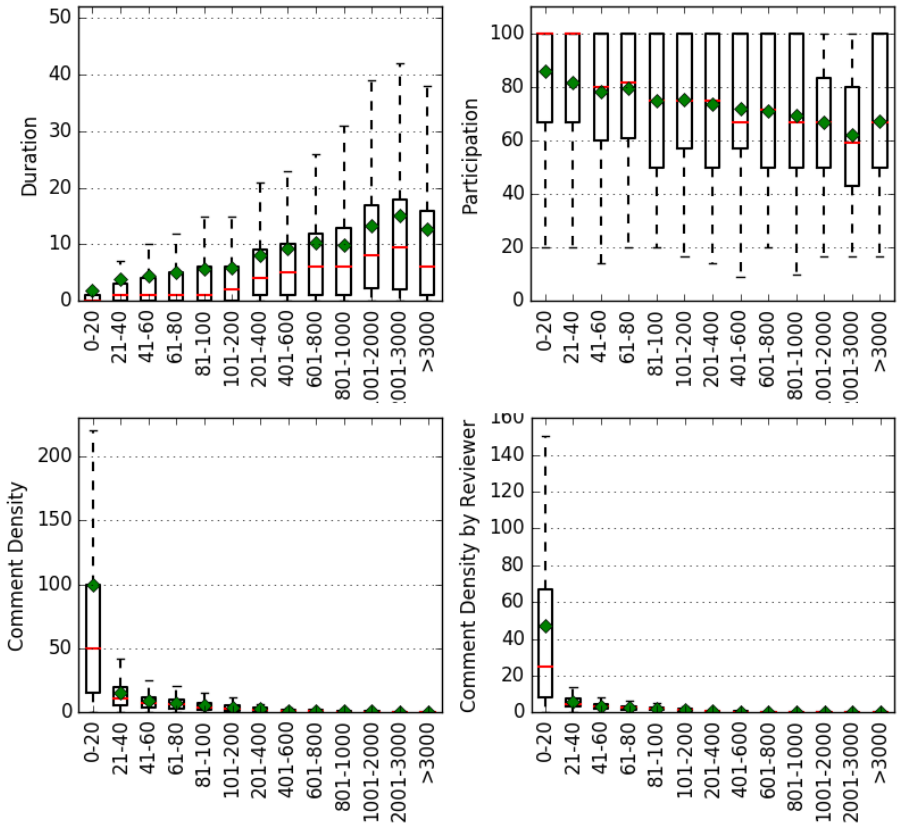


Figure 4.2 – Outcomes by Patch Size (LOC).

Considering participation, we observed that the proportion of invited reviewers that actually provide feedback during the review process decreases when the patch size increases. The difference among the patch size groups is also statistically significant

different ($H=709.58$, $p < 0.05$), mainly due to differences between groups with 600 LOC or less and larger groups. A possible explanation to this is that larger patches likely require more effort from reviewers, discouraging engagement in the process.

When reviewers participate in the code review, the amount of their contributions is measured by the overall comment density and comment density per review. Our data shows that in both cases the larger the patch, the lower the comment density. Regarding overall comment density, there are statistically significant differences ($H=709.58$, $p < 0.05$). According to the post-hoc tests, this is due only to smaller groups. There is significant difference only between few groups with more than 601 LOC, but among groups with less LOC, there are significant differences in most cases. Similarly, the comment density by reviewer decreases as patches are larger ($H=3579.57$, $p < 0.05$), showing similar results in post hoc tests. This indicates that the amount of contribution is highly affected as the patch size increases up to a certain point. Then, the amount of contribution is limited but does not decrease after the patch reaches a certain size ($>$ 601 LOC in our study).

One possible explanation for results regarding patch size is that the patch size have an intimidating effect on invited reviewers, because the time required to provide significant contributions increases. This invested time, in our target project, is not explicitly recorded and is not associated with deliverables considered more relevant, such as produced code.

Conclusions of RQ1: The patch size negatively affects all outcomes of code review that we consider as an indication of effectiveness. Reviewers are less engaged and provide less feedback. Moreover, duration is not linearly proportional to the patch size, which may affect the quality of code review.

As discussed in the related work section, other studies investigated the impact of the patch size in code review. Bosu, Greiler and Bird (2015) showed that for some projects the proportion of relevant comments can decrease by 10% comparing changes in 40 files with changes in a single file, while Baysal et al. (2016) showed that changes with more LOC need more iterations to be concluded, but without considering the time interval. Each iteration is typically the result of an accepted feedback or comment. This indicates that results with respect to patch size in non-distributed scenarios also hold for our investigated scenario.

Table 4.2 – Outcomes by number of teams.

Teams	#Rev	DUR		PART		CD_G		CD_R	
		M	SD	M	SD	M	SD	M	SD
1	4934	3.0	8.3	82.7	24.1	42.2	92.0	22.1	45.1
2	2440	5.8	13.5	76.8	24.0	55.1	112.1	25.5	51.1
3	766	9.4	19.3	75.5	20.1	66.4	140.5	22.1	46.3
4	152	12.4	19.0	73.5	17.7	77.3	177.8	17.8	39.7
5	33	24.1	39.1	73.5	15.8	50.5	79.8	8.9	14.0
6	2	21.5	20.5	63.0	8.4	8.1	7.1	1.6	1.4
7	2	19.0	16.0	58.5	21.5	5.2	0.9	0.6	0.0
Total	8329	4.7	12.1	80.1	23.8	48.8	105.8	22.9	46.9

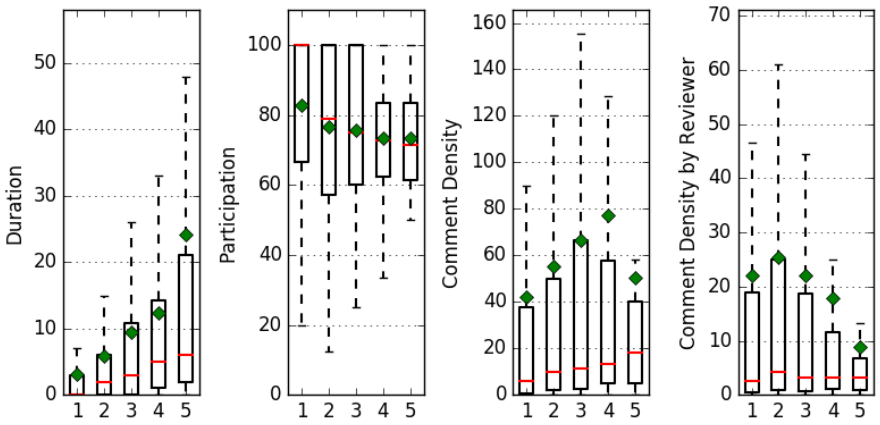


Figure 4.3 – Outcomes by number of teams.

4.3.2 RQ2: Teams

Each code review has a list of involved people, authors and reviewers, and each one works in a single team. Consequently, every code review has also a list of involved teams based on the list of involved people. The results regarding the number of involved teams *vs.* review outcomes are shown in Table 4.2 and Figure 4.3. With respect to the groups with 6 and 7 involved teams, we have only two occurrences of code reviews associated with each of them. We thus omitted them from Figure 4.3, for legibility.

According to our results, the duration of code review is considerably higher if more teams are involved, with higher mean and also standard deviation values. The latter means more dispersion, as can be seen in the corresponding box plots, having more durations that are outliers. This can be partially explained by the working dynamics of teams, which have different goals and tasks, managed by different managers.

Furthermore, technical divergencies are often extensively discussed. When only

one team is involved, such issues are faster addressed, usually mediated or decided by a senior team member or even by the team manager. However, when more teams are involved, the implicit hierarchy among reviewers becomes flatter and reaching a consensus becomes more difficult. In this case, divergences usually reach managers and are resolved after meetings, conferences or e-mail discussions, what slows down the review. In our results, there is a statistically significant difference across the different groups of numbers of involved teams ($H=586.72$, $p < 0.05$). Comparing groups in a post hoc analysis, we observed that this is due to the differences among groups with five or less involved teams, except the difference between code reviews involving 4 and 5 teams.

Similarly, there are also statistically relevant differences with respect to participation ($H=226.72$, $p < 0.05$) and the post hoc analysis showed that the difference is only significant among reviews with 4 or less teams. However, the results indicate only a small negative influence on this review outcome.

Considering the effect on contributions, difference are also significant ($H=184.71$, $p < 0.05$). Post hoc tests showed that this is due to the difference between reviews involving one team and the others. Although Figure 4.3 indicates that the overall comment density increases together with the number of involved teams (except in the case of 5 involved teams), we can see in Table 4.2 that the standard deviation is high, indicating that results vary a lot, justifying the not significant differences. This can be explained by the specific teams involved, whether they are in the same location or not (issue that is investigated in RQ3). In our target project, there is an internal team rotation over the years, as new teams are created, merged or split, with knowledge sharing when teams change, reducing the diversity of skills between author and reviewers and affecting the number of questions, doubts or different opinions. Surprisingly, the comment density by reviewer is higher when two teams are involved, followed by reviews involving one or three teams. The differences among teams are indeed significant ($H=91.94$, $p < 0.05$), with post hoc tests showing that if more than three teams are involved, it actually makes no difference.

Conclusions of RQ2: We found evidences that code review with more involved teams have lower effectiveness considering duration and participation, but higher effectiveness with respect to the overall comment density. There is small, but positive, influence on comment density by reviewer.

Table 4.3 – Outcomes by number of locations.

Locations	#Rev	DUR		PART		CD_G		CD_R	
		M	SD	M	SD	M	SD	M	SD
1	7219	4.1	10.8	80.8	24.1	45.5	98.4	22.5	46.8
2	1076	8.0	17.4	75.2	21.6	68.7	140.2	25.1	47.7
3	34	22.2	31.7	75.9	15.3	132.8	206.6	35.4	52.1
Total	8329	4.7	12.1	80.1	23.8	48.8	105.8	22.9	46.9

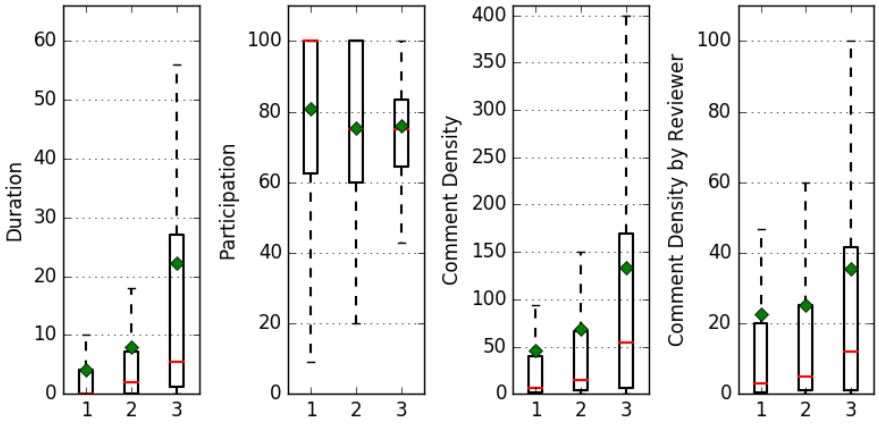


Figure 4.4 – Outcomes by number of locations.

4.3.3 RQ3: Locations

People involved in the code review are not only associated with a single team, but also with a single working site. We now investigate the influence of the number of involved locations on review outcomes. Results associated with this influence factor are show in Table 4.3 and Figure 4.4.

As can be seen, the duration of code review is considerably higher if more locations are involved. With a further analysis of our data, we observed that with two involved locations, reviews that started in the second half of the sprint sometimes were not finished on time, causing a performance penalty to the author's team—as said, code review is mandatory. This can be explained by the natural isolation of people working on different places, which requires daily effort to synchronize priorities and state the importance of every patch under review. Within the same team and location this communication happens on a daily basis in the Scrum daily meetings, or other activities that promotes interaction. There is a statistically significant difference among the different groups ($H=158.0$, $p < 0.05$), in fact, among all groups as shown in the post hoc analysis.

There is also a positive influence on comment density ($H=134.05$, $p < 0.05$) and

comment density by reviewer ($H=56.12$, $p < 0.05$). However, there is a negative impact on participation ($H=86.69$, $p < 0.05$). Post hoc tests show that for these outcomes the differences actually exist only between code reviews with one and two locations, probably because there are few occurrences involving three locations. One possible interpretation of these results, in addition to the geographical distance barrier, is that code reviews with more involved locations have more diversity of technical skills, which is plausible because teams are organized based on groups of related features and technologies. Moreover, there are few rotations of team members among different locations, creating some form of local technical specialization on each location. This diversity promotes feedback, questions and comments, at the cost of requiring more time to complete the review process. Consequently, reviewers from other locations should be invited if there is a good technical reason to do so, otherwise the higher duration is not compensated by a higher level of contributions.

We also observed that the results with respect to comment density by reviewer have large differences when compared to those discussed in the previous sections. Results show that: (i) the average review duration in the same location is 32% greater than the average duration in the same team; (ii) the average duration with two locations is 38% greater than with two teams; and (iii) the average density of review comments with two locations is 24% higher than with two teams.

Conclusions of RQ3: We found evidences that code reviews with more involved locations have lower effectiveness with respect to duration and participation, but higher effectiveness considering contributions. The overall comment density and comment density by reviewer are considerably higher with more involved locations. The participation is slightly lower with multiple involved locations.

4.3.4 RQ4: Active Reviewers

As explained in Section 4.1, in the code review process, the only mandatory reviewer is the maintainer of the module. Other reviewers are invited, but their contribution is optional. Moreover, anyone can invite reviewers. Those that actually contribute are the *active reviewers*. The number of active reviewers might influence how other reviewers engage in the discussion, and this is what we investigate in RQ4. Obtained results regard-

ing the relationship between active reviewers and review outcomes are shown in Table 4.4 and Figure 4.5—the group with 10–12 active reviewers are omitted in this figure because they have only one review occurrence each.

Considering the duration of code review, we intuitively expect higher values as more reviewers provide more feedback and comments and need more time to reach a consensus. This is in fact confirmed by a statistical test ($H=1652.94$, $p < 0.05$), with post hoc tests showing that in most cases the code review with less than ten reviewers takes more time to complete than with more active reviewers.

We highlight that there are reviews, more specifically 1313, involving one active reviewer. This occurs when the author is the module's maintainer, and thus can be the only one required as a review. Consequently, the duration is low in these cases, lasting 1.3 days on average. Exceptional cases occur when the code being reviewed is related to hardware platforms and infrastructure modules of future hardware platforms, where typically one or two developers work on for several months.

The participation of reviewers remains almost the same with more active reviewers. Although there is statistically significant difference among groups, ($H=268.49$, $p < 0.05$), the post hoc tests show that this is due to a few groups that have nearly not significant differences. This indicates that the more invitees, the more active reviewers.

Considering the overall comment density, there is a statistically significant difference ($H=660.89$, $p < 0.05$) when reviewers contribute. However, the post hoc tests show that the presence of more than two active reviewers does not improve significantly the comment density. Moreover, the comment density by reviewer is actually lower with three or more active reviewers ($H=275.55$, $p < 0.05$).

This suggests that a number of two active reviewers seems to be the optimal case considering a trade-off between duration and contributions from reviewers.

Conclusions of RQ4: We found evidences that code review with more active reviewers has lower effectiveness considering the duration. The participation is slightly lower with more active reviewers. Moreover, having more than two active reviewers does not improve the overall comment density and negatively affect the commend density by reviewer.

Table 4.4 – Outcomes by number of active reviewers.

Active Reviewers	#Rev	DUR		PART		CD_G		CD_R	
		M	SD	M	SD	M	SD	M	SD
1	2431	1.3	4.5	84.3	27.4	26.6	54.4	26.6	54.4
2	2502	3.2	7.6	78.6	24.8	54.5	100.5	27.2	50.2
3	1940	6.0	11.9	78.7	21.2	57.1	116.3	19.0	38.8
4	840	7.9	12.6	77.6	18.5	68.8	155.8	17.2	39.0
5	372	14.0	26.4	76.5	15.9	61.5	151.1	12.3	30.2
6	149	18.7	33.4	78.9	14.5	59.4	116.9	9.9	19.5
7	60	18.9	29.3	78.7	13.7	44.0	73.6	6.3	10.5
8	21	17.2	19.5	77.9	12.7	31.4	48.7	3.9	6.1
9	11	33.2	30.1	81.8	15.3	192.1	321.8	21.3	35.8
10	1	35.0	0.0	37.0	0.0	6.2	0.0	0.6	0.0
11	1	15.0	0.0	84.6	0.0	262.5	0.0	23.9	0.0
12	1	20.0	0.0	85.7	0.0	5.0	0.0	0.4	0.0
Total	8329	4.7	12.1	80.1	23.8	48.8	105.8	22.9	46.9

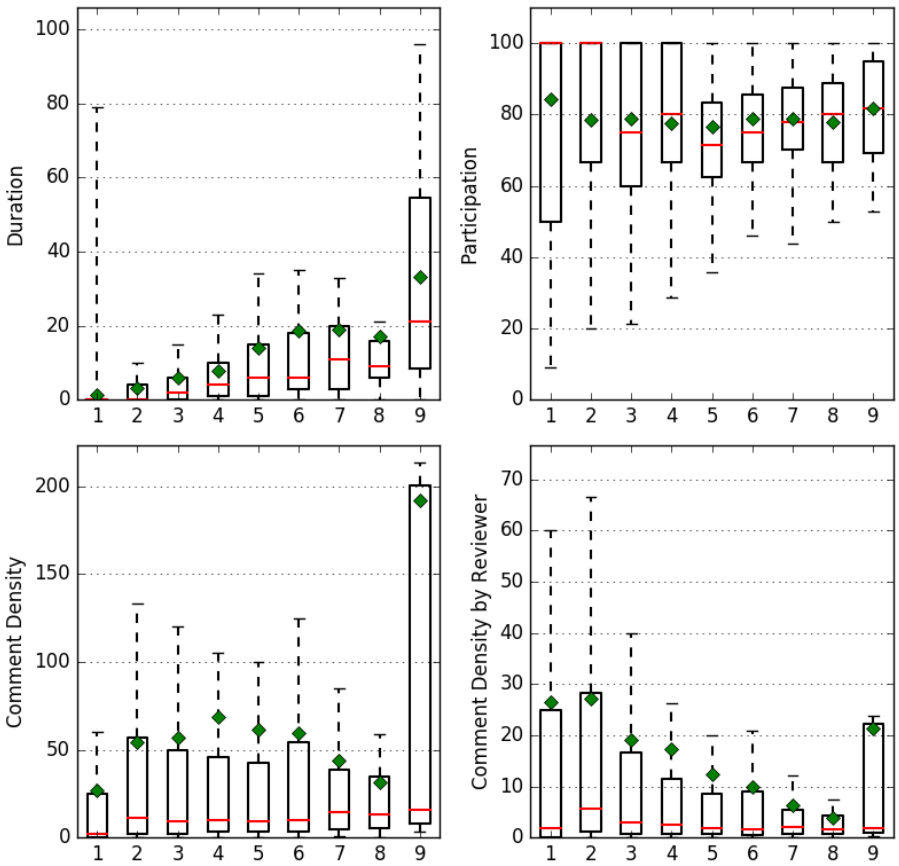


Figure 4.5 – Outcomes by Number of Active Reviewers.

Table 4.5 – Summary of Results.

Influence Factor	**DUR**	**PART**	**CD$_G$**	**CD$_R$**
Patch Size (LOC)	↑	↓	↓	↓
Teams	↑	↓	↕	↓
Locations	↑	↓	↑	↑
Active Reviewers	↑	↕	↕	↓

4.4 Discussion

In this section, we present insights and lessons learned from this study, followed by a discussion of threats to its validity.

4.4.1 Lessons learned

We summarize, in Table 4.5, the conclusions derived regarding how our investigated factors influence review outcomes, considering our investigated scenario.

As can be seen, the amount of LOC to be reviewed affects all outcomes considered in this study. Considering duration, we found that it does not increase at least linearly proportionally to the patch size. This suggests that the rate of 200 LOC/hour proposed by Kemerer and Paulk (2009) is not followed, potentially making code review less effective and more error prone. A lower review coverage is another possible explanation to justify these results, which may lead to more post release defects (SHIMAGAKI et al., 2016; MCINTOSH et al., 2014). This suggests that large patches should be avoided.

Code reviews with many involved teams showed to have negative effects on duration, almost no influence on participation and only slightly better total density of review comments. This shows that inviters should carefully consider who from other teams they invite. Different results were observed regarding locations—discussions were more fruitful with multiple locations involved.

Positive experiences on distributed code review were reported by Meyer (2008), although in the context of code inspection practices (as opposed to MCR). According to him, for a good experience of authors and reviewers, there must be a stable, reliable communication infrastructure with good usability, in addition to the preparation and organization for the code review activity.

Our study gives evidence that code review with more involved (active) reviewers

are less effective, with very significant drawbacks on the average duration and density of review comments per reviewer. This suggests, again, the importance of choosing adequate reviewers and calls for sophisticated code reviewer recommenders.

4.4.2 Threats to Validity

Construct Validity As mentioned in Section 4.1, our target project involves many programming languages, including Yang, which is a way to represent data. When counting and analyzing lines of code, code written in all these languages are treated equally. Reviewing the same amount of code in one language may require more time than in another. However, considering the developers' expertise, they do not state more difficulty in reviewing code in particular languages, and also the involved languages are not largely different with respect to verbosity. Furthermore, many medium and large software projects use many programming languages. Therefore, the amount of code in different languages is considered a random variable rather than a confounding variable of the study.

Internal Validity We identified three internal threats. First, given that we analyzed an extensive period of our target project, its developers changed over time. However, as the number of developers and analyzed reviews is large, individual developers' behavior and expertise have a low impact on the obtained results. Moreover, this change in development teams is expected in any software project.

Second, in most of the cases, authors and reviewers communicate using Gerrit to provide feedback, even when they are on the same team or location. However, there is no explicit obligation in the target project to record in Gerrit feedback given by means of other forms of communication, such as telephone or informal meetings. Nevertheless, this is very unusual for this project—developers tend to use the available tools to ensure that relevant questions will not be forgotten by the authors. Isolated occurrences thus do not significantly affect the results.

Finally, the participation outcome may have been affected due to the automatic addition of reviewers by Gerrit's reviewers-by-blame plugin. The plugin may add, as reviewers, developers that no longer work in the same team or even in the company. Consequently, their participation was not expected. As what matters is the relative comparison of participation for groups of each outcome, this likely has not affected the results. The probability of having reviewers that fall into this category is the same for the different reviews.

External Validity Generalizing the results of empirical studies is always an issue, because the collected and analyzed data may not generally represent software projects. Although we focused on a single project, our results are based on a large amount of data of a large project. Therefore, we were able to identify trends and statistically significant results. However, we emphasize that our results are potentially generalizable only for distributed development environments similar to that of our target project. Although geographically distributed, development locations occur in the same country and mostly involve developers of the same nationality. Therefore, further studies should investigate whether our results hold to *globally* distributed development environments, which may impose additional barriers to MCR, such as different time zones, communication languages, and culture.

4.5 Final Remarks

In this study, we presented the results of an empirical research. We extracted a large amount of code review information from a software project whose aim is to develop an operating system for embedded systems. The analyzed data were obtained from a project with many developers and teams, geographically spread in four different cities. We investigated how the patch size (in terms of lines of code), the number of teams, the number of locations, and the number of active reviewers influence the duration, reviewer participation, and comment density (general and by reviewer) of the review. In order to complement and contrast these results, Chapter 5 presents our second foundational study, which investigates the developers' perspective on this matter by surveying experienced code review practitioners.

5 SURVEY WITH CODE REVIEW PRACTITIONERS

In the previous chapter, we presented a study that mined objective data from software repositories and code review databases. To complement our findings from that study, we now present a study that explores subjective data. We detail a developer-centric survey conducted in a Brazilian medium-sized company, in which we focus on the perception of developers on how MCR works or should work, how developers interact, and their motivations to participate in reviews—given that reviewer partiticipation is encouraged, but not enforced. We next describe our study settings in Section 5.1. Then, we present and discuss obtained results in Section 5.2. Finally, in Section 5.3 we present the final remarks.

5.1 Study Settings

MCR leads to a set of *outcomes*, which are observed consequences of the adoption of the practice. These outcomes can be *external* (e.g. product and code quality), when they are relevant for the project, company, or client, or *internal*, when they consist of measurable properties (e.g. review duration) that tell us more about the MCR internal dynamics. Outcomes can be affected by a wide range of *influencing factors*, associated with the code change to be reviewed (e.g. its size), its author (e.g. expertise), and the reviewers (e.g. code familiarity). In addition to influencing factors that are software-related characteristics, human aspects—such as the developers' motivations to participate in reviews and their social interactions—can also play a key whole in if, how, and when reviews are done.

Based on these concepts, our survey targeted five key aspects associated with MCR, presented in Figure 5.1. First, we focus on outcomes. We investigate how developers assess the impact of MCR on *external outcomes* and what values for *internal outcomes* they believe that would maximize the benefits of MCR. This allows us to evaluate to what extent MCR provides its pros and cons and also the desired levels for internal outcomes for MCR to work, such as expected amount of feedback provided by reviewers. As internal outcomes are affected by influencing factors, we then examine the developers' perception of the *relationship between influencing factors and internal outcomes*—complementing the results observed in studies based on data from MCR repositories. In addition to influencing factors, MCR is affected by the *developers' personal motivations*

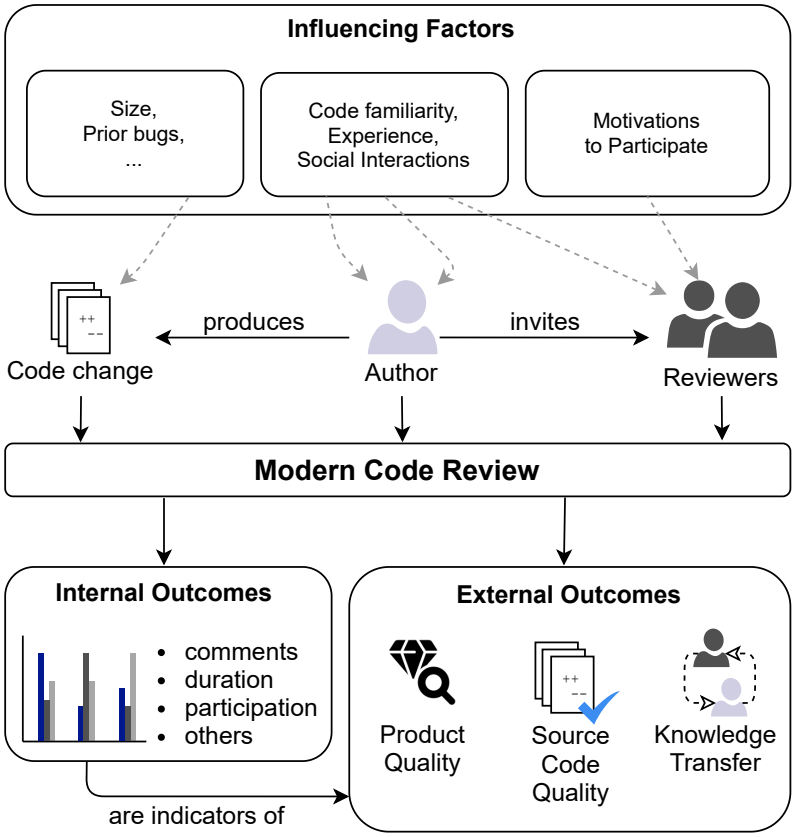


Figure 5.1 – The Dynamics of Modern Code Review.

to participate in reviews. We thus analyze these motivations so that it is possible to provide incentivization mechanisms. Finally, as MCR requires interaction among developers, we inspect which *means of communication* they adopt besides MCR tools. This allows us to identify whether important discussions about projects may remain undocumented due to how developers interact.

We surveyed developers, who work in medium-scale proprietary projects of embedded systems. MCR is used in these projects as a verification technique. We collected the demographic characteristics of the participants as well as a self-report of their level of experience in topics related to the study (see Table 5.1).

From the 79 respondents, we excluded those that report (very) low experience in MCR, resulting in 73 subjects. Almost all subjects are male, which is in accordance with the gender distribution in the target company. To further explain the answers obtained in our initial survey, 29 of the subjects volunteered to answer an additional follow-up questionnaire. The questionnaires are provided in Appendix A, and both questionnaires and collected data are available online[1].

[1]<https://inf.ufrgs.br/prosoft/resources/2020/ieee-sw-mcr-survey>

Table 5.1 – Demographic data of survey participants (N = 73).

Age		20–29 years 20 (27%)	30–39 years 43 (59%)	> 39 years 10 (14%)	
Gender		Male 70 (96%)	Female 2 (3%)	Other 1 (1%)	
Education		B.Sc. 52 (71%)	M.Sc. 19 (26%)	Ph.D. 2 (3%)	
Years of Experience in Software Development		2–5 years 7 (10%)	5–10 years 30 (41%)	> 10 years 36 (49%)	
Experience in	**Software Development**	Low 0 (0%)	Average 9 (12%)	High 44 (60%)	Very High 20 (27%)
	MCR as Reviewer	Low 0 (0%)	Average 24 (33%)	High 40 (55%)	Very High 9 (12%)
	MCR as Authors	Low 0 (0%)	Average 21 (29%)	High 42 (58%)	Very High 10 (14%)
	Software Development involving Multiple Teams	Low 3 (4%)	Average 16 (22%)	High 34 (47%)	Very High 20 (27%)
	Software Development involving Multiple Sites	Low 6 (8%)	Average 18 (25%)	High 34 (47%)	Very High 15 (21%)

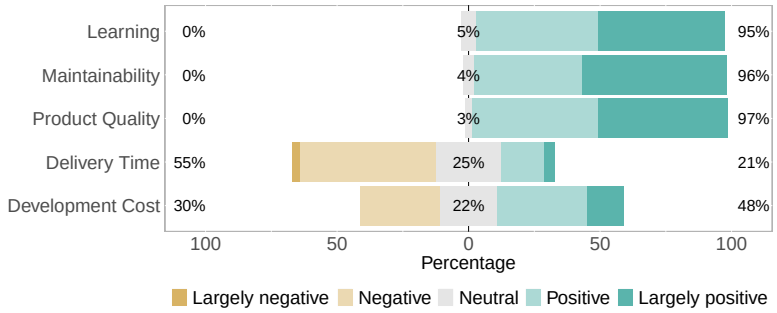
5.2 Results

In this section, we present and discuss the results of this study. These results are categorized according to three aspects: (i) external and internal outcomes, (ii) influencing factors and outcomes, and (iii) motivations and reviewer interaction.

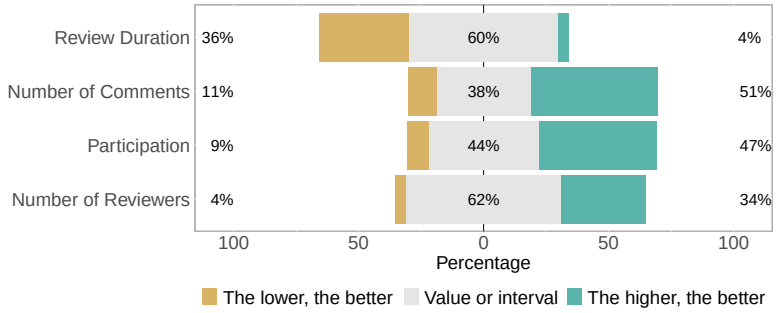
5.2.1 External and Internal Outcomes

Previous studies conducted in companies such as Microsoft (BACCHELLI; BIRD, 2013; MacLeod et al., 2018) surveyed benefits of code review. In our survey, we—in addition—asked developers to *what extent* MCR contributes, positively or negatively, to the main previously identified benefits—*maintainability* and *product quality*. We also added *learning*, as it has been pointed out as a potential benefit in studies done in other environments (BAUM et al., 2016).

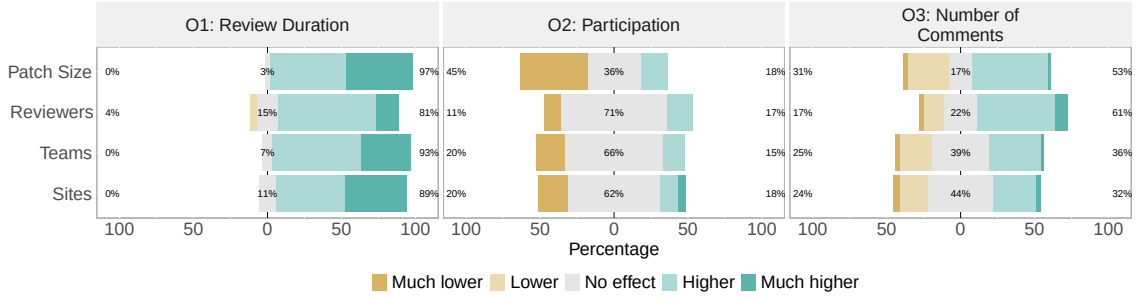
The provided answers are shown in Figure 5.2a. As expected, almost all developers perceive that MCR promotes these benefits. The most positive contribution is associated with maintainability, but its difference to the other two benefits is not significant. Respondents had the opportunity to list other external outcomes. With respect to main-



(a) Level of contribution of MCR to external outcomes



(b) Expectations for internal outcomes



(c) Effects on internal outcomes when influencing factors are higher

Figure 5.2 – Evaluation of MCR outcomes and associated influencing factors.

tainability, three emphasized that MCR helps follow standards in projects. Eight listed *knowledge dissemination* as an outcome, not only within the team, but also with managers and testers. This benefit was also identified in other surveys. Additionally, nine developers pointed out that MCR has a positive impact on *team integration*.

In addition to looking at these external outcomes seen as benefits, we also assessed the contribution of MCR to two key external outcomes relevant in software projects, namely *delivery time* and *development costs*. Generally, we assume that if bugs are caught in earlier stages of the development and the code is more maintainable, both time and costs are reduced in the long-term. However, surprisingly, 55% and 30% of the participants perceive that MCR has a negative impact on these two outcomes, respectively. To

clarify why the impact is stronger for delivery time, we further asked our subjects which scope they considered when assessing these outcomes: a particular *task*, a project *release* or the whole *project*. The results explain the variation between time and costs. Considering *costs*, most of the participants (64%) considered the whole project, while only a few others (29%) considered the task. And, for *time*, 58% considered the task, and the remaining considered in equal proportion the release and project scopes. This shows that, when developers think about time, most tend to consider the short-term (task); while when they think of costs, most tend to take into account the long-term (the costs of the whole project). These questions on external outcomes show that MCR provides benefits and can even reduce costs in the long-term, but there is a penalty in the short-term, which is the delay in the task execution. Consequently, MCR benefits for a project should be made explicit to make involved stakeholders willing to pay off.

Despite the importance of external outcomes, they are hard to measure and control from a management perspective. Therefore, we also looked at internal outcomes of code review. These can be measured with lightweight metrics that can be used as indicators for controlling, understanding, and directing a software project, as suggested by Rigby et al. (RIGBY et al., 2012). For example, by controlling the review duration, it is possible to evaluate if reviews are delaying deliveries; or, if many reviewers decline participating in reviews, management should investigate why this is happening and how to incentivize participation.

We asked our subjects what values they consider adequate for the following internal outcomes: *review duration*, *number of comments*, number of developers that accept an invitation to review a code change (*participation*), and *number of reviewers* participating in a review. Possible answers are the lower, the better; the higher, the better; or a specific number or interval is the most adequate—in this case, subjects were requested to specify values and add complementary information. Figure 5.2b shows the developers' opinions, from which we derive the following conclusions.

- *Reviews should take at most a couple of days to be done.* Considering our context (proprietary projects and medium-sized companies), reviews should be done *quickly* (from 1 hour to 2 days). Longer periods must be justified by, e.g., complex code to be reviewed. Feedback given in shorter periods may indicate that the review has been done superficially (possibly due to overly large code changes). Therefore, reviews should be thought as activities done in the developers' *daily* routines.

- *Participation in reviews should be high, with the provision of many comments but*

not too many. With respect to participation, at least 50% of the invited reviewers are expected to participate in the review discussion. Because there is an expectation of participation, mechanisms to show to authors the reviewers' availability might be useful. The expected range for number of comments is 5–15, but the developers emphasized that the comments should be constructive and not involve *personal opinion*.

- *Having more reviewers is generally considered good, but there is a limit.* There is a consensus that having 2–3 reviewers is the most adequate number. However, more than this number of reviewers may lead to lengthy discussions (and not valuable comments, as said above).

5.2.2 Influencing Factors and Outcomes

Now, we look at what factors influence MCR internal outcomes. This has been done in previous studies (SANTOS; NUNES, 2017; BAYSAL et al., 2016) based only on the analysis of objective data through *repository mining*, and in a *preliminary* survey with developers (SANTOS; NUNES, 2018).

We asked our participants about the perceived effects on internal outcomes when a given influencing factor increases. For example, what happens to the review duration when the number of reviewers gets higher: does it get (much) lower, (much) higher, or is there no effect? The four considered influencing factors, previously investigated in a study with objective data (SANTOS; NUNES, 2017), are: *patch size* (the number of lines of code to be reviewed), the number of *reviewers*, the number of involved *teams*, and the number of involved development *sites*. The last two factors target the the relationship between the author and reviewers, considering projects where there are system modules for which distinct teams, possibly in different geographical locations, are responsible.

Figure 5.2c shows the obtained results. Over 80% of the respondents believe that the duration is prolonged by increasing any of the influencing factors, mainly the patch size (97%). Consequently, large patches should be avoided. The effect of the number of reviewers is less intense than the other factors. As can be seen, involving external reviewers (developers outside the author's team) is believed to prolong the review. Therefore, there should be guidelines to select reviewers, mainly related to the number of invited reviewers, to prevent undesired long durations and project delays. Moreover, as pointed

out in the previous section, developers indicated a preference for 2–3 reviewers and not more.

The only influencing factor that seems to negatively affect participation is the patch size; the remaining factors are believed to have no effect on this outcome. Thus, again, large patches have a negative effect. This is consistent with results of studies using objective data (RIGBY et al., 2012; SANTOS; NUNES, 2017; BAYSAL et al., 2016). Interestingly, two subjects stated that more involved sites lead to much higher participation. A possible explanation lies in the developers' motivations to participate in reviews, as we will discuss later.

Lastly, looking at the number of comments, half of the subjects stated that larger patches lead to more comments. Note that the patch size leads to a higher (and not much higher) number of comments. Therefore, this may not be a positive influence because the comment density (comments per line of code) may be the same. A similar finding holds for the number of reviewers. The number of comments may increase, but the number of comments per reviewer may be the same, as suggested by one of the participants and also indicated by objective data (SANTOS; NUNES, 2017). Another participant pointed out that knowledge on particular technologies plays a role in the number of comments. This influencing factor has in fact been explored in an existing automated reviewer recommender (RAHMAN et al., 2016b). Finally, there are inconclusive results associated with teams and sites. This is in contrast with a previous study that mined objective data, which has shown that higher values for these factors lead to more comments (SANTOS; NUNES, 2017). This suggests that developers may not be aware of the factors (e.g. involving external reviewers) that lead to more comments. It is important, therefore, to provide MCR tools with features that inform developers that particular choices for reviewers might have certain consequences. For example, involving reviewers of other teams and sites can lead to more feedback; or, if a high number of reviewers is invited, this may delay the review.

By also asking participants whether they agree with a set of statements on this topic, we investigated the impact of *large patches* and *social interactions*. Almost all of them agree that reviewers tend to avoid and make superficial reviews of large patches (90% and 88%, respectively). With respect to team membership, 71% of the participants consider that face-to-face meetings within a team are helpful to prevent delaying reviews. In addition, almost 90% agree that there is a need for external intervention, such as from management, to handle stuck reviews for both other teams and sites. Lastly, with respect

to personal relationships, 85% agree that reviewers may make less rigorous reviews to avoid conflicts with their peers and have mixed opinions about the relationship between the rigorousness of the review and the fact of not personally knowing the author of a patch to be reviewed. This indicates that social connection is an important influencing factor in MCR, which seems unexplored and can be addressed in the future. Moreover, there must be means of tracking reviews requested by external teams.

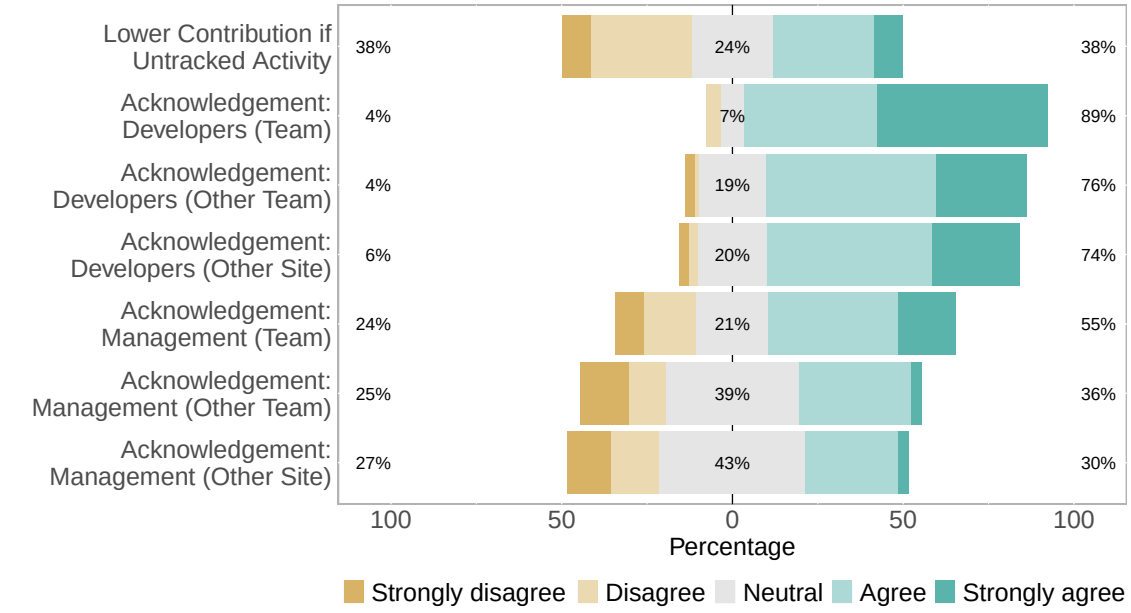
5.2.3 Motivations and Reviewer Interaction

Most studies that investigate motivations for code review consider the goals that can be achieved by the adoption of the practice (BACCHELLI; BIRD, 2013; MacLeod et al., 2018). We, in contrast, analyze what motivates developers to engage in code reviews, that is, what their *personal motivations* are. We split such motivations in two groups: (i) *extrinsic motivations*, which are those associated with the developers' context; and (ii) *intrinsic motivations*, which are those related to individual beliefs and benefits. Given that code reviews are often a task that is not mandatory to be done, it is important to understand what personally motivates developers in order to create mechanisms of incentivization that match the developers' motivations to contribute reviews. Extrinsic motivations require more straightforward mechanisms, such as feedback from management, while intrinsic motivations need creating a culture.

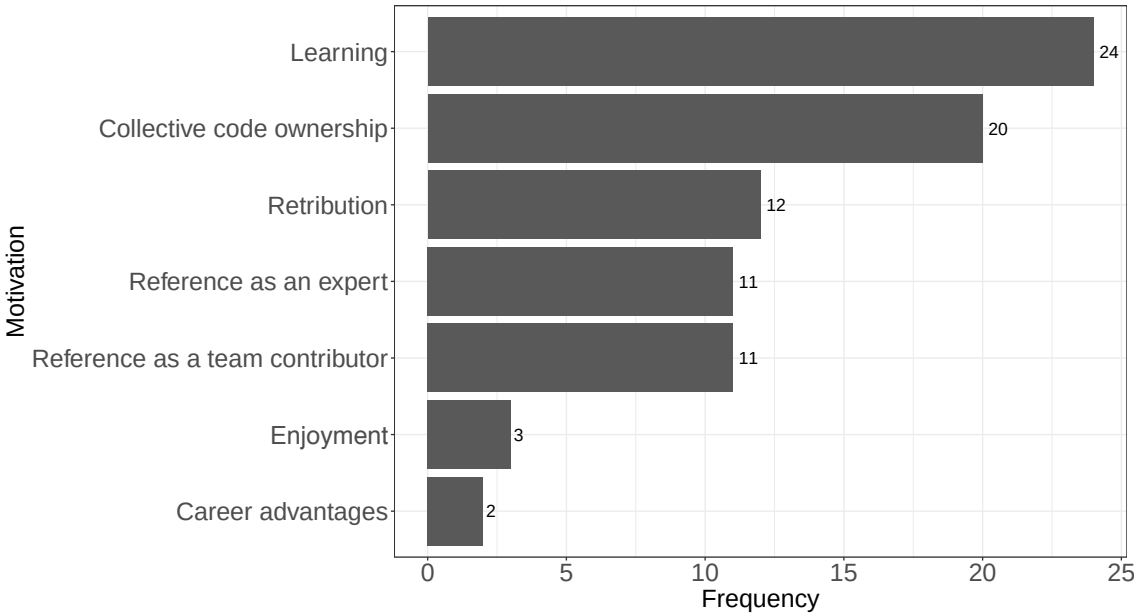
We consider three types of extrinsic motivations. The first refers to the *tracking of code reviews* as a registered performed activity in projects. That is, the time devoted to reviews is logged and transparent, and not diluted in other development activities. The other two types of motivation consist of the *acknowledgment from other developers* and *managers* for performed reviews. We thus asked our participants questions in these directions. The results are shown in Figure 5.3a.

We first inquired whether reviewers tend to contribute less if code review is an untracked activity. There is no convergent answer regarding this. Still, as a number of participants (38%) indicated that untracked review activities can decrease the feedback given by reviewers, the tracking of this type of activity can be used as a means of incentivizing reviewers' contributions.

Concerning acknowledgment for code reviews, we requested participants to inform if there is acknowledgment from other developers and managers from three perspectives: within the team, other teams, and other sites. We observed that developers tend to



(a) Extrinsic motivations



(b) Intrinsic motivations

Figure 5.3 – Motivations for developers to perform code reviews.

appreciate reviews done by others, and this is stronger within teams. As developers from other teams and sites also appreciate performed reviews, this could explain why participation is positively affected by involving other teams and sites, as discussed previously. Different observations are made regarding the feedback from management, as the majority of the subjects does not agree that there is acknowledgment from managers of other teams and sites. Only 36% and 30% agree with this, respectively. Managers from the team seem to have greater appreciation for performed reviews than external managers, but still to a limited extent. This is an indicator that a stronger explicit acknowledgment from superiors can have a positive effect on MCR.

We now look into the intrinsic motivations, which were investigated as part of our follow-up questionnaire. Figure 5.3b presents the number of developers that informed which motivations—from those listed in the y-axis of the chart—cause them to contribute reviews. It shows that a representative amount of them (80%) *review others' code for learning*. The second most frequent motivation is the sense of collective code ownership. This is perhaps a two-way lane: the sense of collective code ownership motivates developers to review code, and code reviews increase the sense of collective ownership. With respect to the remaining intrinsic motivations, nearly half of the participants do code reviews as a retribution for the reviews done by others, and to be seen as a reference, as an expert or a team contributor. Enjoyment or career advantages do not seem to be a motivational component. In summary, the results suggest that in companies where code review is successfully adopted, MCR works due to the developers' personal interest (i.e. learning), and two social motivations, namely sense of collective code ownership, and the acknowledgment that they receive from peers for contributing reviews. This shows that the success of MCR highly depends on individuals and their personal commitment to MCR. Consequently, to motivate developers and increase participation in reviews, it is important: (i) for management, both from the team and other teams/sites, to acknowledge the work done in reviews; and (ii) discuss in team meetings the work done in reviews so developers become aware that peers are also contributing.

Lastly, we analyze how reviewers interact, i.e. the means of communication used to discuss issues raised in reviews. MCR is a tool-supported activity, and discussions should occur within it. However, there is no enforcement that this is always the case. In our initial survey, we asked developers whether communication by means of other tools (such as e-mail or chat) and informal feedback (i.e. undocumented discussions) is frequent during code reviews. More than half of the respondents (59%) (strongly) disagree

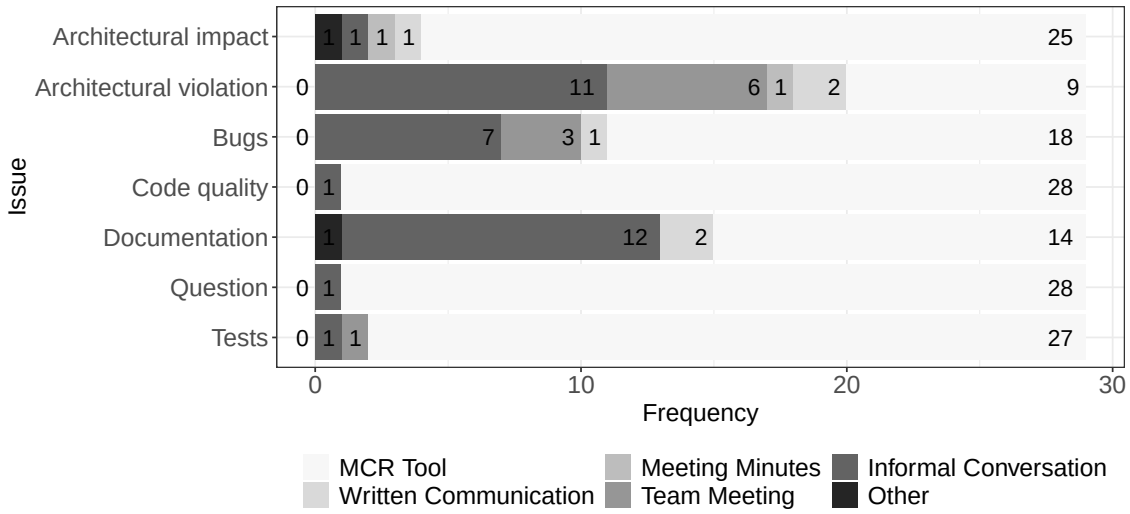


Figure 5.4 – Number of participants that mostly adopt each means of communication according to the code review issue being discussed.

that the use of other tools is frequent, but almost a half (47%) pointed out informal conversations as a frequent form of communication. We then asked developers in our follow-up questions what is the most used form of communication for different types of issues that may appear in code reviews.

The provided answers are shown in Figure 5.4. The employed MCR tool is widely used for localized code issues, which are those related to bugs, code quality, tests or documentation. However, *architectural issues* tend to be discussed not only outside the MCR tool but also using informal forms of communication. 59% and 34% of the developers informed that they discuss issues associated with architectural impact and violations, respectively, in informal conversations or team meetings. Informal conversations are also the most used form of communication to ask questions for half of the developers.

To understand why developers choose to have informal conversations instead of tools, we additionally asked our subjects if the size and the severity of the issue are key factors to choose a certain form of communication. With respect to the former, 49% of the respondents stated that the size influences it. This could justify why some questions are made in informal conversations, as long discussions are easier to be done verbally. Two developers explained that they document the issue in the MCR tool and then informally communicate with the author. Because some discussions are not documented in the MCR tool, it is important to understand the impact of having them undocumented. Moreover, severity plays a key role for 76% of the developers. Six of our subjects mentioned that examples of severe issues are those related to the software architecture. This corroborates the results of our closed-ended question: knowledge and decisions associated with the

software architecture discussed in code reviews remain undocumented. Therefore, these recurrent informal communications regarding the software architecture can contribute to its knowledge vaporization (BOSCH, 2004) and, possibly, to future architecture degeneration.

5.3 Final Remarks

Despite being supported by tools, modern code review is essentially a human-based activity. As suggested by the results of our study, how close developers are from each other may interfere in the rigorousness of reviews. This reveals an issue that is currently unexplored, motivating the need for a deeper understanding of the impact of these relationships on MCR.

Our study also showed that the key drivers for the successful MCR adoption are the developers' own motivations and their sense of collective code ownership. Moreover, the acknowledgment and feedback from management seem limited. This can be addressed by considering MCR as a first-class citizen in projects, treating review activities similarly to those associated with coding and testing. That is, MCR activities should be tracked and the time spent on them should be acknowledged.

Even though MCR focuses on localized code changes, changes may have a larger impact, causing the revision and evolution of the software architecture. Our study indicated that discussions on this are often undocumented, which can lead to a degeneration of the architecture and, thus, higher future maintenance time and costs. Consequently, how to disseminate within the whole team architectural decisions made in code reviews and further understand the interplay between code reviews and architecture evolution remain as open challenges.

In this chapter, we complemented the data-centric study presented in Chapter 4 with a survey that provides a developer-centric perspective. Given the knowledge and insights from these two foundational studies, we present a study that implements and evaluates two code review prediction models in the next chapter.

6 TEAM-RELATED FEATURES IN CODE REVIEW PREDICTION MODELS

Because MCR is essentially a collaborative activity based on developers' contributions, finding suitable reviewers is key for receiving useful feedback (BACCHELLI; BIRD, 2013; RIGBY; BIRD, 2013). However, finding suitable reviewers, mainly in large projects, can be time-consuming and a not-so-straightforward task. In this chapter, we present a study that implemented and evaluated two code review predictors based on the results and insights of our two presented foundational studies. In Section 6.1, we present the features used by the mentioned predictors. The research questions, procedure, and dataset are presented in Section 6.2, and the results are presented in Section 6.3. Obtained results are discussed in Section 6.4, followed by the final remarks in Section 6.5.

6.1 Team-related Features

We now describe the features that we explore in this work. Before doing so, we first introduce the project organizational structure we consider in this study and the adopted terms. Next, we present each of the three sets of features, which are code ownership (CO), workload (WL), and team relationship (TR).

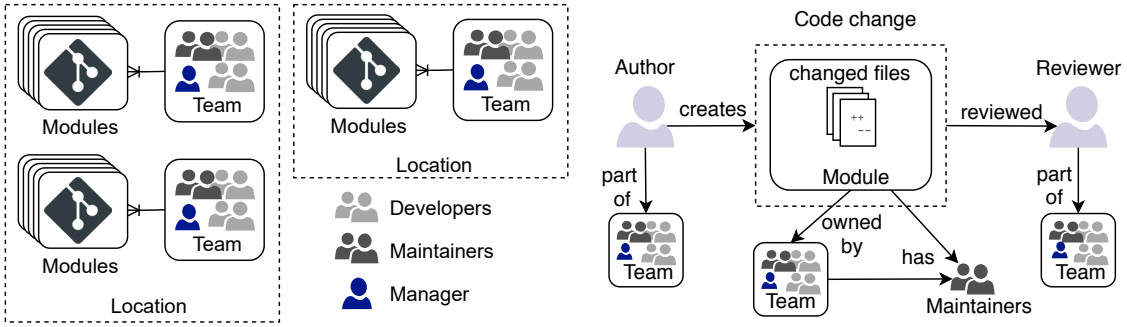
6.1.1 Terminology

Software projects can be organized and managed in different ways. In our work, we consider the scenario where there is a large system being developed or evolved, creating the need for having various *teams*, possibly working on different *locations*. These teams develop different parts of the software, which we refer to as *modules*. Each module has at least one developer that is its *maintainer*, who is responsible for that module. Each team is led by a *manager*. These terms are illustrated in Figure 6.1a and more precisely defined as follows.

Module: a repository with source code that contains the implementation of a well-defined component of a system.

Developer: someone who contributes to the development of software modules.

Maintainer [of a module]: a developer who can approve other developers' work and



(a) Project organization and roles

(b) MCR process

Figure 6.1 – Overview of the adopted terminology.

accept it into the codebase of a module. Typically, maintainers are senior developers expected to enforce quality standards and keep the technical debt under control.

Manager: someone to whom developers report. Usually, a manager is responsible for hiring processes and performance evaluations of developers, for example.

Team: a group of developers that report to the same manager.

Location: a geographic location where a team member works.

Within teams, MCR is adopted as a quality assurance technique, with a process overviewed in Figure 6.1b. An *author*, who is a developer, implements and creates a code change. This code change is associated with a module and involves a set of added, deleted or modified *files*. Each code change is reviewed by a set of *reviewers*. The only mandatory reviewer is the maintainer of the module. If the maintainer does not explicitly accept the code change, it will not be merged into the module's repository.

6.1.2 Feature Sets

Given the organizational structure described above, we now describe the features explored in this study. As discussed before, the idea behind two of these sets (code ownership and collaboration) has been explored in previous work but none has refined them taking teams and project modules into account. Workload, in turn, has been pointed out as an issue but also not yet considered for predictions.

Code Ownership Features.

Previous work considered code familiarity for recommending reviewers, taking into account previous experiences as author and/or reviewer. In order to exploit the project

Table 6.1 – Description of code ownership (CO) features.

Feature	Description
File Reviewer	The number of changes in the *same file* that a developer *reviewed* in the past. If the code change involves multiple files, it is the sum of the values of each file.
File Author	The number of changes in the *same file* that a developer *authored* in the past. If the code change involves multiple files, it is the sum of the values of each file.
Module Reviewer	The number of changes in the *same module* that a developer *reviewed* in the past.
Module Author	The number of changes in the *same module* that a developer *authored* in the past.
Is Maintainer	A Boolean value that is true if a developer is a maintainer of the module on which the change was made.

organization, we also consider the module structure of the project in features related to code ownership. Therefore, we not only look at code familiarity at the file level as features, but also at the module level. For both levels, we consider both previous experience as an author and as a reviewer. Furthermore, given that modules are associated with maintainers, we also add as a feature in this set the information if the developer is the maintainer of the module. These are the features included in the code ownership (CO) set, which is summarized in Table 6.1.

Workload Features.

Most of the existing work to recommend reviewers aimed to identify developers that are *able* (in the sense of having the background knowledge) to review a particular piece of code. However, it is as important to consider whether developers are *available*. In fact, there is recent work that raised the issue that the developers' workload should be taken into account while recommending reviewers (MIRSAEEDI; RIGBY, 2020). Consequently, the current assigned tasks of the developers can be used as features to capture their current workload. This information can be estimated from code review repositories by looking at the currently open code reviews in which a developer is participating, either as an author or as a reviewer. This leads to our two workload (WL) features, which are listed in Table 6.2.

Team Relationship Features.

Although previous studies investigated the social network of developers, the relationship among them that follows from how they are organized in teams has not been explored. To recommend reviewers, this is actually an important issue to be taken into ac-

Table 6.2 – Description of workload (WL) features.

Feature	Description
Author Workload	The number of open code reviews that are authored by a developer.
Reviewer Workload	The number of open code reviews in which the developer is participating as reviewer.

Table 6.3 – Description of team relationship (TR) features.

Feature	Description
Same Team	A Boolean value that is true if the author and reviewer are in the same team.
Same Location	A Boolean value that is true if the author and reviewer are in the same location.
Team Interactions (Rev)	The number of changes in *modules* of the author's team that a developer *reviewed* in the past.
Team Interactions (Aut)	The number of changes in *modules* of the author's team that a developer *authored* in the past.

count because there are studies that give evidence that (i) the amount of feedback provided during code review is affected when reviewers and authors are from different teams of the same company (BOSU; GREILER; BIRD, 2015; SANTOS; NUNES, 2017) or from different organizations when they contribute to the same project (BAYSAL et al., 2016); and (ii) the collaboration among developers is also affected when they are not located in the same geographic location (OLSON; OLSON, 2000; OLSON; OLSON, 2013; SANTOS; NUNES, 2017). The findings of these studies emphasize the relevance of considering teams in reviewer recommenders. Therefore, we explore the relationship of developers within teams using the four features listed in Table 6.3. The first two capture whether the reviewer works in the same team or the same location as the author, while the two last features capture the interaction of the reviewer with the author's team. This interaction can have occurred by means of contribution as an author or reviewer.

Given our introduced sets of features, we next evaluate their effectiveness to identify developers that will participate in reviews and predict how much they will contribute as reviewers. The former has been used as the criterion to evaluate reviewer recommenders and is, therefore, and important prediction to make. The latter is related to the ultimate goal of code review, which is to receive feedback from reviewers.

6.2 Evaluation

We now focus on evaluating the effectiveness of the proposed features to predict reviewers' participation and the amount of feedback that they would provide. In this section, we detail the design of study for such an evaluation. We first detail our research questions, then describe our study procedure and the used dataset for training and testing our prediction models. The replication package, including the data with anonymized information from the reviewers, is available online.[1]

6.2.1 Research Questions

In this study, we aim to answer the following research questions.

RQ1: What is the prediction power of each of the proposed sets of features to predict reviewer participation and amount of feedback?

RQ2: What combination of individual features provides the best performance to make such predictions?

RQ3: What is the impact of different timeframes of past data used to train the prediction models?

With RQ1, we aim to evaluate the performance of the proposed feature sets—code ownership (CO), workload (WL), and team relationship (TR)—to make predictions related to reviewer recommendation. We not only compare the results obtained for each feature set with each other, but also compare them to two baselines. Because individual groups of features are likely not optimal to make our target predictions, in RQ2, we aim to identify the best set of features by means of feature selection. Finally, an important—and unexplored—aspect to consider when building prediction models is the amount of data used to train models. This is a relevant issue because training prediction models regularly using all past data might be computationally expensive due to the large amount of data available for long-lived projects or companies with a high number of repositories. Moreover, this might not lead to the best results. In addition, gathering data about teams and managers for an extended period in the past is not always practically feasible to do.

[1] <https://www.inf.ufrgs.br/prosoft/resources/2021/emse-mcr-prediction-models>

Consequently, in RQ3, we explore the timeframe that leads to the best results to predict reviewer participation and contribution, considering the results obtained in RQ1 and RQ2.

6.2.2 Procedure

Given our research questions, we now describe the procedure we followed to answer them. We first describe our two target predictions associated with reviewer recommendation. Then, we detail the adopted learning algorithms and associated performance metrics. Next, we present the comparisons made for each research question. Finally, we characterize our dataset.

6.2.2.1 Prediction Models

Most of the reviewer recommenders have been evaluated with the goal of finding the reviewers that actually reviewed a code change. However, there might be other developers, not invited for the review, who could have contributed—and perhaps these recommendations would have been useful because they are not obvious and indicate reviewers that would not be invited without a recommendation. We envision that different predictions, such as whether a reviewer will participate in a review or the delay to provide feedback, are helpful to build a reviewer recommender by combining their predictions. Therefore, in this work, we evaluate the use of the proposed features to build two prediction models, described as follows.

Reviewer Participation. The goal of this prediction model is, similarly to previous work, to identify the reviewers that participated in a review, using our proposed features. It is modeled as a binary classification problem. The proposed features are used to build a model that is able to predict a target feature, which in this case indicates whether a developer participated in a specific review (target feature is true) or not (target feature is false). Three classification algorithms are explored to train this prediction model: (i) Linear Support Vector Classification (SVC); (ii) Logistic Regression; and (iii) Random Forest.

Reviewer Feedback. Not only is it important for reviewers to participate in reviews, but also to provide valuable feedback. We assume that providing more comments is a way of measuring this. Our second prediction model is framed as a regression problem, with the target feature being the number of comments provided by a reviewer

in a specific review. Note that the number of comments has a skewed distribution (most of the reviews have 2–3 comments and only a few have a higher number of comments). Therefore, the distribution of the number of comments is normalized with a log function. Three commonly used regression algorithms are used to train this prediction model: (i) k-Nearest-Neighbors (kNN); (ii) Linear Regression; and (iii) Random Forest Regressor.

In both cases, each training example of the respective dataset for supervised learning corresponds to a review made by a particular reviewer. The provided features are the three proposed sets of features (CO, WL, and TR) together with two baseline features. The first is solely *CO - File Reviewer*, because this refers to the criterion used in *RevFinder* (THONGTANUNAM et al., 2014), which focused on the experience while reviewing files with similar paths. *RevFinder* has been the baseline approach used by the majority of existing techniques proposed to recommend reviewers. The second is *Lines of Code* (LOC), justified by previous studies that identified that the number of changed LOC impacts on both the reviewer participation and amount of feedback (SANTOS; NUNES, 2017). Moreover, Baysal et al. (2016) observed that changes that modified more LOC usually take more review rounds to be completed, where each review round is typically the result of addressed feedback or comments. Although our dataset includes, for each training example the identifier of the code review and the reviewer, we do not include them as features. The goal is not to make predictions for a specific developer, but to identify characteristics of developers that make them suitable to review a certain code change. This also makes the models suitable to cope with changes in the development team. In Figure 6.2, we summarize our two prediction models. It overviews the involved features as well as the selected algorithms.

For each prediction model, we select three algorithms after exploratory tests using our dataset. Given the amount of available data and the number of required training and evaluation runs for each research question, we discarded algorithms with high computational cost. After that, we selected algorithms based on their complexity in terms of the number of hyperparameters, which influences the time needed for hyperparameter optimization.

We used, for all algorithms, the implementations provided by Scikit-learn (PEDREGOSA et al., 2011). The hyperparameters were defined using a grid search approach, with F1 and R^2 as the scoring metrics for Reviewer Participation and Reviewer Feedback, respectively.

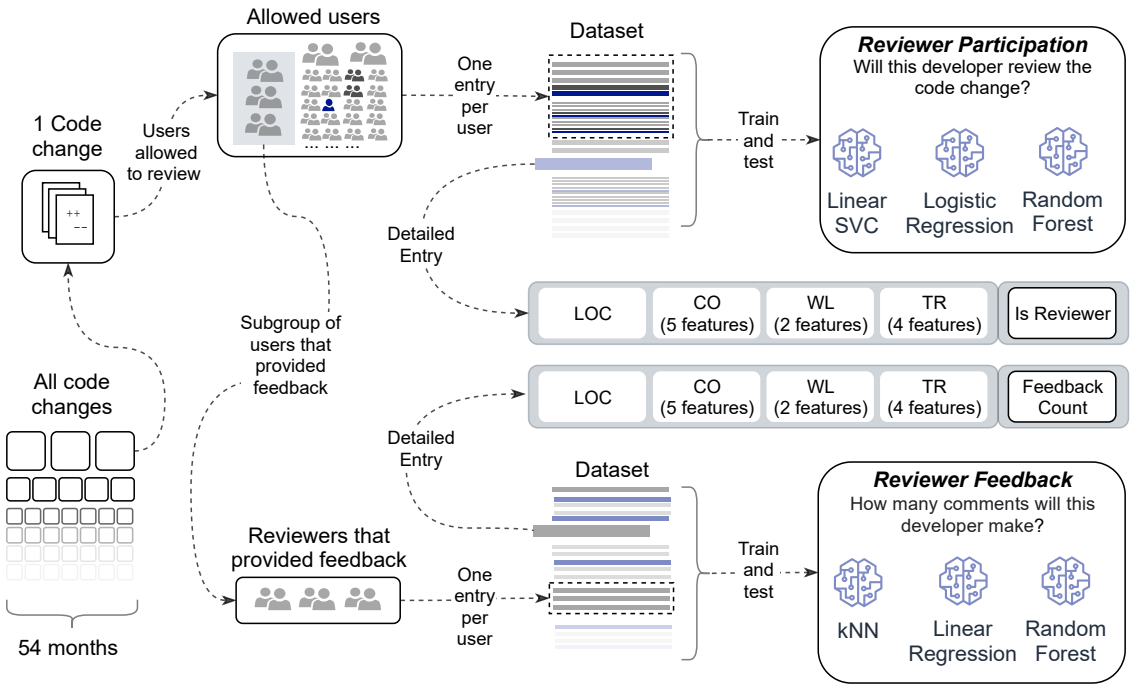


Figure 6.2 – Overview of our prediction models: Reviewer Participation and Reviewer Feedback.

6.2.2.2 Performance Metrics

Classifiers and regressors are implemented using different algorithms, such as those mentioned above, and have their performance measured with different metrics. The prediction of reviewer participation is a classification problem. To evaluate the obtained results, we consider the metrics *precision* (Pr), *recall* (Re), *f-measure* (F1), and *area under precision-recall curve* (AUPRC). The first three are widely used in this context, and these are calculated considering the identification of the reviewers that actually participated in the review as the positive class. AUPRC is also often used, and is a more suitable metric than area under receiver operating characteristic (AUROC) curve for imbalanced classes (SAITO; REHMSMEIER, 2015). This is our case because the number of reviewers that actually participated in a review is small in comparison to the total number of developers.

For the regression problem, i.e. the prediction of the amount of feedback, three other metrics are used: (i) *root mean square error* (RMSE), which measures the differences between the predicted values and the values observed; (ii) *Pearson correlation coefficient* (r), which measures the strength of a linear association between two variables; and (iii) coefficient of determination (R^2), used to measure the proportion of the variance in the target variable that is predictable based on the features. In other words, R^2 measures how good a linear model explains the variance in the target variable based on its features.

Although these metrics are correlated to each other, they provide different perspectives to analyze and interpret the results.

6.2.2.3 Comparisons

In order to answer our research questions, we elaborated a three-step study procedure. We make predictions using various configurations as summarized in Table 6.4. Further details are provided as follows.

The prediction power of our proposed feature sets (RQ1) is evaluated by building models using: (i) LOC (baseline 1); (ii) the File Reviewer (FR) metric (baseline 2); (iii) CO features; (iv) WL features; (v) TR features; (vi) proposed features (CO+WL+TR); and (vii) all features (CO+WL+TR+LOC). Note that the feature used in baseline 1 (selected based on *RevFinder*) is included in the CO feature set.

In addition to exploring different algorithms and using a set of performance metrics, both previously described, we must also deal with the class imbalance in our dataset. Typically, at most five from a set of $\sim$200 developers participate in a code review in our dataset. Therefore, the number of training examples in the negative class is much higher than the number of training examples in the positive class. In principle, to cope with class imbalance, it is possible to oversample the minority (i.e. positive) class and undersample the majority (i.e. negative) class. In our setting, it does not make sense to oversample the positive class, which would mean that the same reviewer participated in a code review more than once. Therefore, we adopt undersampling, exploring multiple rates (5%, 10%, 15%, 20%, 25%, and 50%). For example, an undersampling rate of 10% means that only 10% of the training examples of the majority class are considered, and 90% are discarded. This is only done in the training data so that the learning algorithm is able to build a model that is able to distinguish the positive and the negative classes.

RQ1 allows to understand the contribution of the proposed feature sets to build our two prediction models. Nevertheless, providing an optimal prediction model requires us to explore which subset of features provides the best results (RQ2). To select the best features, many automated approaches can be used to evaluate different feature sets while maximizing a scoring metric, e.g. F1.

Because of our number of features (13, in total), there are approaches that are computationally expensive, such as an exhaustive evaluation of all combinations of features. We use a recursive feature elimination (RFE) (GUYON et al., 2002) approach for this purpose, which starts with all features and tries to remove one feature at a time, checking

Table 6.4 – Summary of the execution configurations by research question and prediction model.

RQ	Prediction Model	Features	Variations	Algorithms	Metrics	#Configurations
RQ1	Reviewer Participation	LOC FR CO WL	5%, 10%,15%, 20%,25%, 50% (sampling rates)	Random Forest Linear SVC Logistic Regression	Pr Re F1 AUPRC	108
	Reviewer Feedback	TR Proposed features All features	-	kNN Linear Regression Random Forest	RMSE r R^2	18
RQ2	Reviewer Participation	All features	-	Best F1 in RQ1 including sampling rate	F1	1
	Reviewer Feedback		-	Best R^2 in RQ1	RMSE R^2	2
RQ3	Reviewer Participation	Features selected in RQ2	3, 6, 9, 12 (months)	Best F1 in RQ1 including sampling rate	Pr Re F1 AUPRC	4
	Reviewer Feedback	Features selected in RQ2		Best F1 in RQ1	RMSE r R^2	4

if a selected scoring metric is improved. For predicting participation, we maximized F1, while for predicting the amount of feedback, we used both RMSE and R^2. Although RFE is less computationally costly than an exhaustive approach, running all the configurations of RQ1 is still challenging. Therefore, we select the learning algorithm and undersampling rate that achieve the best results in RQ1. A custom cross-validation strategy is also used because our dataset consists of code review data associated with events in a specific time sequence, so a k-fold cross-validation strategy would use data from future events to predict the past, which is incorrect. Finally, for both prediction models, we use the `RFECV` method provided by the feature selection module of Scikit-learn.

The last analysis we perform (RQ3) is related to another issue that might influence the prediction results, namely the amount of past data used to make predictions. This is an issue that has not been evaluated in previous work that aimed at recommending reviewers. We evaluate the results obtained with four timeframes (3, 6, 9, and 12 months) of past data. For each timeframe, we build and test a learning model with 5 distinguished periods of past data, as detailed in the next section. As this also leads to many configurations to evaluate, we select the algorithm, undersampling rate, and feature subset based on the results of RQ1 and RQ2. This research question allows us to understand if higher computational cost should be spent to build models (i.e. use more data) or using only more recent data produces as good, or even better, results.

6.2.3 Dataset

The datasets used for the development of our approach and its evaluation were obtained from a proprietary project from a software company. We extracted information from October 2014 to March 2019 (54 months). During this period, 260 developers of 21 teams from 4 locations (located in different cities) participated in 21,796 code reviews of 380 modules, containing proprietary source code written in C, C++, Python, YANG, and Lua. All modules are part of an operating system of network devices, such as switches and routers. This dataset is built using two types of information: *code review data* and *organizational data* related to developers and managers, which are detailed next.

Code review data inform how authors and reviewers interacted in every source code change in all repositories, including the list of changed files, comments, replies, and votes, for instance. These data were obtained directly from the databases provided by

Gerrit[2]. Each training example of the dataset only contains information associated with code review records that were available at the point in time of the example. That is, we were careful to respect temporal aspects while building the dataset. To build the dataset used in this work, we discarded code reviews from repositories that only contain documentation, Infrastructure as Code (IaC), and third-party code. Moreover, we discarded changes with more than 5000 lines of code (usually, migrations from other repositories) and reviews that lasted for more than 30 days—this refers to staled work in our project. Furthermore, changes created by *bots* were discarded, as well as the feedback provided by them. The use of a bot is common (an automated reviewer), which checks the code changes for the compliance of standards and conventions in the organization or project that can be automated with tools and scripts.

Organizational data, in turn, required higher effort to be obtained. In short, it can be seen as records associating developers to their managers (or leaders) during a specific interval between two dates, as exemplified in Appendix B using anonymized data. We extracted information from project management tools to create these records and then refined and checked this information with interviews with managers and human resources staff. This information is not trivial to be obtained in some situations. For instance, multiple developers changed from a team to another, while other developers worked more than once in the company in different periods.

To develop our approach we used data from 2014 to 2016, testing different alternatives. These data were not used in our evaluation. The remaining data—from 2017 to 2019—was split into training, test, and validation sets for each research question, as detailed in Figure 6.3. Note that in RQ3, we evaluate the predictions using different timeframes of past data. We measured the predictions made for five releases (each roughly consisting of three months), using 3, 6, 9, and 12 months of past data. Three months is roughly the frequency of the releases in our target project. Therefore, we explore the use of data from the 1–4 previous release periods to predict the next one.

6.3 Results and Analysis

Given the description of our study procedure, we now present and discuss our results answering our research questions.

[2]<https://gerrit-review.googlesource.com/Documentation/cmd-query.html#_schema>

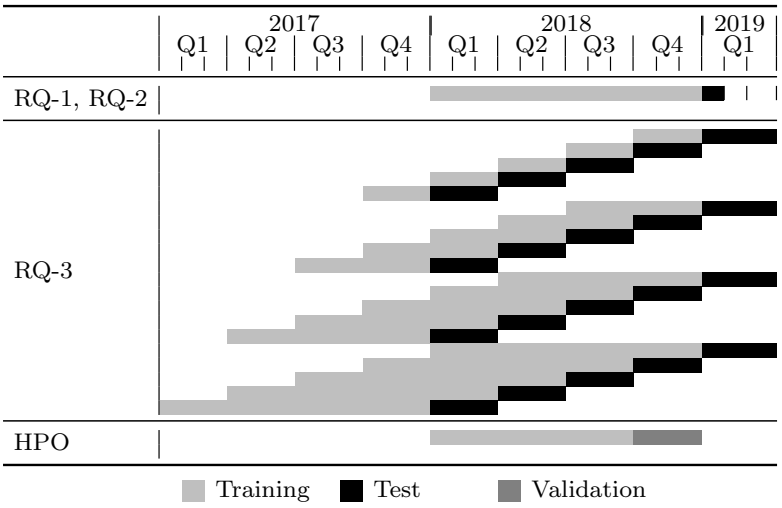


Figure 6.3 – Periods of time used to evaluate each research question.

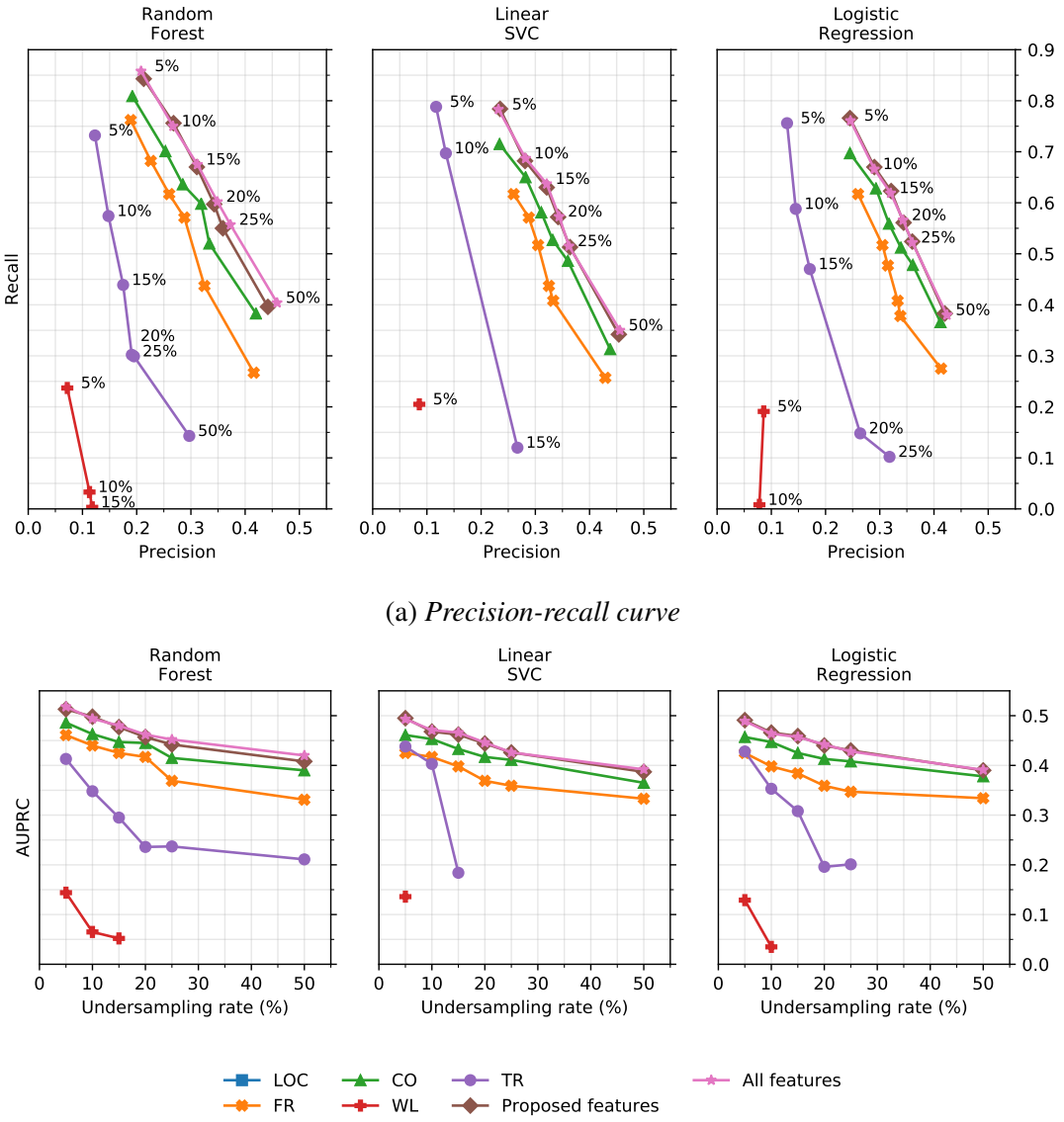
6.3.1 RQ1: Prediction Power of the Feature Sets

Our first analysis consists of evaluating how the different groups of proposed features—Code Ownership (CO), Workload (WL), and Team Relationship (TR)—perform individually and combined to predict reviewer participation and reviewer feedback. Given that we need to handle unbalanced classes in the former, we separately discuss the results obtained with each prediction model as follows.

6.3.1.0.1 Reviewer Participation. The use of our proposed features for predicting reviewer participation led to varying results depending on the used sampling rate and learning algorithm. We present these results in the charts in Figure 6.4 and in detail in Table 6.5. Note that for certain combinations of feature set and algorithm, there are no results. This is the case when the features do not provide enough information for the algorithm to distinguish the two classes and classify all instances with the majority class. Consequently, the performance metrics are undefined because the denominator is zero.

A first observation is that there is a trade-off between precision and recall based on the sampling rate (see Figure 6.4a[3]). The higher the sampling rate, the higher the precision; while the higher the sampling rate, the lower the recall. This can also be seen in the values of AUPRC presented in Figure 6.4b. This trade-off is in general expected when undersampling the negative majority class. When the undersampling of the negative majority class is stronger, there are relatively more positive examples in the training data,

[3]For readability, the sampling rates are shown in only one of the lines. The other lines follow a similar pattern and the sampling rates appear in the same order as those explicitly shown in the chart.



(a) *Precision-recall curve*

(b) AUPRC

Figure 6.4 – Results obtained with the different sets of features for predicting reviewer participation.

Table 6.5 – Results obtained with the different sets of features for predicting reviewer participation.

Metric	Feature Group	Random Forest Undersampling rate						Linear SVC Undersampling rate						Logistic Regression Undersampling rate					
		5%	10%	15%	20%	25%	50%	5%	10%	15%	20%	25%	50%	5%	10%	15%	20%	25%	50%
Precision	LOC	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	FR	0.19	0.23	0.26	0.29	0.33	0.42	0.26	0.29	0.30	0.33	0.33	0.43	0.26	0.30	0.32	0.33	0.34	0.41
	CO	0.19	0.25	0.28	0.32	0.33	0.42	0.23	0.28	0.31	0.33	0.36	0.44	0.24	0.29	0.32	0.34	0.36	0.41
	WL	0.07	0.11	0.12	0.00	1.00	0.00	0.09	0.00	0.00	0.00	0.00	0.00	0.09	0.08	0.00	0.00	0.00	0.00
	TR	0.12	0.15	0.17	0.19	0.20	0.30	0.12	0.14	0.27	0.00	0.00	0.00	0.13	0.14	0.17	0.26	0.32	0.00
	Proposed Features	0.21	0.27	0.31	0.34	0.36	0.44	0.23	0.28	0.32	0.34	0.36	0.45	0.24	0.29	0.32	0.34	0.36	0.42
	All Features	0.21	0.27	0.31	0.35	0.37	0.46	0.23	0.28	0.32	0.34	0.36	0.46	0.25	0.29	0.32	0.34	0.36	0.42
Recall	LOC	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	FR	0.76	0.68	0.62	0.57	0.44	0.27	0.62	0.57	0.52	0.44	0.41	0.26	0.62	0.52	0.48	0.41	0.38	0.28
	CO	0.81	0.70	0.64	0.60	0.52	0.38	0.71	0.65	0.58	0.53	0.49	0.31	0.70	0.63	0.56	0.51	0.48	0.37
	WL	0.24	0.03	0.00	0.00	0.00	0.00	0.20	0.00	0.00	0.00	0.00	0.00	0.19	0.01	0.00	0.00	0.00	0.00
	TR	0.73	0.57	0.44	0.30	0.30	0.14	0.79	0.70	0.12	0.00	0.00	0.00	0.76	0.59	0.47	0.15	0.10	0.00
	Proposed Features	0.84	0.76	0.67	0.60	0.55	0.40	0.78	0.68	0.63	0.57	0.51	0.34	0.77	0.67	0.62	0.56	0.52	0.38
	All Features	0.86	0.75	0.68	0.60	0.56	0.40	0.78	0.69	0.64	0.57	0.52	0.35	0.76	0.67	0.62	0.57	0.52	0.38
F1	LOC	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
	FR	0.30	0.34	0.37	0.38	0.37	0.33	0.37	0.38	0.38	0.37	0.37	0.32	0.37	0.38	0.38	0.37	0.36	0.33
	CO	0.31	0.37	0.39	0.42	0.41	0.40	0.35	0.39	0.41	0.41	0.41	0.36	0.36	0.40	0.41	0.41	0.41	0.39
	WL	0.11	0.05	0.01	-	0.00	-	0.12	-	-	-	-	-	0.12	0.01	-	-	-	-
	TR	0.21	0.23	0.25	0.23	0.24	0.19	0.20	0.23	0.17	-	-	-	0.22	0.23	0.25	0.19	0.15	-
	Proposed Features	0.34	0.40	0.42	0.44	0.43	0.42	0.36	0.40	0.42	0.43	0.43	0.39	0.37	0.41	0.42	0.43	0.43	0.40
	All Features	0.34	0.39	0.43	0.44	0.45	0.43	0.36	0.40	0.43	0.43	0.42	0.40	0.37	0.40	0.42	0.43	0.43	0.40
AUPRC	LOC	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
	FR	0.46	0.44	0.42	0.42	0.37	0.33	0.42	0.42	0.40	0.37	0.36	0.33	0.42	0.40	0.38	0.36	0.35	0.33
	CO	0.49	0.46	0.45	0.45	0.41	0.39	0.46	0.45	0.43	0.42	0.41	0.36	0.46	0.45	0.42	0.41	0.41	0.38
	WL	0.14	0.07	0.05	-	-	-	0.14	-	-	-	-	-	0.13	0.04	-	-	-	-
	TR	0.41	0.35	0.29	0.24	0.24	0.21	0.44	0.40	0.18	-	-	-	0.43	0.35	0.31	0.20	0.20	-
	Proposed Features	0.51	0.50	0.48	0.46	0.44	0.41	0.49	0.47	0.46	0.44	0.43	0.39	0.49	0.47	0.46	0.44	0.43	0.39
	All Features	0.52	0.49	0.48	0.46	0.45	0.42	0.49	0.47	0.47	0.45	0.43	0.39	0.49	0.46	0.46	0.44	0.43	0.39

which helps the model to better identify the positive cases. It may, in fact, also learn to predict the positive class more often in general. Thus, the true positive (TP) cases are found more often, and fewer cases are falsely predicted to be negative (FN). Given Recall = TP / (TP + FN), we thus obtain higher recall as TP increases and FN decreases. At the same time, however, the number of False Positives (FP) may increase (as the model somehow may learn to predict the positive class too often from the undersampled data). Since Precision = TP / (TP + FP), we observe a drop in precision due to the increased FP and a probably lower rate of identified TPs.

Regarding the alternative algorithms (Random Forest, Linear SVC, and Logistic Regression), although they lead to different absolute results, the relative performance of the different configurations is consistent in all of them. This occurs not only for the varying sampling rates (as discussed above) but also for the investigated feature sets. Figure 6.4a shows that, despite that our baseline FR consists of a single feature, it achieves relatively good results. This confirms that previous work (THONGTANUNAM et al., 2014), which often relies on previous reviewers of a particular file, exploited a feature that indeed is relevant for recommending reviewers. Our second baseline (LOC), however, achieved poor results. For the prediction of reviewer participation this is expected because LOC influences mostly the review time and feedback (SANTOS; NUNES, 2017; BAYSAL et al., 2016) and this feature alone cannot predict reviewer participation. If so, it would be the case that reviewers with particular characteristics typically review longer (or shorter) code changes.

Focusing on our proposed set of features, it is possible to observe that CO achieves the best results, while TR and mainly WL achieve poor results. This indicates that being involved (as an author or a reviewer) with the files modified in a code review is the most important factor to participate in a review. Moreover, our additional features related to CO improve our baseline, with F1 improvements of up to 7% (mainly due to better results in the recall). Even though TR and WL alone poorly predict reviewer participation, when combined with CO, they increase F1 up to 3%. Because we do not evaluate all the combinations of features in this research question, it is not possible to understand if this increase is due to TR, WL, or both. This is investigated in the next research question by means of feature selection.

Finally, note that the best results are obtained with all features (proposed sets of features together with LOC). Consequently, LOC can aggregate some value in the prediction of reviewer participation. The best results, considering F1 that balances precision

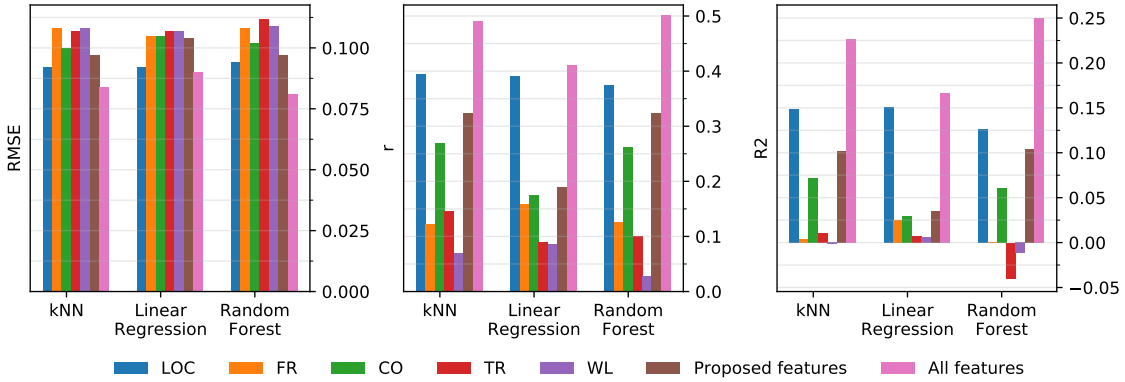


Figure 6.5 – Results obtained with the different sets of features for predicting reviewer feedback.

and recall, are obtained with the Random Forest algorithm, using a sampling rate of 25%. This leads to F1 = 0.45, precision = 0.37, and recall = 0.56. The best precision (0.46) is obtained with Random Forest or Linear SVC, and 50% of sampling rate, while the best recall (0.86) is obtained with Random Forest and 5% of sampling rate. Note that, although the best precision is lower than 0.5, it is better than a random prediction model because the prediction of reviewer participation is a problem that involves highly imbalanced classes.

6.3.1.0.2 Reviewer Feedback. Given that the prediction of reviewer feedback is a regression problem (rather than a classification problem as the prediction of review participation), we evaluate our proposed features sets using RMSE, r, and R^2. The obtained results are presented in the bar charts in Figure 6.5 and detailed in Table 6.6. Recall that the distribution of the number of comments have been normalized with a log function. Consequently, the absolute number of the error metrics is low (even rounded to zero). Moreover, the results of all feature sets are consistent across all learning algorithms, so we limit ourselves to explicitly state the numbers obtained with the algorithm with the best overall results (lowest RMSE and highest r and R^2), namely Random Forest, as follows.

While LOC poorly performs for predicting review participation, it achieves the best results for predicting reviewer feedback. It achieves the lowest RMSE (0.09) and highest r (0.37) and R^2 (0.13), considering individual feature sets. Previous work has shown, for example, that (very) large patches tend to receive limited feedback (SANTOS; NUNES, 2017). Similar to reviewer participation, CO achieves results that are better than the other sets. This indicates again that experience in the files to be reviewed has an important role. However, the difference between CO and FR is larger for reviewer feedback, which gives evidence that the experience as an author is crucial to be taken into account for this problem. For reviewer feedback, our baseline FR, differently from

Table 6.6 – Results obtained with the different sets of features for predicting reviewer feedback.

Algorithm	Metric	LOC	FR	CO	TR	WL	Proposed Features	All Features
kNN	RMSE	0.09	0.11	0.10	0.11	0.11	0.10	0.08
	r	0.39	0.12	0.27	0.15	0.07	0.32	0.49
	R^2	0.15	0.00	0.07	0.01	-0.00	0.10	0.23
Linear Regression	RMSE	0.09	0.10	0.10	0.11	0.11	0.10	0.09
	r	0.39	0.16	0.17	0.09	0.09	0.19	0.41
	R^2	0.15	0.03	0.03	0.01	0.01	0.04	0.17
Random Forest	RMSE	0.09	0.11	0.10	0.11	0.11	0.10	0.08
	r	0.37	0.13	0.26	0.10	0.03	0.32	0.50
	R^2	0.13	0.00	0.06	-0.04	-0.01	0.10	0.25

the previous prediction, achieves results that are similar to the worst results, which are obtained with TR and WL. WL, again, performs poorly. TR and WL have a negative R^2, meaning that these feature sets achieve a result worse than using the average. However, this poor performance is expected because the reviewer workload should not be a single factor for a reviewer to provide feedback (although it might influence it).

The proposed features combined are less powerful to predict reviewer feedback than LOC alone. This shows that LOC is indeed the main factor for reviewers to provide less or more feedback. However, by combining LOC with the other features, results are largely improved. All features combined achieves the as follows lowest RMSE = 0.08, and highest r = 0.50 and R^2 = 0.25.

RQ1 answer: Our baselines—reviewer experience in the reviewed files and LOC—are relevant features for predicting reviewer participation and reviewer feedback, respectively. Considering our proposed feature sets, those related to code ownership (including features that refer to authors) provide large improvements for both prediction models. While team relationship and workload features are individually not enough to build these models, they improve both of them. The Random Forest algorithm led to the best results for predicting both reviewer participation and reviewer feedback. Decreasing the sampling rate to deal with the imbalanced classes in the former prediction model increases recall, but decreases precision. Considering the trade-off between them, as reported by F1, a 25% sampling rate has been shown to be the optimal value using our dataset.

6.3.2 RQ2: Feature Selection

Our previous research question allowed us to understand to what extent the proposed feature groups have the potential to lead to good predictions related to code review. However, not all features in each group might be needed to achieve the best prediction results. To investigate which features produce the optimal results for each of our prediction models, our second research question focuses on feature selection. Because feature selection is computationally costly to be performed, we take into account the results of RQ1. We perform feature selection using the algorithms and sampling rate that led to the best results, namely Random Forest (for both prediction models) and 25% as an undersampling rate. Moreover, as discussed, we adopt a commonly used feature selection algorithm—recursive feature elimination (RFE)—instead of an exhaustive evaluation, which is computationally challenging. As detailed in our study procedure, the goal is to maximize F1 and R^2, for predicting reviewer participation and reviewer feedback, respectively.

6.3.2.0.1 Reviewer Participation. After applying RFE, we found that no feature is suggested to be eliminated for predicting reviewer participation, that is, all features are reported as relevant. Because RFE does not explore feature combinations, we further investigated the relative importance of our features as a refinement step. The usual metrics that indicate relative importance are: (i) Information Gain; (ii) Gini Importance; (iii) Gain Ratio (QUINLAN, 1986); and (iv) Chi^2 (ROKACH; MAIMON, 2005). The obtained results are shown in Table 6.7. These metrics indicate that, although some features are more important than others, no feature is irrelevant, which is consistent with the RFE results.

6.3.2.0.2 Reviewer Feedback. Differently, RFE reported that two features can be removed for the prediction of reviewer feedback, namely Same Team and Same Location. To refine these results, similarly as above, we made an additional investigation of the relative feature importance. Given that reviewer feedback is a regression problem, we used in this case the RReliefF algorithm (ROBNIK-SIKONJA; KONONENKO, 1997). As can be seen in Table 6.7, both SameTeam and SameLocation have a zero score, which indicates that they are not relevant. This is consistent with the RFE results. Based on these results, we compare the performance in terms of RMSE, r and R^2, when predicting reviewer feedback with all features and with all features except Same Team and Same Location, i.e. those that could be eliminated. Table 6.8 shows that there is only a marginal

Table 6.7 – Analysis of the relative importance of features for predicting reviewer participation and reviewer feedback. Features removed by recursive feature elimination are highlighted in gray.

Feature	Reviewer Participation				Reviewer Feeback
	Info. gain ($\times 10^{-2}$)	Gini ($\times 10^{-3}$)	Gain Ratio ($\times 10^{-3}$)	Chi^2	RReliefF ($\times 10^{-2}$)
ChangedLOC	0.10	0.1	0.5	42	8.71
File Reviewer	7.14	14.7	142.3	21364	6.21
File Author	4.22	9.8	133.7	14822	4.80
Module Reviewer	7.46	12.7	90.0	16268	5.63
Module Author	5.90	10.6	87.0	14807	5.38
Is Maintainer	2.73	6.2	153.9	4090	0.08
Author Workload	0.55	0.5	3.8	448	6.87
Reviewer Workload	1.80	1.5	9.6	1149	8.17
Same Team	5.31	7.1	95.8	4198	0.00
Same Location	1.91	1.3	19.1	456	0.00
Team Interactions (Rev)	4.40	4.7	23.2	2363	4.23
Team Interactions (Aut)	4.45	5.1	24.8	3061	4.63

Table 6.8 – Analysis of the gains when predicting reviewer feedback by removing features based on the RFE results.

Metric	All features	After RFE	Difference
RMSE	0.08	0.08	0.000
r	0.50	0.50	0.000
R^2	0.25	0.25	**+0.001**

gain by removing these features and only in terms of the R^2 measure.[4] This suggests that the features, when present, do not introduce noise.

RQ2 answer: No feature from any of the investigated feature groups has been eliminated by the recursive feature elimination (RFE) algorithm, except Same Team and Seam Location (from the Team-related feature set), for predicting review feedback. However, after further examination, we could not determine specific features that, when removed, would improve the performance of the analyzed models. Although some features have more relevance, every feature provides at least a small performance improvement. The most important features for predicting reviewer participation are having authored or reviewed the file or module in the past, and the number of changed LOC, while having reviewed the file and the reviewer workload are the most important features for predicting reviewer feedback.

[4]We also made this comparison using kNN and Linear Regression, which led to similar results.

6.3.3 RQ3: Timeframes of Past Data to Build Models

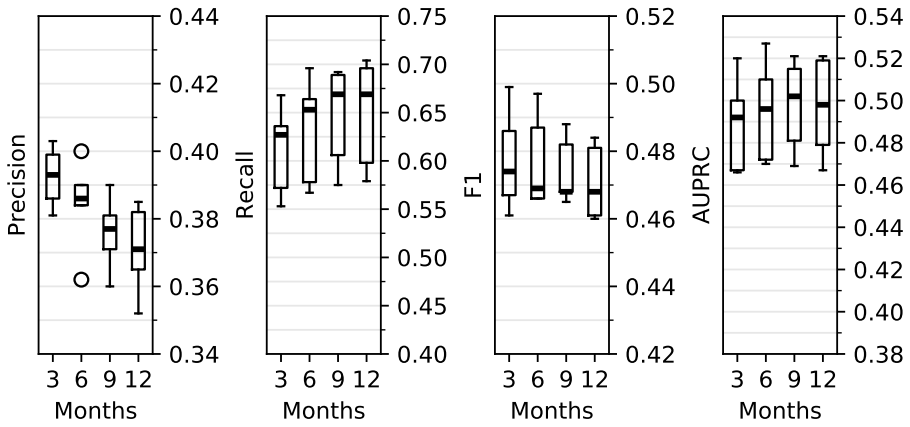
Reviewer recommenders were built and evaluated using different datasets, as reported in the literature review by Davila and Nunes (2021). The results of these studies provide evidence of the performance of features, algorithms, and proposed heuristics to employ reviewer recommenders in specific software projects. In real settings, data of a particular project must be typically used to build a suitable model to make recommendations in that context. However, no previous study investigated the impact of the amount of past data on the performance of reviewer recommenders.

In RQ3, we thus make such an analysis, investigating the effect of using four alternative timeframes (3, 6, 9, and 12 months) of past data to make future predictions. As this exploration already involves running different alternatives, we use the results of RQ1 and RQ2 to select the learning algorithm and features. As our previous results showed, Random Forest, all features, and 25% as an undersampling rate (for predicting review participation) lead to the best performance. Each timeframe alternative is evaluated five times, as detailed in Figure 6.3, predicting the next release period based on past data.

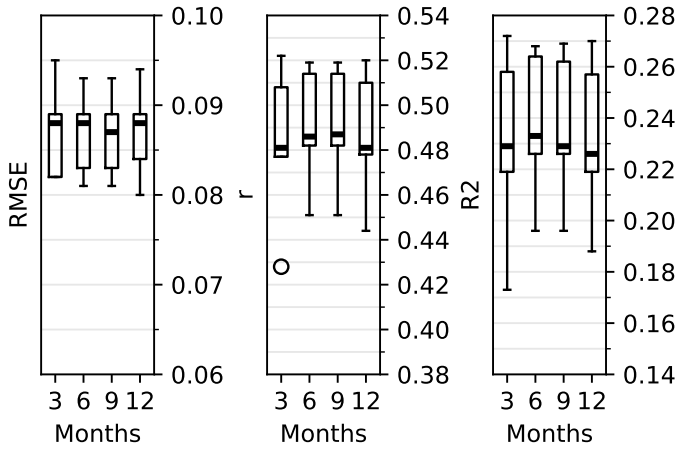
The results of exploring the different timeframes are shown in Figure 6.6b and further detailed in Table 6.9.[5] By analyzing in Figure 6.6a the performance when predicting reviewer participation, we observe that shorter timeframes of past data achieve higher precision. However, the difference is marginal—the average precision is 0.39 with 3 months and 0.37 with 12 months. Similar behavior but in the opposite direction is observed with respect to recall—the average recall is 0.61 with 3 months and 0.65 with 12 months. As a consequence, F1 is similar across different timeframes. This indicates that new projects can rely on prediction models having data of only a single past release and models can be trained frequently and with lower cost (less data).

Similar conclusions are reached by analyzing the prediction of the amount of review feedback, as can be seen in Figure 6.6b. There is negligible variance in the results obtained with the various timeframes. In fact, the performance achieved in each of the five predicted periods is similar for all timeframes, e.g. R^2 is lower in the second period for all number of months. This corroborates our findings that computational resources can be saved by training models with less data, and eventually update them more frequently.

[5]We do not run statistical tests to test the significance of differences among average values of the performance metrics because there are only five values for each explored timeframe. Nevertheless, the presented results can be analyzed individually (Table 6.9) and trends can already be observed in the results. Note that obtaining past data related to employment history is not trivial in software projects and this imposes challenges in making studies covering larger periods.



(a) *Reviewer Participation*



(b) *Reviewer Feedback*

Figure 6.6 – Comparison of the performance achieved with alternative timeframes of past data to predict reviewer participation and review feedback.

Table 6.9 – Values of performance metrics when predicting reviewer participation and review feedback using different timeframes of past data.

	Metric	Months	Period 1	2	3	4	5
Reviewer Participation	Precision	3	0.39	0.40	0.38	0.40	0.39
		6	0.40	0.39	0.36	0.39	0.38
		9	0.39	0.38	0.36	0.38	0.37
		12	0.38	0.39	0.35	0.37	0.37
	Recall	3	0.57	0.55	0.63	0.67	0.64
		6	0.57	0.58	0.65	0.70	0.66
		9	0.58	0.61	0.67	0.69	0.69
		12	0.58	0.60	0.67	0.70	0.70
	F1	3	0.46	0.47	0.47	0.50	0.49
		6	0.47	0.47	0.47	0.50	0.49
		9	0.47	0.47	0.47	0.49	0.48
		12	0.46	0.47	0.46	0.48	0.48
	AUPRC	3	0.47	0.47	0.49	0.52	0.50
		6	0.47	0.47	0.50	0.53	0.51
		9	0.47	0.48	0.50	0.52	0.52
		12	0.47	0.48	0.50	0.52	0.52
Reviewer Feedback	RMSE	3	0.08	0.10	0.08	0.09	0.09
		6	0.08	0.09	0.08	0.09	0.09
		9	0.08	0.09	0.08	0.09	0.09
		12	0.08	0.09	0.08	0.09	0.09
	r	3	0.51	0.43	0.52	0.48	0.48
		6	0.51	0.45	0.52	0.48	0.49
		9	0.52	0.45	0.51	0.49	0.48
		12	0.52	0.44	0.51	0.48	0.48
	R^2	3	0.26	0.17	0.27	0.22	0.23
		6	0.26	0.20	0.27	0.23	0.23
		9	0.27	0.20	0.26	0.23	0.23
		12	0.27	0.19	0.26	0.22	0.23

RQ3 answer: Using different amounts of past data to predict reviewer participation and reviewer feedback has marginal impact on the obtained results. Consequently, models can be trained with less computational resources and more frequently. Moreover, new projects (with a single past release) can already benefit from these prediction models to select reviewers. However, using shorter timeframes slightly increased precision but slightly decreased recall to predict reviewer participation. Therefore, if precision is a priority in a particular project, longer timeframes can be used to predict reviewer participation.

6.4 Discussion

Our study allowed us to assess the value of the proposed features and the impact of different timeframes of past data on predictors related to the selection of code reviewers. Based on the obtained results, there are important issues to be considered in the development of reviewer recommends. These are discussed in this section together with the threats to the validity of our study.

Building Reviewer Recommenders Most of the existing code recommenders aim to identify the reviewers that actually participated in a past review. However, as in typical recommender systems (GE; DELGADO-BATTENFELD; JANNACH, 2010), accuracy (according to past data) might not be the best measurement to assess the performance of reviewer recommenders. Therefore, in this work, we focused on predicting components, namely reviewer participation and reviewer feedback, that are helpful to choose a reviewer. Moreover, we also made sure that our models do not learn that a particular reviewer is suitable for a particular review (by considering the reviewer identifier in the learned model). Instead, our features consist only of general characteristics of the reviewer. Nevertheless, using our models to build a review recommender is left as an open issue. The results of various predictors can be combined in different possibilities—such as building many rankings and combining them with a social choice strategy—so as to construct a sophisticated reviewer recommender.

Tailoring Reviewer Recommenders for Specific Needs As we discussed, there are features, learning algorithm, undersampling rate, and timeframe that lead to the overall optimal results. However, trade-offs must be made, mainly related to precision and recall. As shown in Figure 6.4, for example, the higher the undersampling rate, the lower the

precision, but the higher the recall. For making a choice, we considered the F1 measure, which is the harmonic mean between these two measurement. Consequently, by making a choice based on F1, we are considering the trade-off between precision and recall. However, software projects may have specific needs. As a consequence, when building a reviewer recommender for a particular project, one might select other parameters (such as another undersampling rate) in order to prioritize another performance metric.

Impact of Large Organizational Changes We collected code review data of a particular software company, covering a period of 54 months (starting in October 2014). When analyzing the data, we observed major changes. During this period, two events affected all developers, causing a significant reorganization in modules, teams, managers, and locations. For instance, in August 2016, a development location was shut down, which lead to merges in teams and ownership changes for several modules. Given that both events affected teams and ownership of modules, which are the aspects we explore in this work, these team and module reorganizations might largely impact on the results and be a threat to its validity. Therefore, we selected a subset of the data to be used for the execution of our study. Nevertheless, software companies are susceptible to this kind of event. Thus, it is interesting to investigate in future work how this kind of major changes in the data affect the predictions and reviewer recommenders.

Use of Cross-project Data The focus of our study is to use data from a particular project to make predictions for this project. Although our results show that collecting data from a single 3-month software release is enough to make predictions, we did not evaluate how data from one project can be used to make predictions for another project. Given that our features refer to general characteristics of reviewers and the code, it is possible to create a dataset with data from various projects. However, it is not clear if this results in a good performance because each project may have a particular behavior. This might be interesting, for example, if a company have a single dataset with code review data and developers can be assigned to different projects overtime.

Threats to Validity The evaluation of our proposed was done by means of an empirical study and, as such, there are threats to its validity. Threats to construct validity are related to how we designed the experiment. In order to address that, we adopted a series of design choices to guarantee the validity of our results: (1) we selected different widely used learning algorithms; (2) we optimized their parameters; (3) we executed our evaluation with Python scripts but also verified the correction of the scripts by executing

the study also in Orange [6]; (4) we used widely used metrics to analyze our results; (5) we considering how classes are balanced in the classification problem; (6) we did not biased our results by using reviewer identifiers in the dataset; and (7) we respected the temporal aspects of the data. Regarding the threat to external validity, we understand that the results use data from a single project. We highlight that the project from which we extracted data is a typical software project, which used a common code review process. Therefore, it is representative and provides a large amount of code review data. Moreover, despite being a single project, our results brings novelty as it gathers data from companies' personnel, which is not trivial to be obtained for research purposes. Consequently, our study provides novel results, with evidence backed up by representative data. Nevertheless, To further validate the results, of course, more studies are needed with data from other projects.

6.5 Final Remarks

In this study, we proposed the use of three sets of features—namely code ownership features, workload features, and team relationship features—to build models able to predict reviewer participation and reviewer feedback. These features explore aspects that have not been previously taken into account in previously proposed recommenders, mainly because they are not available in open source projects, which have been used in the majority of previous studies. This study explores aspects that are crucial for identifying code reviewers in software projects that include developers assigned to software teams, which are responsible for particular project modules. This matches the reality of many closed software projects. By means of an empirical evaluation to assess the performance of our proposed features and make our target predictions, we reached the following conclusions.

- All three sets of features are relevant features for predicting reviewer participation and reviewer feedback, being the set of code ownership features able to achieve the best performance. A feature selection process showed that all these features should be used for both predictors, together with one of our baselines (lines of code).

- Among the investigated learning algorithms, random forest led to the best results (in both the classification and regression problems). Moreover, as in our classification problem (prediction of reviewer participation) we have an unbalanced dataset, using

[6] <https://orangedatamining.com/>

25% of undersampling rate achieves the best trade-off between precision and recall.

- By using different amounts of past data (3, 6, 9, and 12 months) to build a model and make predictions for the next 3 months ($\sim$ software release duration in the target project), it is possible to achieve similar prediction performance. There is, however, a trade-off between precision and recall—shorter periods lead to slightly higher precision and slightly lower recall. This indicates that it is possible to frequently update the learning models, with lower computational resources without loosing prediction performance.

We highlight that our study procedure was carefully designed to respect the restrictions of our problem domain. To build our dataset, we did not use the reviewer identifiers, considered the information that was actually available in a particular code review time (e.g. the number of reviews performed by a reviewer before the target review), and used as test and validation sets only future data.

In the next chapter, we conclude this dissertation by summarizing our contributions and presenting opportunities for future work.

7 CONCLUSION AND FUTURE WORK

Code review is an important and widely adopted static verification technique for improving software quality and promoting knowledge sharing and collective code ownership within a software project. It is essentially based on the technical collaboration of authors and reviewers and thus might face additional challenges in the context of distributed software development. However, few studies have investigated the influence of team-related aspects in code reviews. Moreover, finding suitable reviewers is a critical step of code reviews, and team-related information is not usually taken into account.

In this dissertation, we conducted studies to understand the effects of team-related aspects in modern code review. First, we used a data-centric approach, using information about teams (managers, experience, locations, etc.), mining software repositories, and code review databases for a software project with many teams and developers. Then, we surveyed code review practitioners to complement our findings in order to provide a developer-centric perspective of code reviews. Finally, based on the insights and lessons learned from these investigations, in our last study, we proposed team-related features and evaluated how they influence the performance of prediction models used to recommend reviewers in the context of distributed software development. This last study has shown the usefulness of our set of features to predict reviewer participation and reviewer feedback.

In the following sections, we detail our contributions and discuss future work.

7.1 Contributions

Below, we list the main contributions of this dissertation.

Quantitative Study Based on Repository Mining. In Chapter 4, we presented the results of a quantitative study (SANTOS; NUNES, 2017) based on data from a project with geographically distributed developers and many teams. Our results have shown the effects of factors such as teams and locations over key code review outcomes, such as its duration and number of comments provided by reviewers, providing a significant advance in the research on how to improve code review. Moreover, most similar studies use data from open-source projects, as it is readily available online. In contrast, this study uses a commercial target project, on which teams are well-defined in terms of their members, maintainers, and managers. Practitioners in similar contexts might benefit from the

insights and lessons learned.

Survey with Experienced Code Review Practitioners. In Chapter 5, we presented the results of a survey that aims to understand the inner workings of MCR from a developer's perspective. This study presented the developers' preferences in terms of code review outcomes, such as the number of reviewers and review duration. We also investigated how developers interact during code reviews and what motivates them to contribute as reviewers. Many software aspects are discussed during these investigations, from software architecture to reviewers' work acknowledgment by peers and managers. We believe the insights and conclusions from this study are useful to code review practitioners in corporate environments, managers, and software architects.

Implementation and Evaluation of Predictors for Code Review. In Chapter 6, we presented the results of a study (SANTOS; NUNES; JANNACH, 2021) that implemented and evaluated predictors for code review using team-related features. Our results suggest that these features are relevant to predict whether a given reviewer will review a code change, and how much feedback will be provided. This is a significant novelty, as the majority of the existing techniques to recommend reviewers do not take advantage of this type of information.

7.2 Future Work

In this dissertation, we provided relevant contributions towards the understanding and improvement of modern code review in the context of geographically distributed teams, especially in the context of software developed within organizations. Nevertheless, further research is necessary due to the limitations of our studies. We next list the opportunities we envisage for further research.

Use of Prediction Models for Reviewer Recommendations. Our work has shown the usefulness of our set of features to predict reviewer participation and reviewer feedback. However, how these must be used to integrate a reviewer recommender is left for future work.

User Studies to Evaluate Reviewer Recommenders. Given that good recommendations might include reviewers that have not actually reviewed a code change, user studies are needed to assess the effectiveness of reviewer recommenders.

Improve Reviewer Recommenders. Code review is known for its benefits for knowledge sharing due to the natural interaction of authors and reviewers during the pro-

cess. However, reviewer recommenders are not conceived to enforce recommendations that mix reviewers with different experience levels. Similarly, recommenders could be tailored to enforce a better workload balance, which was taken into account in our work.

Replication studies. Usually, FLOSS communities have readily available code review data for a large number of projects. However, having the same information concerning closed-source, proprietary software projects developed within organizations is a challenge. Both practitioners and the scientific community would benefit from the knowledge derived from replications of our two foundational studies in companies with well-defined team structures.

Given the three presented studies, this dissertation provided complementary perspectives of how modern code review is affected by team-related aspects, especially in the context of software developed within organizations with geographically distributed teams. Nevertheless, there is still space to investigate how the team-related aspects affect code review in other organizations and how recommenders can be exploited to help developers overcome daily code review challenges while preserving its quality benefits.

REFERENCES

AURUM, A.; PETERSSON, H.; WOHLIN, C. State-of-the-art: software inspections after 25 years. **Software Testing, Verification and Reliability**, Wiley Online Library, v. 12, n. 3, p. 133–154, 2002.

BACCHELLI, A.; BIRD, C. Expectations, outcomes, and challenges of modern code review. In: **Proceedings of the 2013 International Conference on Software Engineering**. Piscataway, NJ, USA: IEEE Press, 2013. (ICSE '13), p. 712–721. ISBN 978-1-4673-3076-3. Available from Internet: <http://dl.acm.org/citation.cfm?id=2486788.2486882>.

BALACHANDRAN, V. Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation. In: **Proceedings of the 2013 International Conference on Software Engineering**. Piscataway, NJ, USA: IEEE Press, 2013. (ICSE '13), p. 931–940. ISBN 978-1-4673-3076-3. Available from Internet: <http://dl.acm.org/citation.cfm?id=2486788.2486915>.

BASILI, V. R.; SELBY, R. W.; HUTCHENS, D. H. Experimentation in software engineering. **IEEE Trans. Softw. Eng.**, IEEE Press, Piscataway, NJ, USA, v. 12, n. 7, p. 733–743, jul. 1986. ISSN 0098-5589.

BAUM, T. et al. Factors influencing code review processes in industry. In: **FSE 2016**. [S.l.]: ACM, 2016. p. 85–96. ISBN 9781450342186.

BAYSAL, O. et al. Investigating technical and non-technical factors influencing modern code review. **Empirical Software Engineering**, Springer, v. 21, n. 3, p. 932–959, 2016.

BECK, K. **Extreme programming explained: embrace change**. [S.l.]: addison-wesley professional, 2000.

BELLER, M. et al. Modern code reviews in open-source projects: Which problems do they fix? In: **Proceedings of the 11th Working Conference on Mining Software Repositories**. New York, NY, USA: ACM, 2014. (MSR 2014), p. 202–211. ISBN 978-1-4503-2863-0. Available from Internet: <http://doi.acm.org/10.1145/2597073.2597082>.

BISANT, D. B.; LYLE, J. R. A two-person inspection method to improve programming productivity. **IEEE Transactions on Software Engineering**, IEEE Computer Society, v. 15, n. 10, p. 1294, 1989.

BOSCH, J. Software architecture: The next step. In: **Software Architecture**. [S.l.]: Springer Berlin Heidelberg, 2004. p. 194–199. ISBN 978-3-540-24769-2.

BOSU, A.; GREILER, M.; BIRD, C. Characteristics of useful code reviews: An empirical study at microsoft. In: IEEE. **Mining Software Repositories (MSR), 2015 IEEE/ACM 12th Working Conference on**. [S.l.], 2015. p. 146–156.

CRASWELL, N. Mean reciprocal rank. In: ____. **Encyclopedia of Database Systems**. Boston, MA: Springer US, 2009. p. 1703–1703. ISBN 978-0-387-39940-9. Available from Internet: <https://doi.org/10.1007/978-0-387-39940-9_488>.

CZERWONKA, J.; GREILER, M.; TILFORD, J. Code reviews do not find bugs: How the current code review best practice slows us down. In: **Proceedings of the 37th International Conference on Software Engineering - Volume 2**. Piscataway, NJ, USA: IEEE Press, 2015. (ICSE '15), p. 27–28. Available from Internet: <http://dl.acm.org/citation.cfm?id=2819009.2819015>.

DAVILA, N.; NUNES, I. A systematic literature review and taxonomy of modern code review. **Journal of Systems and Software**, p. 110951, 2021. ISSN 0164-1212. Available from Internet: <https://www.sciencedirect.com/science/article/pii/S0164121221000480>.

FAGAN, M. E. Design and code inspections to reduce errors in program development. **IBM Syst. J.**, IBM Corp., Riverton, NJ, USA, v. 15, n. 3, p. 182–211, sep. 1976. ISSN 0018-8670. Available from Internet: <http://dx.doi.org/10.1147/sj.153.0182>.

FAGAN, M. E. Advances in software inspections. **IEEE Transactions on Software Engineering**, IEEE Press, v. 12, n. 1, p. 744–751, 1986.

FEJZER, M.; PRZYMUS, P.; STENCEL, K. Profile based recommendation of code reviewers. **Journal of Intelligent Information Systems**, Springer, v. 50, n. 3, p. 597–619, 2018.

FERREIRA, A. L. et al. An approach to improving software inspections performance. In: IEEE. **ICSM 2010**. [S.l.], 2010. p. 1–8.

GE, M.; DELGADO-BATTENFELD, C.; JANNACH, D. Beyond accuracy: Evaluating recommender systems by coverage and serendipity. In: **Proceedings of the Fourth ACM Conference on Recommender Systems**. New York, NY, USA: Association for Computing Machinery, 2010. (RecSys '10), p. 257–260. ISBN 9781605589060. Available from Internet: <https://doi.org/10.1145/1864708.1864761>.

GITHUB. **Github code review documentation**. 2017. [Online; accessed 07-Jun-2017]. Available from Internet: <https://github.com/features#code-review>.

GOOGLE. **Gerrit Code Review v2.11.2, Queries**. 2017. Available from Internet: <https://gerrit-documentation.storage.googleapis.com/Documentation/2.12.2/cmd-query.html>.

GOOGLE. **Gerrit Plugin: Reviewers by Blame**. 2017. Available from Internet: <https://gerrit-documentation.storage.googleapis.com/Documentation/2.12.2/cmd-query.html>.

GUYON, I. et al. Gene selection for cancer classification using support vector machines. **Machine Learning**, p. 389–422, 2002. ISSN 0885-6125.

HANNEBAUER, C. et al. Automatically recommending code reviewers based on their expertise: An empirical comparison. In: ACM. **Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering**. [S.l.], 2016. p. 99–110.

HUNDHAUSEN, C. D.; AGRAWAL, A.; AGARWAL, P. Talking about code: Integrating pedagogical code reviews into early computing courses. **Trans. Comput. Educ.**, ACM, New York, NY, USA, v. 13, n. 3, p. 14:1–14:28, aug. 2013. ISSN 1946-6226. Available from Internet: <http://doi.acm.org/10.1145/2499947.2499951>.

IEEE-1028. Ieee standard for software reviews and audits. **IEEE Std 1028-2008**, p. 1–52, Aug 2008.

(IETF), I. E. T. F. **RFC 6020: YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF)**. 2010. [Online; accessed 11-Jun-2016]. Available from Internet: <http://tools.ietf.org/html/rfc6020>.

JEONG, G. et al. Improving code review by predicting reviewers and acceptance of patches. **Research on software analysis for error-free computing center Tech-Memo (ROSAEC MEMO 2009-006)**, p. 1–18, 2009.

JIANG, J.; HE, J.-H.; CHEN, X.-Y. Coredevrec: Automatic core member recommendation for contribution evaluation. **Journal of Computer Science and Technology**, Springer, v. 30, n. 5, p. 998–1016, 2015.

KALMAN, D. A singularly valuable decomposition: the svd of a matrix. **The college mathematics journal**, Taylor & Francis, v. 27, n. 1, p. 2–23, 1996.

KEMERER, C. F.; PAULK, M. C. The impact of design and code reviews on software quality: An empirical study based on psp data. **IEEE Transactions on Software Engineering**, v. 35, n. 4, p. 534–550, July 2009. ISSN 0098-5589.

KOLLANUS, S.; KOSKINEN, J. Survey of software inspection research. **The Open Software Engineering Journal**, v. 3, n. 1, p. 15–34, 2009.

KONONENKO, O.; BAYSAL, O.; GODFREY, M. W. Code review quality: How developers see it. In: **ICSE '16**. [S.l.]: ACM, 2016. p. 1028–1038. ISBN 9781450339001.

KOREN, Y. Factorization meets the neighborhood: a multifaceted collaborative filtering model. In: ACM. **Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining**. [S.l.], 2008. p. 426–434.

KOVALENKO, V. et al. Does reviewer recommendation help developers? **IEEE Transactions on Software Engineering**, IEEE, 2018.

LI, Z.-X. et al. What are they talking about? analyzing code reviews in pull-based development model. **Journal of Computer Science and Technology**, v. 32, n. 6, p. 1060–1075, Nov 2017. ISSN 1860-4749.

MacLeod, L. et al. Code reviewing in the trenches: Challenges and best practices. **IEEE Software**, v. 35, n. 4, p. 34–42, July 2018. ISSN 1937-4194.

MÄNTYLÄ, M. V.; LASSENIUS, C. What types of defects are really discovered in code reviews? **IEEE Transactions on Software Engineering**, IEEE, v. 35, n. 3, p. 430–448, 2009.

MARTIN, J.; TSAI, W.-T. N-fold inspection: A requirements analysis technique. **Communications of the ACM**, ACM, v. 33, n. 2, p. 225–232, 1990.

MCINTOSH, S. et al. The impact of code review coverage and code review participation on software quality: A case study of the qt, vtk, and itk projects. In: **Proceedings of the 11th Working Conference on Mining Software Repositories**. New York, NY, USA: ACM, 2014. (MSR 2014), p. 192–201. ISBN 978-1-4503-2863-0. Available from Internet: <http://doi.acm.org/10.1145/2597073.2597076>.

MEYER, B. Design and code reviews in the age of the internet. **Commun. ACM**, ACM, New York, NY, USA, v. 51, n. 9, p. 66–71, sep. 2008. ISSN 0001-0782. Available from Internet: <http://doi.acm.org/10.1145/1378727.1378744>.

MILLER, J.; WOOD, M.; ROPER, M. Further experiences with scenarios and checklists. **Empirical Software Engineering**, Springer, v. 3, n. 1, p. 37–64, 1998.

MIRSAEEDI, E.; RIGBY, P. Mitigating turnover with code review recommendation: Balancing expertise, workload, and knowledge distribution. In: **ICSE'20**. New York, NY, USA: ACM, 2020.

OLSON, G. M.; OLSON, J. S. Distance matters. **Human-computer interaction**, L. Erlbaum Associates Inc., v. 15, n. 2, p. 139–178, 2000.

OLSON, J. S.; OLSON, G. M. Working together apart: Collaboration over the internet. **Synthesis Lectures on Human-Centered Informatics**, Morgan & Claypool Publishers, v. 6, n. 5, p. 1–151, 2013.

OUNI, A.; KULA, R. G.; INOUE, K. Search-based peer reviewers recommendation in modern code review. In: IEEE. **2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)**. [S.l.], 2016. p. 367–377.

PANICHELLA, S. et al. Would static analysis tools help developers with code reviews? In: IEEE. **Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on**. [S.l.], 2015. p. 161–170.

PARNAS, D. L.; WEISS, D. M. Active design reviews: principles and practices. In: IEEE COMPUTER SOCIETY PRESS. **Proceedings of the 8th international conference on Software engineering**. [S.l.], 1985. p. 132–136.

PEDREGOSA, F. et al. Scikit-learn: Machine learning in Python. **Journal of Machine Learning Research**, v. 12, p. 2825–2830, 2011.

QUINLAN, J. R. Induction of decision trees. **Machine learning**, Springer, v. 1, n. 1, p. 81–106, 1986.

RADERMACHER, A. D.; WALIA, G. S. Investigating the effective implementation of pair programming: An empirical investigation. In: **Proceedings of the 42Nd ACM Technical Symposium on Computer Science Education**. New York, NY, USA: ACM, 2011. (SIGCSE '11), p. 655–660. ISBN 978-1-4503-0500-6. Available from Internet: <http://doi.acm.org/10.1145/1953163.1953346>.

RAHMAN, M. M. et al. Correct: Code reviewer recommendation at github for vendasta technologies. In: IEEE. **Automated Software Engineering (ASE), 2016 31st IEEE/ACM International Conference on**. [S.l.], 2016. p. 792–797.

RAHMAN, M. M. et al. Correct: Code reviewer recommendation at GitHub for Vendasta technologies. In: **ASE 2016**. [S.l.]: ACM, 2016. p. 792–797. ISBN 9781450338455.

RIGBY, P. et al. Contemporary peer review in action: Lessons from open source development. **IEEE Software**, IEEE Press, Washington, DC, USA, v. 29, n. 6, p. 56–61, nov. 2012. ISSN 0740-7459.

RIGBY, P. C.; BIRD, C. Convergent software peer review practices. 2013.

ROBNIK-SIKONJA, M.; KONONENKO, I. An adaptation of relief for attribute estimation in regression. In: **Machine Learning: Proceedings of the Fourteenth International Conference (ICML'97)**. [S.l.: s.n.], 1997. v. 5, p. 296–304.

ROKACH, L.; MAIMON, O. Top-down induction of decision trees classifiers-a survey. **IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)**, IEEE, v. 35, n. 4, p. 476–487, 2005.

SABALIAUSKAITE, G.; KUSUMOTO, S.; INOUE, K. Assessing defect detection performance of interacting teams in object-oriented design inspection. **Information and Software Technology**, Elsevier, v. 46, n. 13, p. 875–886, 2004.

SADOWSKI, C. et al. Modern code review: A case study at Google. In: **ICSE-SEIP '18**. [S.l.]: ACM, 2018. p. 181–190. ISBN 9781450356596.

SAITO, T.; REHMSMEIER, M. The precision-recall plot is more informative than the roc plot when evaluating binary classifiers on imbalanced datasets. **PloS one**, Public Library of Science, v. 10, n. 3, p. e0118432, 2015.

SANTOS, E. W. d.; NUNES, I. Investigating the effectiveness of peer code review in distributed software development. In: **SBES'17**. [S.l.]: ACM, 2017. p. 84–93. ISBN 978-1-4503-5326-7.

SANTOS, E. W. d.; NUNES, I. Investigating the effectiveness of peer code review in distributed software development based on objective and subjective data. **Journal of Software Engineering Research and Development**, Springer, v. 6, p. 14:1–14:31, 2018. ISSN 2195-1721.

SANTOS, E. W. d.; NUNES, I.; JANNACH, D. Team-related features in code review prediction models. **Empirical Software Engineering**, Submitted, 2021.

SCHENK, J.; PRECHELT, L.; SALINGER, S. Distributed-pair programming can work well and is not just distributed pair-programming. In: **Companion Proceedings of the 36th International Conference on Software Engineering**. New York, NY, USA: ACM, 2014. (ICSE Companion 2014), p. 74–83. ISBN 978-1-4503-2768-8. Available from Internet: <http://doi.acm.org/10.1145/2591062.2591188>.

SHIMAGAKI, J. et al. A study of the quality-impacting practices of modern code review at sony mobile. In: **Proceedings of the 38th International Conference on Software Engineering Companion**. New York, NY, USA: ACM, 2016. (ICSE '16), p. 212–221. ISBN 978-1-4503-4205-6. Available from Internet: <http://doi.acm.org/10.1145/2889160.2889243>.

THONGTANUNAM, P. et al. Improving code review effectiveness through reviewer recommendations. In: ACM. **Proceedings of the 7th International Workshop on Cooperative and Human Aspects of Software Engineering**. [S.l.], 2014. p. 119–122.

THONGTANUNAM, P. et al. Investigating code review practices in defective files: An empirical study of the qt system. In: **Proceedings of the 12th Working Conference on Mining Software Repositories**. Piscataway, NJ, USA: IEEE Press, 2015. (MSR

'15), p. 168–179. Available from Internet: <http://dl.acm.org/citation.cfm?id=2820518.2820540>.

THONGTANUNAM, P. et al. Review participation in modern code review. **Empirical Software Engineering**, Springer, p. 1–50, 2016.

THONGTANUNAM, P. et al. Revisiting code ownership and its relationship with software quality in the scope of modern code review. In: **Proceedings of the 38th International Conference on Software Engineering**. New York, NY, USA: ACM, 2016. (ICSE '16), p. 1039–1050. ISBN 978-1-4503-3900-1. Available from Internet: <http://doi.acm.org/10.1145/2884781.2884852>.

THONGTANUNAM, P. et al. Who should review my code? a file location-based code-reviewer recommendation approach for modern code review. In: IEEE. **Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on**. [S.l.], 2015. p. 141–150.

WANG, H.; YIN, G.; LING, C. Who should review this pull-request: Reviewer recommendation to expedite crowd collaboration. In: **Proceedings of IEEE APSEC**. [S.l.: s.n.], 2014.

XIA, X. et al. Who should review this change?: Putting text and file location analyses together for more accurate recommendations. In: IEEE. **Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on**. [S.l.], 2015. p. 261–270.

XIA, Z. et al. A hybrid approach to code reviewer recommendation with collaborative filtering. In: IEEE. **2017 6th International Workshop on Software Mining (SoftwareMining)**. [S.l.], 2017. p. 24–31.

YANG, X. Social network analysis in open source software peer review. In: ACM. **Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering**. [S.l.], 2014. p. 820–822.

YING, H. et al. Earec: leveraging expertise and authority for pull-request reviewer recommendation in github. In: ACM. **Proceedings of the 3rd International Workshop on CrowdSourcing in Software Engineering**. [S.l.], 2016. p. 29–35.

YOURDON, E. **Structured walkthroughs**. [S.l.]: Prentice Hall PTR, 1979.

ZANJANI, M. B.; KAGDI, H.; BIRD, C. Automatically recommending peer reviewers in modern code review. **IEEE Transactions on Software Engineering**, IEEE, v. 42, n. 6, p. 530–543, 2016.

APPENDIX A — SURVEY QUESTIONNAIRES

This appendix contains the two questionnaires used in the study presented in Chapter 5. In Section A.1, we provide the transcription of the questions that are part of the main questionnaire, as well as the introductory text with the research context, terminology, and participation agreement. Similarly, Section A.2 contains the questions and introductory text of the follow-up questionnaire, which was used to complement the main questionnaire.

A.1 Main Questionnaire

Introduction

Thank you for your interest in participating in our research.

This research is part of a study that aims to understand which factors are relevant for a code review to be more effective. In Modern Code Review, authors submit the changes for review by their peers (i.e. reviewers), without the need for meetings or synchronous participation of the involved people, because this activity is in general supported by specific tools for code annotation and feedback provision, with interaction done in an asynchronous way. The reviewers are invited to contribute, offering feedback in the form of comments, votes or questions, although some abstain themselves or do not provide input in the process. The changes made in the code can be in various sizes in terms of lines of code.

The research is being conducted by Eduardo Witter dos Santos and Prof. Ingrid Nunes of the Instituto de Informática, Universidade Federal do Rio Grande do Sul (UFRGS), Porto Alegre, Brasil.

Participation agreement

- PARTICIPATION: You will be asked to answer a set of questions about your experience and perceptions with respect to the process of code review in software development. We estimate that it takes 15-20 minutes of your time.
- VOLUNTARY PARTICIPATION: You have the right to withdraw from this study at any moment. To withdraw, leave this website before you conclude the survey.

- Risks: There are no risks associated with the participation in this study.
- CONFIDENTIALITY and ANONYMITY: The web application does not collect any data that allow us to personally identify you. Only the researchers listed about will have normal access to the underlying data. Any other access to this information, in the form of a publication, for example, will have all your personal data removed. Authorities of countries where Google operates, including Brazil, can possibly look for access to the data by legal processes. However, it is unlikely.
- DATA CONSERVATION: The data will be stored after the project conclusion for at least 5 years.

If you have any question about this study, please do not hesitate to contact the researchers via e-mail: ewitter@gmail.com

Indicate if you agree or not in participating in this research.

☐ I agree to participate.

☐ I don't agree and, therefore, prefer not to participate.

Terminology

Throughout this questionnaire, you will have to answer a set of questions. You will find some terms, which are described below. Please, read the definitions to guarantee that the understanding of these terms are what we expect.

- Authors: who creates or modifies artefacts to be reviewed.
- Reviewers: who review the work made by authors, providing feedback and helping in various forms, such as comments, votes, suggestions of alternative approaches.
- Participation: the authors can invite a set of reviewers, from which only a fraction in fact interact—these are the active reviewers. Reviews where all invited reviewers participate have 100% of participation.
- Team: collaborations that work under the same immediate supervision, in the same project.
- Site: a location. We consider to be distinct locations: another floor, another building, another city, or another country.

Participant data

1. Age: _____
2. Gender

   ☐ Male ☐ Other
   ☐ Female ☐ I prefer not to inform

3. Which is your level of education? (*in Computer Science or similar programs)

   ☐ Primary/Middle school ☐ Incomplete master degree*
   ☐ High school ☐ Master degree*
   ☐ Incomplete undergraduate degree* ☐ Incomplete doctorate degree*
   ☐ Undergraduate degree* ☐ Doctorate degree*

4. How many years of experience in software development do you have?

   ☐ Less than 2 years ☐ More than 10 years
   ☐ Between 2 and 5 years ☐ I don't have professional experience
   ☐ Between 5 and 10 years

5. Evaluate your experience in the topics below

	Very low	Low	Average	High	Very high
Software development	☐	☐	☐	☐	☐
Code review, as reviewer	☐	☐	☐	☐	☐
Code review, as author	☐	☐	☐	☐	☐
Projects with multiple teams	☐	☐	☐	☐	☐
Projects with multiple sites	☐	☐	☐	☐	☐

About Code Review

6. Evaluate the contribution of code review to each of the items listed below

	Largely negative	Negative	Neutral	Positive	Largely positive	N/A
Participant Learning	☐	☐	☐	☐	☐	☐
Code maintainability	☐	☐	☐	☐	☐	☐
Product quality	☐	☐	☐	☐	☐	☐
Delivery time	☐	☐	☐	☐	☐	☐
Development cost	☐	☐	☐	☐	☐	☐

Code review contributes to something not listed above?
If yes, list the missing item(s) and evaluate its contribution below.

_____

_____

7. Evaluate each of the characteristics related to code review below.

	The lower, the better	The closest to a value/interval, the better	The more, the better	N/A
Review duration (in hours, days, etc.)	☐	☐	☐	☐
Participation (%) of invited reviewers	☐	☐	☐	☐
Delivery time	☐	☐	☐	☐
Number of participating reviewers	☐	☐	☐	☐
Number of reviewer comments (by means of suggestions, requests, feedback, questions, votes, etc.)	☐	☐	☐	☐

Considering the items above, for each characteristic you selected "The closest to a value/interval, the better", indicate which is this value/interval.

_____

_____

Personal Experience in Code Review

In each question in this section, consider your broad PERSONAL experience, not only in the project or company where you currently work.

8. About the interaction between reviewers and authors: Based on the reviews in which I participated, I noticed that...

	Disagree	Partially disagree	Neutral	Partially agree	Agree	N/A
... some people are more willing and like more to contribute as reviewers than others.	☐	☐	☐	☐	☐	☐
... reviewers feel less comfortable to rigorous review if they do not personally know the authors.	☐	☐	☐	☐	☐	☐
... reviews of very large artefacts tend to be avoided or postponed by reviewers.	☐	☐	☐	☐	☐	☐
... reviews of very large artefacts tend to be done with less rigour.	☐	☐	☐	☐	☐	☐
... there are reviewers that generally provide more relevant feedback and better contribution to the process than others.	☐	☐	☐	☐	☐	☐

9. With respect to tracking during code review: Based on the reviews in which I participated, I noticed that...

	Disagree	Partially disagree	Neutral	Partially agree	Agree	N/A
... some reviewers tend to contribute less if the code review is not properly tracked as a performed activity.	☐	☐	☐	☐	☐	☐
... it is frequent to provide informal feedback related to the reviews, storing incomplete records in the code review tool.	☐	☐	☐	☐	☐	☐
... it is frequent to perform a review by e-mail, chat or other electronic communication means, without storing records in the code review tool.	☐	☐	☐	☐	☐	☐

10. With respect to agility in code review: Based on the reviews in which I participated, I noticed that...

	Disagree	Partially disagree	Neutral	Partially agree	Agree	N/A
... interventions are sometimes needed for the reviewers of other SITES to perform a pending review.	☐	☐	☐	☐	☐	☐
... interventions are sometimes needed for the reviewers of other TEAMS to perform a pending review.	☐	☐	☐	☐	☐	☐
... periodic meetings and interaction among the team members are important for the reviews to be done in adequate time.	☐	☐	☐	☐	☐	☐

11. About the appreciation of reviewers and the code review as a development activity

	Disagree	Partially disagree	Neutral	Partially agree	Agree	N/A
MY managers notice and value when I contribute with code reviews.	☐	☐	☐	☐	☐	☐
Managers of other TEAMS notice and value when I contribute with code reviews.	☐	☐	☐	☐	☐	☐
Managers of teams from other SITES notice and value when I contribute with code reviews.	☐	☐	☐	☐	☐	☐
Developers from MY team notice and value when I contribute with code reviews.	☐	☐	☐	☐	☐	☐
Developers from other TEAMS notice and value when I contribute with code reviews.	☐	☐	☐	☐	☐	☐
Developers from other SITES notice and value when I contribute with code reviews.	☐	☐	☐	☐	☐	☐

Factors that Influence in the Code Review

In each question in this section, tell us about your perception regarding the influence of the three factors below about a specific aspect of code reviews.

12. Influence on review duration (in hours, days, etc.)

	much slower is the review.	slower is the review.	the duration is not affected.	faster is the review.	much faster is the review.	N/A
The higher the number of lines of code to be reviewed...	☐	☐	☐	☐	☐	☐
The higher the number of teams involved in the review...	☐	☐	☐	☐	☐	☐
The higher the number of sites involved in the review...	☐	☐	☐	☐	☐	☐
The higher the number of reviewers effectively participating in the review...	☐	☐	☐	☐	☐	☐

In case the relationship between the duration of the review and some of the listed items are not directly or inversely proportional, mark the option "N/A" and explain your choice below.

_____

13. Influence on the motivation of peers to be reviewers

	much lower is the participation.	lower is the participation.	the participation is not affected.	higher is the participation.	much higher is the participation.	N/A
The higher the number of lines of code to be reviewed...	☐	☐	☐	☐	☐	☐
The higher the number of teams involved in the review...	☐	☐	☐	☐	☐	☐
The higher the number of sites involved in the review...	☐	☐	☐	☐	☐	☐
The higher the number of reviewers effectively participating in the review...	☐	☐	☐	☐	☐	☐

In case the relationship between the motivation of collaborators to be reviewers and some of the listed items are not directly or inversely proportional, mark the option "N/A" and explain your choice below.

_____

14. Influence on the number of review comments (with suggestions, requests, feedback, questions, votes, etc.) generating more interaction among reviewers and authors

	much lower is the number of comments.	lower is the number of comments.	the number of comments is not affected.	higher is the number of comments.	much higher is the number of comments.	N/A
The higher the number of lines of code to be reviewed...	☐	☐	☐	☐	☐	☐
The higher the number of teams involved in the review...	☐	☐	☐	☐	☐	☐
The higher the number of sites involved in the review...	☐	☐	☐	☐	☐	☐
The higher the number of reviewers effectively participating in the review...	☐	☐	☐	☐	☐	☐

In case the relationship between the number of reviewer comments and some of the listed items are not directly or inversely proportional, mark the option "N/A" and explain your choice below.

_____

Additional Comments

15. Are there any additional comments you would like to make?

_____

_____

16. If you would like to receive the results of this survey, please, provide your e-mail below.

_____

A.2 Follow-up Questionnaire

This questionnaire contains questions that have as objective to complement our survey on Code Review that you previously participated in. This research is being conducted by Eduardo Witter dos Santos and Prof. Ingrid Nunes of the Instituto de Informática, Universidade Federal do Rio Grande do Sul (UFRGS), Porto Alegre, Brasil. Thank you for your continued interest to participate in our research.

1. The majority of the participants in the previous questionnaire reported the perception that the fact of not having reviews tracked in an explicit way as an activity performed in the context of a project does not demotivate developers to perform reviews. Based on this, in your opinion, which are the main motivations for developers to contribute reviewing when the activity is not tracked?

   ☐ Learning with the reviews
   ☐ Benefits to the career/salary
   ☐ Perceptions of collective code ownership
   ☐ Retribution (reviewers are also producers that demand reviews by other)
   ☐ Personal satisfaction by performing the task
   ☐ Being acknowledged as a technical reference
   ☐ Being acknowledged for the contribution to the team
   ☐ Other (please specify):_____

2. In the previous questionnaire, we asked participants to evaluate "the contribution of code review to delivery time". How have you understood "delivery time" in this context?

   ☐ I understood it as being the duration of the task being done (development and its review), in the short term
   ☐ I understood it as being the duration of the development of the software release, in the medium term
   ☐ I understood it as not only referring to the time to do the specific task/release, but also to the impact that changes have in the future software evolution (future tasks/releases), in the medium-long term
   ☐ Other (please specify):_____

3. In the previous questionnaire, we asked for participants to evaluate "the contribution of code review to the development costs". How have you understood the term "development costs" in this context?

   ☐ I understood it as being the cost of the task, in the short term
   ☐ I understood it as being the cost of the development of the software release, in the medium term
   ☐ I understood it as not only referring to the cost to do the specific task/release, but also the impact that changes have in the future software evolution (future tasks/releases), in the medium-long term

☐ Other (please specify):_____

4. Which was your understanding of the relationship between development time and cost in the two questions that were made in the previous questionnaire?

_____

5. Which means of communication is most used by reviewers for them to give feedback in code reviews in the following situations?

	Documented, in the code review tool	Documented, using chat/email	Documented, in meeting minutes	Not documented, in team meetings	Not documented, in dialogs in person or by phone	Others
Questions about the code being reviewed	☐	☐	☐	☐	☐	☐
Requests to add of comments/code documentation	☐	☐	☐	☐	☐	☐
Indication of possible architecture violation	☐	☐	☐	☐	☐	☐
Indication of problems in automated tests	☐	☐	☐	☐	☐	☐
Indications of possible bugs in the reviewed code	☐	☐	☐	☐	☐	☐
Discussion about the impact of the reviewed code in the software architecture	☐	☐	☐	☐	☐	☐
Suggestions of improvements in the code being reviewed (for maintainability/legibility)	☐	☐	☐	☐	☐	☐

If you selected "Others" for some of the previous items, please detail which would be the means of communication:

_____

In the above situations, does the severity of the raised questions in the review impact the choice of the means of communication? If yes, how?

_____

In the above situations, does the amount of the raised questions in the review impact the choice of the means of communication? If yes, how?

_____

6. Additional Comments:

_____

_____

APPENDIX B — TEAM STRUCTURE DATA FORMAT

This appendix contains a short, anonymized example of how team structure is represented in this dissertation. There is a list of work records for each developer with start date, end date, and location working for a specific team, represented by the developer's immediate manager. Whenever a developer changes its location or team, a new entry is created.

```
{
    "developer1": [
        {
            "start": "2014-01-10",
            "end": "2016-08-03",
            "location": "city3",
            "team": "manager1"
        },
        {
            "start": "2016-08-04",
            "end": "2017-02-07",
            "location": "city3",
            "team": "manager2"
        }
    ],
    "developer2": [
        {
            "start": "2014-01-10",
            "end": "2018-08-31",
            "location": "city2",
            "team": "manager3"
        },
        {
            "start": "2018-09-01",
            "end": "2019-07-02",
            "location": "city2",
            "team": "manager4"
        }
    ],
    "developer3": [
        {
            "start": "2014-01-10",
            "end": "2016-08-03",
            "location": "city3",
            "team": "manager5"
        },
        {
            "start": "2016-08-04",
            "end": "2018-10-29",
            "location": "city3",
            "team": "manager6"
        }
    ],
    "developer4": [
        {
            "start": "2014-01-10",
            "end": "2016-08-03",
            "location": "city1",
            "team": "manager7"
        },
        {
            "start": "2016-08-04",
            "end": "2017-02-07",
            "location": "city1",
            "team": "manager8"
        },
        {
            "start": "2017-02-08",
            "end": "2019-04-01",
            "location": "city1",
            "team": "manager9"
        }
    ],
    "developer5": [
        {
            "start": "2014-01-10",
            "end": "2016-08-03",
            "location": "city2",
            "team": "manager10"
        },
        {
            "start": "2016-08-04",
            "end": "2019-07-02",
            "location": "city1",
            "team": "manager11"
        }
    ],
    "developer6": [
        {
            "start": "2014-01-10",
            "end": "2016-07-01",
            "location": "city1",
            "team": "manager11"
        }
    ],
    "developer7": [
        {
            "start": "2014-10-01",
            "end": "2019-07-02",
            "location": "city1",
            "team": "manager12"
        }
    ]
}
```