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

NICOLE DA COSTA DAVILA

Modern Code Review: From Foundational Studies to Proposed Approaches and their Evaluation

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

Advisor: Prof. PhD. Daltro Nunes
Coadvisor: Prof. PhD. Ingrid Nunes

Porto Alegre
February 2020

CIP — CATALOGING-IN-PUBLICATION

Davila, Nicole da Costa

Modern Code Review: From Foundational Studies to Proposed Approaches and their Evaluation / Nicole da Costa Davila. – Porto Alegre: PPGC da UFRGS, 2020.

86 f.: il.

Thesis (Master) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2020. Advisor: Daltro Nunes; Coadvisor: Ingrid Nunes.

1. Modern code review, software inspection, software verification, software quality, systematic literature review. I. Nunes, Daltro. II. Nunes, Ingrid. III. Título.

AGRADECIMENTOS

First, I would like to thank my family. Thanks to my parents, Fatima and Jair, for understanding those moments when I couldn't be around, for all the support and love over the years. To Felipe, I thank you for your patience and support, for listening when I needed to organize my ideas (or just talk). I am especially grateful to my advisors, Daltro and Ingrid Nunes. I dedicate a special thanks to Ingrid for her guidance in the development of this thesis. Together with other incredible women, Ingrid is a reference to the researcher I want to be. Thanks to the members of the examining committee for taking the time to read my work and providing valuable feedback. My sincere gratitude to my friends, who also understood those times when I wasn't able to be around. To everyone who helped me develop this research in any way, thank you all.

ABSTRACT

Modern Code Review (MCR) has gained increasing popularity both in academia and industry as a static verification technique that can promote improved product and code quality as well as knowledge sharing and learning. This practice has been target of a large amount of research, including exploratory studies and proposals to support it. However, the body of knowledge of MCR already built is currently not understood as a whole. We thus present a systematic literature review of research work that has been done in this context. Our systematic procedure to select existing work led us to a total of 110 publications. These are analyzed in three main categories that are associated with our research questions. FOUNDATIONAL STUDIES are those that analyze quantitative or qualitative data to extract lessons learned from the adoption of MCR. PROPOSALS consist of techniques and tools proposed to support the MCR process, while EVALUATIONS are studies to assess an individual proposal or compare a set of them. From the analysis of reviewed literature, we identified that most part of the existing studies of MCR consists of FOUNDATIONAL STUDIES that have been conducted to better understand the adoption of the practice and the analysis of which influence factors lead to which MCR outcomes. From the novel approaches to support MCR, the most common is code reviewer recommenders. EVALUATIONS of MCR approaches have been done mostly offline and few studies involving human subjects have been conducted. We describe investigated studies in terms of their key characteristics and contrast their findings. In addition to introducing the state of the art of MCR, we provide insights derived from our review, which point out directions of future work in the area.

Keywords: Modern code review, software inspection, software verification, software quality, systematic literature review.

Revisão de Código Moderna:
Dos Estudos Fundamentais às Abordagens Propostas e sua Avaliação

ABSTRACT

A revisão de código moderna (MCR) tem se popularizado como uma técnica de verificação estática que pode promover tanto a melhoria da qualidade do software e do código, como o compartilhamento de conhecimento e o aprendizado. Diversas pesquisas tem endereçado a prática, incluindo estudos exploratórios e propostas de abordagem para apoiá-la. Entretanto, o corpo de conhecimento já construído sobre MCR não é totalmente compreendido atualmente. Assim, apresentamos uma revisão sistemática da literatura sobre as pesquisas realizadas nesse contexto. Adotando um procedimento sistemático, a seleção dos trabalhos existentes resultou em 110 artigos científicos, os quais são analisados em três principais categorias. ESTUDOS FUNDAMENTAIS são estudos que examinam dados quantitativos ou qualitativos para extrair as lições aprendidas com a adoção do MCR. PROPOSTAS consistem em técnicas e ferramentas propostas para apoiar o processo MCR, enquanto AVALIAÇÕES são estudos para avaliar uma proposta individual ou comparar um conjunto delas. A partir da análise da literatura revisada, identificamos que a maior parte dos estudos sobre MCR é formado por ESTUDOS FUNDAMENTAIS que foram realizados para entender melhor a adoção da prática e para a análise de quais fatores de influência levam a que resultados do MCR. Das novas abordagens para oferecer suporte ao MCR, o tipo mais comum são os recomendadores de revisores de código. As avaliações das abordagens de MCR foram feitas principalmente offline e poucos estudos envolvendo seres humanos foram realizados. Descrevemos os estudos investigados em termos de suas principais características e contrastamos com seus resultados. Além de apresentar o estado da arte do MCR, fornecemos informações derivadas de nossa revisão, que apontam as direções de trabalhos futuros na área.

Keywords: revisão de código moderna, inspeção de software, verificação de software, qualidade de software, revisão sistemática da literatura.

LIST OF ABBREVIATIONS AND ACRONYMS

CFA CodeFlow Analytics

MCR Modern Code Review

OSS Open Source Software

SLR Systematic Literature Review

LIST OF FIGURES

Figure 2.1 An overview of the tasks in modern code review. 16

Figure 3.1 Facets analyzed in each primary study. 22
Figure 3.2 Overview of MCR research based on primary studies of our SLR. 24

Figure 4.1 Studies that analyzed the relationship between influence factors and outcomes. 36

Figure 5.1 Number of papers reporting an evaluation per MCR approach type. 59
Figure 5.2 Number of studies categorized by evaluation type and MCR approach type. 60

LIST OF TABLES

Table 3.1 Search results by source. 19
Table 3.2 Inclusion criteria (IC), exclusion criteria (EC) and the number of papers satisfying them. 20
Table 3.3 Selected primary studies 23

Table 4.1 Taxonomy of topics researched in groups of FOUNDATIONAL STUDIES. 26
Table 4.2 Main expected benefits of the use of MCR. 29
Table 4.3 Influence factors analyzed by the FOUNDATIONAL STUDIES 34
Table 4.4 Internal outcomes as influence factors analyzed by the FOUNDATIONAL STUDIES. 35

Table 5.1 Classification of MCR PROPOSALS. 51
Table 5.2 Summary of the proposed approaches of tool support for MCR. 57
Table 5.3 Classification of MCR EVALUATIONS 59
Table 5.4 Descriptive Statistics of the Target of Offline Evaluations and Experiments. 62
Table 5.5 Results of comparisons of code reviewer recommenders. 64
Table 5.6 Results of comparisons between a reviewer recomender and a baseline technique. 64

Table 6.1 Influence on the MCR practice 69

CONTENTS

1 INTRODUCTION 11
2 BACKGROUND ON CODE REVIEW 14
3 SYSTEMATIC LITERATURE REVIEW 18
3.1 Systematic Review Protocol 18
3.1.1 Research Questions 18
3.1.2 Search Strategy 19
3.1.3 Selection Criteria 20
3.2 Review Execution 21
3.2.1 Search Execution 21
3.2.2 Selection of Primary Studies 21
3.2.3 Analysis Method 22
4 FOUNDATIONAL RESEARCH ON MCR 25
4.1 Overview 25
4.2 Practitioner Perception of the Practice 26
4.2.1 Code Review Process 27
4.2.2 Social Dynamics 28
4.2.3 Expected Benefits 29
4.2.4 Challenges and Difficulties 30
4.3 Experience Report 31
4.4 Analysis of MCR Outcomes 32
4.4.1 Internal Outcomes 33
4.4.1.1 Reviewer Team 36
4.4.1.2 Review Feedback 38
4.4.1.3 Review Intensity 40
4.4.1.4 Review Time 41
4.4.1.5 Review Decision 43
4.4.2 External Outcomes 43
4.5 Human Aspects of Reviewers 46
4.6 Relationship with MCR 47
4.6.1 Comparison with Verification Techniques 47
4.6.2 Interaction with Development Practices 48
5 PROPOSALS AND THEIR EVALUATION 50
5.1 Proposals to Support MCR 50
5.1.1 Review Planning and Setup 50
5.1.1.1 Patch Documentation 50
5.1.1.2 Reviewer Recommender 51
5.1.1.3 Review Prioritization 52
5.1.2 Code Review 52
5.1.2.1 Code Checking 53
5.1.2.2 Feedback Provision 54
5.1.2.3 Review Decision 54
5.1.3 Process Support 55
5.1.3.1 Grounded Theory 55
5.1.3.2 Review Retrospective 56
5.1.3.3 Tool Support 57
5.2 Evaluations of MCR Approaches 58
5.2.1 Evaluations Types 58

5.2.2 Offline Evaluations 61
5.2.2.1 Study Design 61
5.2.2.2 Findings 63
5.2.3 Experiments 65
5.2.3.1 Study Design 65
5.2.3.2 Findings 66
6 DISCUSSION 67
6.1 Foundational Studies of MCR 67
6.1.1 MCR Dynamics and Perceived Benefits 67
6.1.2 Factors that Influence the MCR Practice 68
6.2 Developed Approaches and their Evaluations 68
7 CONCLUSION AND FUTURE WORK 70
REFERENCES 72
APPENDIX A — RESUMO ESTENDIDO 84

1 INTRODUCTION

Code review is a widely known practice of quality assurance in software development, consisting of the manual checking of changes in the source code. It involves a review of code changes that are performed by developers other than the author. The goal is to look for defects or improvement opportunities without the software execution and before the product delivery, thus reducing costs of fixing them later (BAUM et al., 2016a).

In its early stages, reviews were performed in a more structured and rigid form, focusing on different software artifacts, being referred to as *software inspection*. Fagan's method (FAGAN, 1976) made this practice popular by proposing a formal method of finding defects, structured by roles and phases and including an inspection meeting. Software inspection has been, for years, considered an effective means of finding defects in artifacts, such as design, code, and test cases, as reported in existing surveys of work on software inspection (AURUM; PETERSSON; WOHLIN, 2002; MACDONALD et al., 1995; MACDONALD; MILLER, 1999; BRYKCZYNSKI, 1999; LAITENBERGER; DEBAUD, 2000; HERNANDES; BELGAMO; FABBRI, 2014; KOLLANUS; KOSKINEN, 2007). These surveys discussed many studies that provide evidence of the effectiveness of software inspection. Nevertheless, inspection meetings are synchronous and involve many roles, thus can be cumbersome and time-consuming.

Software inspection has been pointed out as a mature and well-studied practice in software engineering (SHULL; SEAMAN, 2008). However, the evolution in the way how software development is done led to changes in code review. The adoption of agile methods and distributed development created a scenario in which there is a need for a less formal review, reducing the inefficiencies of inspections through a flexible, tool-based and asynchronous process, namely *modern code review* (MCR) (BACCHELLI; BIRD, 2013). In this lightweight variant of the method, there is no meeting. The focus is on small code changes, and reviews happen early, quickly, and frequently, which helps detect defects and problem-solving, among other benefits (RIGBY; BIRD, 2013). Moreover, because MCR is supported by tools, it is possible to track activities and improve review-related tasks, such as reviewer recommendation (THONGTANUNAM et al., 2015).

Several empirical studies have demonstrated the increasing adoption of MCR, both in the Open Source Software (OSS) community and in large companies, such as Google (SADOWSKI et al., 2018) and Microsoft (BACCHELLI; BIRD, 2013; BOSU;

GREILER; BIRD, 2015). Therefore, while software inspection has been largely investigated in the past, the fundamental differences between MCR and software inspection call for research to understand the MCR benefits, challenges and how it can be improved. This, together with the current popularity of MCR, motivated many studies on this practice. Such studies include interviews with developers to understand the state of practice (BAUM; LESSMANN; SCHNEIDER, 2017), repository mining (SHIMAGAKI et al., 2016), and proposals to support the MCR process (ZHANG et al., 2011).

In order to aggregate generated knowledge, existing work aimed to classify the MCR process adopted in the industry (BAUM et al., 2016a) based on interviews and a semi-systematic literature review, and to map the existing literature on MCR (BADAMPUDI; BRITTO; UNTERKALMSTEINER, 2019; FRONZA et al., 2020) by means of systematic mapping studies. Badampudi, Britto and Unterkalmsteiner (2019) focused on what and how aspects of the practice have been investigated to observe the evolution of research related to this topic. Similarly, Fronza et al. (2020) aimed to identify which themes are covered in the literature of code review, software inspections, and code walkthroughs, focusing on empirical studies. Even though these studies took a step towards the understanding of research that has been done on MCR, existing work has not been investigated in-depth, leading to the need for constructing a structured body of knowledge.

In this work, we survey the literature to provide an in-depth understanding of research that has been carried out on MCR. Given the many foundational studies that have been conducted to collect and analyze data obtained from developers and MCR repositories, our work classifies and contrasts their results. We also investigate proposals that have been done in this context, by understanding their goals and how they differ. Finally, we observe methodological aspects of how foundational studies have been performed and how proposals have been evaluated. From the analysis of reviewed literature, we identified that most part of the existing studies of MCR consists of foundational studies that have been conducted to better understand the adoption of the practice and the analysis of which influence factors lead to which MCR outcomes. From the novel approaches to support MCR, the most common is code reviewer recommenders. Evaluations of MCR approaches have been done mostly offline and few studies involving human subjects have been conducted.

Our literature review followed a systematic method to select and analyze research studies in order to reduce the researcher bias. As result, 110 (out of 702 retrieved) papers have been included and analyzed in our work. Our main contribution is a survey that con-

veys the current state-of-the-art research on MCR with an in-depth analysis of researched aspects in this context.

The remainder of this thesis is organized as follows. In Chapter 2, we give a background on code review, detailing the software inspection, and its transition to MCR. We then present the methodology to perform our systematic review in Chapter 3, followed by the analysis of its results that are described in Chapter 4 and Chapter 5. Finally, we discuss insights derived from our review in Chapter 6 and our conclusions in Chapter 7.

2 BACKGROUND ON CODE REVIEW

Code review is a static verification technique that consists of the manual checking of code changes before they are integrated into the main code repository. This technique has many variations with different levels of formality, such as pair programming (WILLIAMS, 2001), walkthrough (WEINBERG; FREEDMAN, 1984), and review by circulation (KASSE, 2008).

A formal variation of code review, namely *software inspections*, became popular in the 70s. Software inspections are not restricted to the review of code, but also of other software artifacts, e.g. documentation and test cases. The main motivation for introducing software inspections in the software development process was the early detection of defects, because this is less expensive than identifying and fixing defects in posterior stages of the software development. A widely-known method of software inspection was proposed by Fagan (1976), who described inspections as a well-structured process, involving a set of phases and roles, and following strict guidelines. A key aspect of Fagan's method is the need for in-person, therefore synchronous, review *meetings*, where artifacts are checked for errors. In such meetings, changes in the design, source code and test cases are inspected line by line, and anyone in this meeting can point out errors or nonconformity with standards. Feedback is recorded by a moderator, who also produces a written report with the findings, which are used to promote product and process improvements. Due to the systematic nature of the method, it is said to be *heavyweight*.

Despite the popularity of Fagan's method, other inspection methods were proposed with the goal of increasing its effectiveness and reducing the costs of the practice (AURUM; PETERSSON; WOHLIN, 2002). These methods differ from Fagan's method mainly by having different process steps. Further research work on inspections was developed in other directions. For example, there are studies that aimed to support inspections by proposing reading techniques (THELIN et al., 2004; HALLING et al., 2001), estimation of remaining defects after the review meetings (PETERSSON et al., 2004), and supporting tools (MACDONALD et al., 1995). There are also studies that focused on evaluating the effectiveness of review meetings (VOTTA JR., 1993; MILLER; YIN, 2003).

The need for synchronous in-person meetings is a key limitation of software inspection. Despite the efforts to reduce its costs and support distribution (MASHAYEKHI et al., 1993; LANUBILE; MALLARDO, 2002), the evolution of software development—

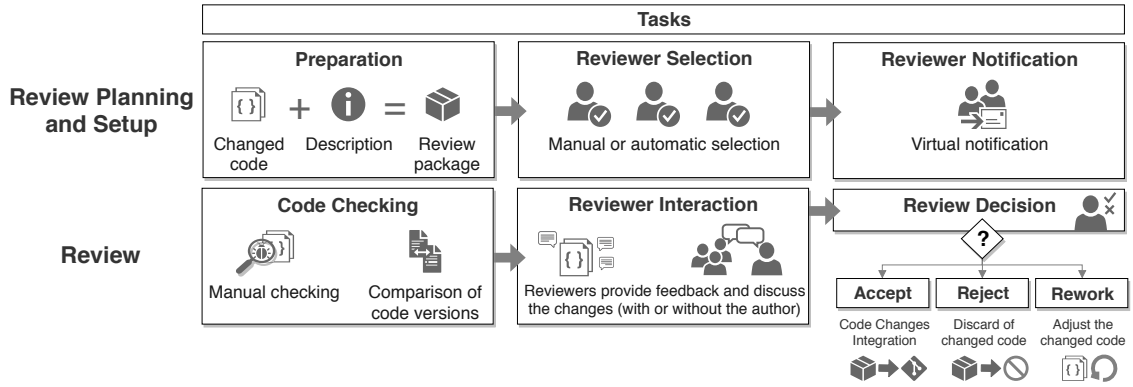
which transitioned from traditional to agile approaches, adopting continuous software delivery practices and becoming globally distributed—led to a decrease in the use of Fagan's inspection method. This scenario in software development motivated the adoption of a *lightweight* code review approach. Different terms have been used to refer to this form of code review, such as code review (MCINTOSH et al., 2014), change-based code review (BAUM; SCHNEIDER; BACCHELLI, 2017), contemporary peer code review (RIGBY; BIRD, 2013), and peer code review (BOSU; CARVER, 2012). However, most of the recent work has been using the term *modern code review* (MCR), which was first used by Cohen (2010) and became popular by the study of Bacchelli and Bird (2013).

Differently from inspections, there are no strict guidelines for adopting MCR, causing it to be a flexible practice. The overall idea is that developers other than the author, which are the reviewers, assess code changes to identify both defects and quality problems and decide whether the changes will be discarded or integrated into the principal repository of the project. MCR is mainly characterized by being asynchronous and supported by tools, occurring regularly integrated with the routine of developers (BACCHELLI; BIRD, 2013; SADOWSKI et al., 2018). Another difference from inspections, which is mainly focused on defect detection, is that it is acknowledged that MCR additionally promotes other benefits, such as knowledge transfer, increased team awareness, and the creation of alternative solutions to problems (BACCHELLI; BIRD, 2013; BOSU et al., 2017; MACLEOD et al., 2018).

In order to understand how MCR works, we present in Figure 2.1 the tasks that are typically performed to review code. However, each organization or environment can adapt how the process is adopted, customizing it with particular tools, policies, and culture. We classify these tasks into two phases, *Review Planning and Setup* and *Review*, where the former consists of tasks to enable the review to take place, while the latter is the phase when the code is assessed and decisions based on the review are made. These tasks are performed by at least two roles, namely author and reviewer. The author is a developer who has changed the source code and created a review request for it, while the reviewer (typically also developers) is responsible for reviewing the code and giving feedback. In particular settings there might be other roles, such as maintainer (SANTOS; NUNES, 2017) (a developer responsible for a system's module and is required to approve code changes in it) and commenter (JIANG et al., 2017) (who can provide comments but not make decisions).

The first phase, *Review Planning and Setup*, consists of three main tasks. In the

Figure 2.1: An overview of the tasks in modern code review.



Source: Author

first (*Preparation*), the author, who has a unit of work to be reviewed, prepares a review package composed of the changed code accompanied by description providing additional details of the change. This description typically summarizes the change in natural language. In the second task (*Reviewer Selection*), suitable reviewers that likely can or should inspect the review package are selected. Finally, in the *Reviewer Notification* tasks, reviewers receive a notification inviting them to perform the review.

In the second phase, *Review*, the process of analyzing the code for defects and improvement opportunities takes place. Reviewers thus perform *Code Checking*, assessing changes and comparing it with previous versions. Each reviewer individually checks the review package. Based on this, reviewers interact among themselves and with the author (*Reviewer Interaction*), writing feedback comments, annotating snippets of code, or promoting a discussion to clarify issues. Based on this interaction, reviewers make a decision on the review request (*Review Decision*), which can be: (i) *accept*, which leads the code to be integrated into the main code repository of the project; (ii) *reject*, so the changes are discarded; or (iii) *rework*, in which reviewers request modifications to adjust the code. The last results in a new review cycle to take place after the code is updated.

How each MCR task occurs in a concrete way depends on the selected supporting tools and customization made in particular projects. Tools can be used to, e.g., manage review requests, select reviewers, visualize and analyze code changes, register annotations, and manage the discussions. Additional tools can be used to automatically assess code changes, check for standards, collect metrics, indicate potential bugs, and so on. Feedback can be given in the form of comments or votes, and a vote can be done in different scales.

The flexibility of MCR allows it to be employed by teams and organizations in various settings using different technologies. It has been used in GitHub to review pull-

requests and in industry, assisted by proprietary software with code review features or well-known code review tools, such as Gerrit. The MCR flexibility and its tailored adoption in different contexts, and also its differences from software inspection, motivated research to identify and understand this practice as well as improve it. This led to many studies to analyze MCR aspects (RIGBY; BIRD, 2013; SADOWSKI et al., 2018), reports of its use (BOSU et al., 2017; MACLEOD et al., 2018), and supporting techniques (THONGTANUNAM et al., 2015; BALACHANDRAN, 2013). The extensive amount of studies in this context calls for a review to understand the body of knowledge that has been built, connecting findings and comparing approaches. As mentioned in the introduction, the only two efforts made in this direction (BADAMPUDI; BRITTO; UNTERKALMSTEINER, 2019; FRONZA et al., 2020) are limited to systematic mapping studies that investigated existing work in a superficial way. We, in contrast, investigate studies with an in-depth analysis, following the methodology described in the next chapter to select and examine primary papers in the context of MCR.

3 SYSTEMATIC LITERATURE REVIEW

A systematic literature review (SLR) aims to identify, evaluate and interpret all available research on a topic of interest. This type of review is characterized by a transparent specification of the study procedure to minimize research biases and support reproducibility. In this chapter, we describe the protocol of our SLR (Section 3.1), following the guidelines proposed by Kitchenham and Charters (2007). We also provide details of the execution of the specified protocol (Section 3.2) that allowed us to identify the literature investigated in our SLR.

3.1 Systematic Review Protocol

The goal of this study is to identify, analyze, and compare the state-of-the-art research on MCR. Given the increasing interest and the amount of research that has been carried out on the topic, we aim to have a collective view of the results of different studies that have been done and what kind of support has been developed to MCR, identifying their strengths and limitations. Such a comprehensive overview of MCR research is helpful for researchers to understand what has already been explored and open questions. It is also useful for practitioners, who can learn how to achieve better outcomes with MCR and becoming aware of existing solutions that can be adopted to support the practice. We next detail our systematic review protocol, describing the strategies to retrieve as many literature contributions as possible to achieve our research goal.

3.1.1 Research Questions

Based on our goal, our main research question is: *what is the state of the art of research on modern code review?* More specifically, we aim to answer the following specific research questions (RQ):

- **RQ-1**: What foundational body of knowledge has been built based on studies of MCR?
- **RQ-2**: What approaches have been developed to support MCR?
- **RQ-3**: How have MCR approaches been evaluated and what were the reached conclusions?

Table 3.1: Search results by source.

Databases	**URL**	**#Studies**
ACM Digital Library	http://portal.acm.org	256
IEEE Xplore Digital Library	http://ieeexplore.ieee.org	321
ScienceDirect	http://www.sciencedirect.com	101
Springer Link	http://link.springer.com	66
Duplicates		42
Total (including duplicates)		**744**
Total (excluding duplicates)		**702**

3.1.2 Search Strategy

The search strategy defines how to retrieve the primary studies of a SLR based on its research questions. To find the studies that are relevant to our work, we selected the four databases shown in Table 3.1. These databases store papers published in many key software engineering conferences and journals. Our search does not include other databases, such as Google Scholar and arXiv, because they provide pointers to papers already stored in our searched databases or include non-peer-reviewed papers. We focused on papers that went through a peer review selection process as a means to obtain evidence of the quality of their research.

To identify the state of the art on MCR, we specified a search string to query the selected databases and retrieve papers. Our focus is on MCR—a code reviewing practice, which is tool-based, informal, occurring regularly integrated with the software development—but the literature has been using different terms to refer to it, as mentioned in the provided background on MCR. Consequently, we used a single term as part of our search string, namely *code review*. To construct our search string, we also added the following alternative terms (synonyms): *code inspection*, *software inspection*, and *formal inspection*. The final search string is as follows.

Search String: code review OR code inspection OR software inspection OR formal inspection

Therefore, we identified whether papers focus on MCR not by the term they used but by how they described the practice. We did not include synonyms of MCR that are already covered by the terms in our search string, e.g. change-based code review.

Table 3.2: Inclusion criteria (IC), exclusion criteria (EC) and the number of papers satisfying them.

Criteria		Abstract Analysis	Full Text Analysis
IC-1	The paper presents a study of foundational aspects of MCR.	196	65
IC-2	The paper proposes a solution to support MCR.	124	46
IC-3	The paper presents an evaluation of an MCR approach.	3	28
EC-1	The paper does not focus on MCR as a reviewing practice made by peers within the software development.	60	27
EC-2	The paper is not written in English.	3	2
EC-3	The content of the paper was also published in another more complete paper, which is already included.	3	8
EC-4	We have no access to the full paper.	2	0
EC-5	The content is not a scientific paper of at least four pages.	56	13
Selected Papers		322	110

3.1.3 Selection Criteria

The papers retrieved by querying search databases are filtered using selection criteria used to identify primary studies relevant to answer our research questions. In this study, we specified three inclusion criteria (IC) and five exclusion criteria (EC), as summarized in Table 3.2.

In order to be selected, a primary study must satisfy at least one inclusion criterion and no exclusion criterion. Each of our RQs has a corresponding IC, which leads to three general types of studies: (i) FOUNDATIONAL STUDIES that investigate aspects of MCR or an experience report of the use of MCR (IC-1); (ii) PROPOSALS of solutions to support the MCR process, such as a tool, technique or method (IC-2); and (iii) EVALUATIONS of MCR solutions (IC-3). Throughout the paper, the terms highlighted are used to refer to these study types. As said above, to be classified as a work on MCR, the paper does not necessarily need to use a particular term, but focus on a lightweight form of code review.

Our EC-1 excludes studies that use MCR as a motivation or as an example of context where a solution can be applied. Their focus is on techniques that can be incorporated to MCR. For example, this is the case of static analysis approaches that can be used by reviewers or as part of an automated reviewer but can also be applied to any context to automatically analyze the source code. This EC also excludes studies that focus on code review being used for purposes other than software development, such as

teaching software engineering. Lastly, the criteria EC-2 to EC-5 are used to guarantee that only non-duplicated scientific primary studies, published as full papers in English, are included.

3.2 Review Execution

Our review protocol detailed above was executed to retrieve the investigated primary studies. The results of our search and selection are detailed as follows.

3.2.1 Search Execution

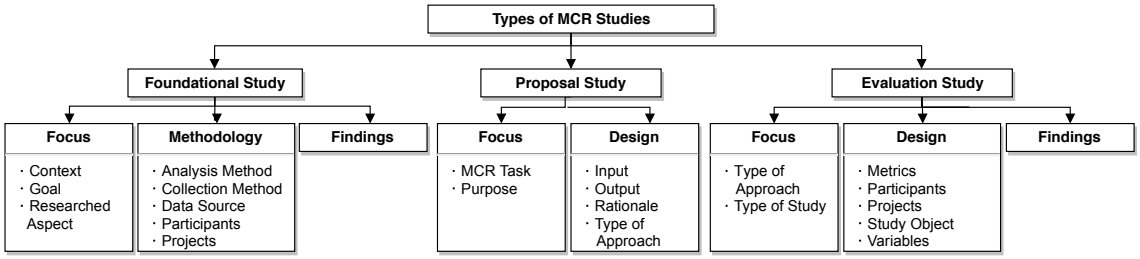
Each target database has a specific syntax in the search engine. Therefore, the search string was customized to each of the selected databases. We searched within the abstracts of the publications. However, due to limitations in the Springer Link database, which does not allow searches to be performed in abstracts, we searched for paper keywords instead. The query in the four selected databases was performed on July 9, 2018. Retrieved and selected papers were analyzed over the course of 1.5 years. As result, we obtained 702 papers, excluding duplicates. Detailed statistics of retrieved papers by database are provided in Table 3.1.

3.2.2 Selection of Primary Studies

Based on the results of the search execution, relevant primary studies were selected using a two-step procedure. In the first step, we analyzed the title and abstract of each retrieved paper to find any evidence that the study matched one of the inclusion criteria. When there was a match, the paper was selected for detailed analysis. In the second step, we analyzed the full text of these papers to evaluate the satisfaction of the selection criteria, if available (if not, it was excluded due to EC-4).

Included papers in both selection steps were associated with at least one IC. Papers that do not focus on MCR do not match any IC. For example, there are studies that are not included because they focus on software inspection as a synchronous process with an inspection meeting. We identified publications satisfying more than one IC. Some papers, e.g. Rahman, Roy and Kula (2017), describe a combination of foundational studies, a

Figure 3.1: Facets analyzed in each primary study.



Source: Author

proposed solution, and an evaluation. In Table 3.2, we present the results of each step of this selection and classification.

Following the procedure defined in our protocol, there are papers discarded due to the satisfaction of ECs. Due to EC-1, we excluded studies of pedagogical code review (HUNDHAUSEN; AGARWAL; TREVISAN, 2011; PETERSEN; ZINGARO, 2018) and static analysis tools (KAWAHARA, 2016; SINGH et al., 2017). As said, automated source code analysis can be used in other contexts and, if we had not excluded these studies, any approach that performs static analysis should have been included.

Each primary study was analyzed by a single researcher following the SLR protocol specified above. If the researcher could not clearly evaluate the satisfaction of any of the criteria, the opinion of a second researcher was requested in order to minimize the potential researcher bias. If there was no agreement, both researchers discussed until they converged to a decision.

Our final set of primary studies selected for review in our SLR has a total of 110 publications. From the 110 publications, 27 present multiple types of study, such as PROPOSAL and EVALUATION or FOUNDATIONAL STUDY and PROPOSAL. In Table 3.3, we show all the studies included for analysis.

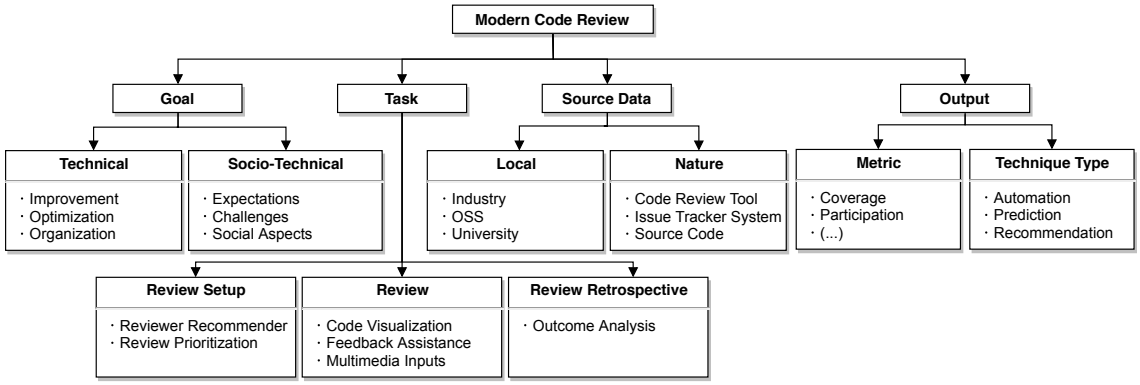
3.2.3 Analysis Method

We analyzed our selected primary studies split into the three groups, as shown in Table 3.3, focusing on answering each of our research questions. To answer each research question, we extracted information from each study according to the facets indicated in Figure 3.1. As aforementioned, there are papers that include a combination of FOUNDATIONAL STUDIES, PROPOSALS and EVALUATIONS. In these cases, each contribution of the paper was analyzed separately.

Table 3.3: Selected primary studies.

FOUNDATIONAL STUDIES
Müller (2005), McIntosh et al. (2014), Shimagaki et al. (2016), Bacchelli and Bird (2013), Albayrak and Davenport (2010), Beller et al. (2014), German et al. (2018), Rahman and Roy (2017), Sadowski et al. (2018), Bosu, Greiler and Bird (2015), Thompson and Wagner (2017), Yang et al. (2017), Kitagawa et al. (2016), Izquierdo-Cortazar et al. (2017), Bosu and Carver (2012), Spadini et al. (2018), Baum et al. (2016b), Bird, Carnahan and Greiler (2015), Santos and Nunes (2017), Kononenko, Baysal and Godfrey (2016), Bosu and Carver (2014), Thongtanunam et al. (2015), Kovalenko and Bacchelli (2018), Efstathiou and Spinellis (2018), Bosu et al. (2014), Rahman, Roy and Kula (2017), Begel and Vrzakova (2018), Dunsmore, Roper and Wood (2000), Hirao, Ihara and Matsumoto (2015), Thongtanunam et al. (2016), Floyd, Santander and Weimer (2017), Uwano et al. (2006), Lee and Carver (2017), Meneely et al. (2014), Bosu et al. (2017), Sutherland and Venolia (2009), Chandrika, Amudha and Sudarsan (2017), Asundi and Jayant (2007), MacLeod et al. (2018), Ueda et al. (2017), Armstrong, Khomh and Adams (2017), Thongtanunam et al. (2015), Ebert et al. (2017), Paixao et al. (2017), Kononenko et al. (2015), Bavota and Russo (2015), Runeson and Andrews (2003), Bosu and Carver (2013), Liang and Mizuno (2011), Biase, Bruntink and Bacchelli (2016), Panichella et al. (2015), Swamidurai, Dennis and Kannan (2014), Morales, McIntosh and Khomh (2015), Baysal et al. (2012), Rigby et al. (2012), Bernhart and Grechenig (2013), Duraes et al. (2016), Murakami, Tsunoda and Uwano (2017), Li et al. (2017), Thongtanunam et al. (2017), Baum, Leßmann and Schneider (2017), McIntosh et al. (2016), Baysal et al. (2016), Hirao et al. (2016), Kerzazi and Asri (2016)
PROPOSALS
Yu et al. (2016), Jiang et al. (2017), Rigby and Bird (2013), Bosu, Greiler and Bird (2015), Müller et al. (2012), Balachandran (2013), Barnett et al. (2015), Rahman, Roy and Collins (2016), Zhang et al. (2015), Sripada, Reddy and Khandelwal (2016), Rahman, Roy and Kula (2017), Thongtanunam et al. (2014), Hao et al. (2013), Tao and Kim (2015), Priest and Plimmer (2006), Soltanifar, Erdem and Bener (2016), Duley, Spandikow and Kim (2010), Uwano et al. (2006), Kalyan et al. (2016), Ge et al. (2017), Zanjani, Kagdi and Bird (2016), Tymchuk, Mocci and Lanza (2015), Ahmed et al. (2017), Baum, Schneider and Bacchelli (2017), Baum et al. (2016a), Nagoya, Liu and Chen (2005), Lanubile and Mallardo (2002), Harel and Kantorowitz (2005), Thongtanunam et al. (2015), Ebert et al. (2017), Pangsakulyanont et al. (2014), Xia et al. (2017), Zhang et al. (2011), Rigby et al. (2012), Mishra and Sureka (2014), Ouni, Kula and Inoue (2016), Menarini, Yan and Griswold (2017), Perry et al. (2002), Wang et al. (2017), Xia et al. (2015), Aman (2013), Fejzer, Przymus and Stencel (2018), Fan et al. (2018), Freire, Brunet and Figueiredo (2018), Li et al. (2017), Baum and Schneider (2016)
EVALUATIONS
Yu et al. (2016), Khandelwal, Sripada and Reddy (2017), Müller et al. (2012), Balachandran (2013), Yang et al. (2017), Barnett et al. (2015), Rahman, Roy and Collins (2016), Zhang et al. (2015), Rahman, Roy and Kula (2017), Thongtanunam et al. (2014), Tao and Kim (2015), Baum et al. (2016), Duley, Spandikow and Kim (2010), Peng et al. (2018), Hannebauer et al. (2016), Ge et al. (2017), Zanjani, Kagdi and Bird (2016), Thongtanunam et al. (2015), Xia et al. (2017), Ouni, Kula and Inoue (2016), Menarini, Yan and Griswold (2017), Xia et al. (2015), Aman (2013), Fejzer, Przymus and Stencel (2018), Fan et al. (2018), Freire, Brunet and Figueiredo (2018), Mizuno and Liang (2015), Runeson and Wohlin (1998)

Figure 3.2: Overview of MCR research based on primary studies of our SLR.



Source: Author

To label the information collected from the studies associated with each facet, we used coding. *Coding* is used to derive a framework of thematic ideas of qualitative data (text or images) by indexing data (GIBBS, 2007). All studies were initially broadly analyzed to identify an initial set of codes. Then, studies were analyzed in-depth, with the codes being refined in the process. Finally, some codes were merged (when they conveyed the same underlying idea), resulting in our final set of codes. The researchers of this work had many discussion meetings for the definitions and refinements of the codes. The studies were labeled with the codes by the author of this thesis and, in cases of uncertainty, the author and co-advisor both analyzed the study and discussed until they converged to the codes that should be used. The coding process was performed at different levels. First, it was used to classify the studies, from which we derived taxonomies for FOUNDATIONAL STUDIES and PROPOSALS, and types of EVALUATIONS. Second, within groups, we also used coding to categorize the study goals, characteristics, and identify their findings (as shown in Figure 3.1).

An analysis of the set of primary studies of MCR selected for being investigated lead us to identify key topics on this research area. We summarize them in Figure 3.2, highlighting (i) their goals, which can be technical or socio-technical; (ii) the MCR task that is target of the work—review setup, review, and review retrospective; (iii) the source data used for developing an approach or conducting a study; and (iv) the output associated with the contribution of the work (which can be assessed metrics or technique type).

Given that we now have introduced MCR studies of our SLR and the method used to analyze them, we next proceed to the results. The next two chapters present our findings[1] of FOUNDATIONAL STUDIES and then PROPOSALS and their EVALUATION).

[1]The complete analysis of each individual primary study can be found online at <http://inf.ufrgs.br/prosoft/resources/2020/tosem-code-review-slr>.

4 FOUNDATIONAL RESEARCH ON MCR

From the identified primary studies, 65 are classified as FOUNDATIONAL STUDIES. These studies aim to provide a better understanding of how MCR works in practice and how it is perceived by practitioners. We next present an overview of the different topics investigated in these studies and, following an in-depth analysis, we discuss their findings.

4.1 Overview

We classified the investigation made in FOUNDATIONAL STUDIES into six groups, as presented in Table 4.1. The first two groups use subjective or qualitative data to provide insights of the adoption of MCR. There are 13 studies—classified as *practitioner perception of the practice*—that collected the opinion of professionals with respect to different topics related to MCR, such as its benefits and challenges. The goal of these studies is to capture the state of the practice of MCR from the developers' perspective. Complementing these studies, there are four *experience reports* that share experiences from the use of MCR in real settings, deriving lessons learned or recommendations considering observations made across multiple case studies.

Given that MCR is tool-based, there are repositories that store code review activity, creating databases of historical information. This allowed many studies to be conducted, exploring objective data available in these repositories. In total, 38 studies went in this direction, providing insights into the content and frequency of review outcomes as well as the relationship between influence factors and the outcomes of MCR in various contexts. These studies are classified into two groups. The first group is *analysis of internal outcomes*, which includes studies that investigate the factors (e.g. patch size) that influence the MCR outcomes related to the MCR process, such as involved reviewers and provided feedback. The second group is *analysis of external outcomes*, which focuses on identifying how factors influence MCR outcomes that are externally visible, that is, code and product quality.

In addition to the common types of data investigated to understand the adoption of MCR, a smaller set of studies explored simulations or human body data (e.g. brain and eye activity) to understand *reviewer behavior*. These seven studies focused on understanding the reviewer motivation to participate in reviews and their behavior while checking the

Table 4.1: Taxonomy of topics researched in groups of FOUNDATIONAL STUDIES.

Topic	Description
Practitioner Perception of the Practice	
Code Review Process	Understanding of the inner-workings of the MCR practice
Social Dynamics	Understanding of the impact of MCR on the social dynamics of peers
Expected Benefits	Investigation of the main expected MCR benefits
Challenges and Difficulties	Investigation of the challenges and difficulties that emerge from the use of MCR
Experience Report	Sharing of insights of the MCR practice based on experience or the analysis of a case study of the use of MCR
Analysis of Internal Outcomes	
Reviewer Team	Understanding of the reviewer participation in code reviews and the factors that contribute to it
Review Feedback	Understanding of the content and proportion of comments of a code review
Review Intensity	Understanding of the iterative code review process and its outcomes
Review Time	Understanding of the delay and duration of a review request
Review Decision	Understanding of the outcome of a review request (its rejection or acceptance)
Analysis of External Outcomes	
Code Quality	Understanding of the effect of MCR on the quality of source code reviewed
Product Quality	Understanding of the influence of MCR on the incidence of defects in the software product
Reviewer Behavior	Understanding of the reviewer behavior, which includes how reviewers check the code change and their willingness to participate in MCR
Relationship with MCR	
Comparison with Techniques	Investigation of the outcomes of MCR in comparison with other verification techniques
Interaction with Development Practices	Investigating of the relationship between other development practices and MCR

code. Finally, there are papers that did not focus on internal aspects of MCR, but its relationship with other practices. These other seven papers, classified as *relationship with MCR*, evaluated how MCR compares to other verification techniques (such as pair programming) or its impact on other practices, such as automated build. We next analyze each of these groups of studies, highlighting their key findings.

4.2 Practitioner Perception of the Practice

We identified in the literature 13 FOUNDATIONAL STUDIES that explored multiple perspectives of the MCR practice by means of collecting the perceptions of the practitioners. To obtain the opinion and thoughts of practitioners that participated in the studies, most of the researchers used as instrumentation questionnaires (GERMAN et al., 2018; KONONENKO; BAYSAL; GODFREY, 2016; BOSU et al., 2017; SUTHERLAND; VENOLIA, 2009; BOSU; CARVER, 2013; BERNHART; GRECHENIG, 2013; BAUM;

LESSMANN; SCHNEIDER, 2017) or semi-structured interviews (BOSU; GREILER; BIRD, 2015; SPADINI et al., 2018; BAUM et al., 2016b; BIRD; CARNAHAN; GREILER, 2015). In one case (SADOWSKI et al., 2018), both forms of data collection were adopted. Two studies conducted at Microsoft (BACCHELLI; BIRD, 2013; MACLEOD et al., 2018) complemented their studies by also using observation to collect data. These FOUNDATIONAL STUDIES typically involve participants from the industry. Only five studies had as participants developers from OSS projects (GERMAN et al., 2018; SPADINI et al., 2018; KONONENKO; BAYSAL; GODFREY, 2016; BOSU et al., 2017; BOSU; CARVER, 2013).

The studies that provide insights of the practitioner perception of the practice focus on four difference MCR aspects, namely (i) *code review process*; (ii) *social dynamics*; (iii) *expected benefits*; and (iv) *challenges and difficulties*. The majority of the studies target only one aspect, while six of them (BACCHELLI; BIRD, 2013; SADOWSKI et al., 2018; BAUM et al., 2016b; KONONENKO; BAYSAL; GODFREY, 2016; BOSU et al., 2017; MACLEOD et al., 2018) study multiple aspects. We next detail the findings provided by these studies.

4.2.1 Code Review Process

More than a half of the studies that collected the opinion of practitioners (BOSU; GREILER; BIRD, 2015; BAUM et al., 2016b; KONONENKO; BAYSAL; GODFREY, 2016; BOSU et al., 2017; SUTHERLAND; VENOLIA, 2009; MACLEOD et al., 2018; BERNHART; GRECHENIG, 2013; BAUM; LESSMANN; SCHNEIDER, 2017) aim to better understand the inner-workings of the code review process. The goal is to identify the process variations and how developers perform code review.

Two studies at Microsoft (SUTHERLAND; VENOLIA, 2009; BOSU; GREILER; BIRD, 2015) aim to understand how the communication occurs during code reviews. First, a survey conducted in 2009 (SUTHERLAND; VENOLIA, 2009) explored the nature of dialog in code reviews, identifying different channels used for conversation during a review, as well as the perception of the usefulness of different alternatives. For practitioners, the dialogs helped them to understand the design rationale. Then, in 2015, a second study (BOSU; GREILER; BIRD, 2015) explored the review communication to characterize what makes a feedback useful according to developers. They indicated that useful comments is those that help the author to improve the code quality identifying

functional issues, validation issues, or alternatives scenarios, for example. Nevertheless, comments with questions for clarifications are considered not useful. Complementing these studies, Bernhart and Grechenig (2013) reported a positive effect of a continuous code review practice on the understandability and collective ownership in a particular project.

Considering OSS, Kononenko, Baysal and Godfrey (2016) analyzed the MCR practice in Mozilla and how developers perceive it. The study identified a dedicated group of developers responsible for reviewing code changes. Despite the availability of a specialized review tool, this group of developers relied on an issue tracker system to support the reviews. Similarly to the study made at Microsoft (SUTHERLAND; VENOLIA, 2009), this study found the use of both synchronous and asynchronous channels for communication. The authors also suggested that the review quality is associated with the thoroughness of the feedback and the reviewer's familiarity with the code, which is also observed in studies relying on objective data (discussed in later sections).

Two studies aimed to understand *why* code reviews are used (BAUM et al., 2016b) and how prevalent is the change-based format (BAUM; LESSMANN; SCHNEIDER, 2017). Baum et al. (2016b) interviewed professionals to identify factors influencing the adoption and the shape of MCR in the industry, reporting several variations observed in the review process. They also presented a list of factors shaping that process, which are organized into five categories: culture, development team, product, development process, and tools. The authors also reported that code review is most likely to remain in use when embedded into the development process. A posterior study (BAUM; LESSMANN; SCHNEIDER, 2017) refined this finding through a survey with professionals from commercial teams. This research concluded that the risk of review fade away increases when there are no rules or conventions, which supports previous findings. The study also reported that review based on code changes is prevalent among surveyed professionals.

4.2.2 Social Dynamics

MCR is a collaborative activity that relies on intensive human interaction. This motivated the researchers to investigate the social issues that emerges from the interaction among peers. There are three studies that focus on the social dynamics associated with MCR. These studies, as above, are based on the developers' perceptions. The first study (GERMAN et al., 2018) investigated *fairness*. It analyzed how practitioners per-

Table 4.2: Main expected benefits of the use of MCR.

Study	Position			
	First	**Second**	**Third**	**Fourth**
Bacchelli and Bird (2013)	Defect Identification	Code Improvement	Alternative Solutions	Knowledge Transfer
Baum et al. (2016b)	Code Improvement	Defect Identification	Learning	Responsibility
Bosu et al. (2017)	Code Improvement	Knowledge Sharing	Defect Identification	Community Building
Sadowski et al. (2018)	Learning	Norm Compliance	History Tracking	Gatekeeping
MacLeod et al. (2018)	Code Improvement	Defect Identification	Knowledge Sharing	Alternative Solutions

ceive the treatment given and received in the code review. The results indicated that a significant proportion of participants perceives unfairness in the MCR process. This observation is more common in authors than in reviewers. The other two studies (BOSU et al., 2017; BOSU; CARVER, 2013) explored the *impression* of reviewers about their *teammates*, more specifically, how it is formed and its impact on the practice. A key finding in both these studies is that MCR might impact on the impression formation, especially in building a perception of expertise. However, a poorly made code change may negatively impact this impression, also affecting how reviewers treat particular authors of changes in future reviews.

4.2.3 Expected Benefits

The key benefit expected by the use of software inspection was the early defect identification. Most of the studies of MCR analyzed in our SLR also reported this as an expected benefit. However, only one of the surveys (BACCHELLI; BIRD, 2013) that investigated this reported defect identification as the primary expected benefit. The other studies, performed after this survey, also pointed this out as an expectation, but they reported other benefits as the main expected benefit, as shown in Table 4.2. This table details the main expected benefits according to the five studies, listed in chronological order, that investigated this aspect.

As can be seen, three surveys reported the improvement of code quality as the key desired benefit and, in the other two studies, this is the second benefit. Moreover, learning and knowledge transfer have also been identified as an expected result of MCR, especially at Google (SADOWSKI et al., 2018). Therefore, beyond technical effects, practitioners

also expect to gain non-technical benefits when adopting code review. This is also observed in the results of surveys made at Microsoft. In a study done in 2013 (BACCHELLI; BIRD, 2013), defect identification was the key expectation among developers. However, in the studies performed in 2017 (BOSU et al., 2017) and 2018 (MACLEOD et al., 2018), the key expectation was code improvement, pushing defect identification to the second position.

In summary, we observed that (1) the two main expected benefits of MCR are code quality improvement and defect identification; and (2) MCR promotes additional benefits, such as knowledge sharing and learning.

4.2.4 Challenges and Difficulties

Three of the studies (BAUM et al., 2016b; SADOWSKI et al., 2018; MACLEOD et al., 2018) that identified the benefits expected from MCR also investigated what difficulties developers face when performing the code review activity, especially in an industrial environment. Additionally, there are two works that also explore this (KONONENKO; BAYSAL; GODFREY, 2016; SPADINI et al., 2018).

The results of the extensive survey done at Microsoft (BACCHELLI; BIRD, 2013), from which we already reported findings, pointed out that *understanding* of code changes is the main challenge for reviewers. This was identified in interviews and further explored by the researchers, who also identified that the most challenging task from the understanding perspective is finding defects, followed by the suggestion of alternative solutions. In a more recent study conducted in 2018 at the same company (MACLEOD et al., 2018), practitioners were asked to rank the challenges of code review by importance. Understanding of the purpose of the code and the motivations for the change remain among the top five difficulties. Furthermore, studies at Mozilla (KONONENKO; BAYSAL; GODFREY, 2016) and Google (SADOWSKI et al., 2018) also reported in their findings the difficulty of practitioners to gain familiarity with the code and the misunderstandings that can arise based on not knowing what caused a particular a change. Another study (SPADINI et al., 2018) explored review of test code and found an additional challenge. Reviewing test files requires developers to understand not only the code being reviewed (test files) but also the associated production files.

The other challenges reported in these studies are more scattered. Two of the five studies (KONONENKO; BAYSAL; GODFREY, 2016; MACLEOD et al., 2018) indi-

cated the time management as a challenge faced during the review activity. In the study that focused on reviewing test code (SPADINI et al., 2018), developers indicated that they have a limited amount of time to spend on reviewing, being driven by management policies to review production code instead of test code, which imposes a difficulty to review this type of code. Tools are also reported as challenges in more than one study (SADOWSKI et al., 2018; KONONENKO; BAYSAL; GODFREY, 2016) because they are not suitable for a particular context or its customization may lead to misunderstanding.

In summary, *code comprehension* has been the main challenge faced by developers when reviewing a code change. Other difficulties are also reported, such as time pressure and tool support.

4.3 Experience Report

The majority of the studies reported in the previous section collected data using surveys and interviews. In order to also understand MCR in practice, four studies went to another direction: they relied on experience; in most cases, case studies. Three of these works focus on sharing insights of the use of MCR in particular projects (BIRD; CARNAHAN; GREILER, 2015; IZQUIERDO-CORTAZAR et al., 2017; BAYSAL et al., 2012), while one (RIGBY et al., 2012) provided lessons learned and recommendations based on previous experiences.

To better understand how MCR affects software development, two studies (BAYSAL et al., 2012; IZQUIERDO-CORTAZAR et al., 2017) measured a set of outcomes from code review from specific projects. Izquierdo-Cortazar et al. (2017) presented an analysis conducted in the Xen Project to understand how the performance of code review evolves. Together with developers of the Xen Project, the researchers analyzed the evolution of comments and patch series, the time-to-merge, and the impact of patch series complexity, using another project as a benchmark. The main contribution of this work is the approach they used for the analysis of code review data. In another study, Mozilla Firefox, an OSS project, had the change on its patch lifecycle examined by Baysal et al. (2012). They verified the differences between pre- and post-rapid release development, assuming that this change affects how code is reviewed. Their findings suggest a mostly unchanged lifecycle, considering the states of the patch lifecycle (landed, resubmitted, abandoned, and timeout). But, because patches are reviewed faster in the post-rapid release, contributions from core developers are rejected faster and those from casual con-

tributors are accepted faster. This indicates that switching to post-rapid release is a successful approach to reduce the time patches wait for review, mainly considering patches from casual contributors, which are disproportionally more likely to be abandoned.

The third work that relied on a case study (BIRD; CARNAHAN; GREILER, 2015) described the building and the use of CodeFlow Analytics (CFA), an internal platform from Microsoft, which allows developers to explore the historical data of code review. Besides a description of CFA, the researchers shared the challenges faced and decisions made during the development of CFA. For example, they focused on many ways to access the data, considering multiple types of users, but faced challenges due to the diversity of tools and data sources used at Microsoft. The authors of this study also presented findings of interviews with users of the CFA about their experiences, how they discovered the tool, why and how they use it, the impact of this usage, and the faced challenges. Therefore, the study presented several findings and lessons learned from the experience with CFA. A highlighted contribution is the observation of a positive impact of the tool on development teams and the possibility of research on many aspects of code review using CFA.

Finally, Rigby et al. (2012), with previous experiences obtained by examining multiple code reviews in OSS case studies, presented five lessons learned and three recommendations that are arguably transferable to proprietary projects. They shared as lessons learned: (i) asynchronous reviews; (ii) frequent reviews; (iii) incremental reviews; (iv) experienced reviewers to conduct reviews; and (v) empower expert reviewers, letting them self-select changes and assigning reviewers when nobody selects a request. They also recommended the usage of lightweight review tools, nonintrusive metrics, and the implementation of a review process.

4.4 Analysis of MCR Outcomes

A substantial amount of FOUNDATIONAL STUDIES mined objective data from code review repositories to derive findings, being this the most common type of studies on MCR. These studies analyze the outcomes of code review and how they are influenced by characteristics of a particular review, i.e., *influence factors*. We split these outcomes into two groups: (i) *internal outcomes*, which are those associated with the MCR process, with a total of 29 studies; and (ii) *external outcomes*, which are those observed in the software product (code quality and defects), with a total of 12 studies.

We overview the studies of the relationship between influence factors and out-

comes in Figure 4.1. This figure shows on the left-hand side the number of publications exploring a factor, while on the right-hand side there is an indication of how many times the influence of a factor over an MCR outcome was investigated. Table 4.3 then describes both non-technical and technical investigated factors, indicating which studies explored its influence. Similarly, Table 4.4 shows a description of the internal outcomes examined as an influencing factor. We identified 24 studies exploring the correlation between 40 influence factors and 14 outcomes. Moreover, the majority of these studies used data from OSS repositories, while four (BOSU; GREILER; BIRD, 2015; RAHMAN; ROY; KULA, 2017; SANTOS; NUNES, 2017; SHIMAGAKI et al., 2016) of them collected information from commercial projects and two (ALBAYRAK; DAVENPORT, 2010; MURAKAMI; TSUNODA; UWANO, 2017) from experiments. The results of these studies are discussed as follows.

4.4.1 Internal Outcomes

Internal outcomes of code review are those associated with characteristics of the MCR process. Examples of such characteristics are the number of involved reviewers, the number of provided comments and the amount of time that reviewers took to reach a decision about a review request.

Some studies are limited to the characterization of internal outcomes of specific projects. They assess and analyze the values of these outcomes. Others go further and inspect the relationship between influence factors and outcomes. From the 29 works that focused on internal outcomes, 23 used data from OSS projects, while the remaining ones used data from proprietary projects (BACCHELLI; BIRD, 2013; BOSU; GREILER; BIRD, 2015; SANTOS; NUNES, 2017; RAHMAN; ROY; KULA, 2017). Moreover, two of the studies collected and used data produced by participants of an experiment (RAHMAN; ROY; KULA, 2017; MURAKAMI; TSUNODA; UWANO, 2017).

We grouped the internal outcomes analyzed by the 29 identified works into five groups: *reviewer team*, *review feedback*, *review intensity*, *review time*, and *review decision*. We next discuss the findings associated with each group of internal outcome.

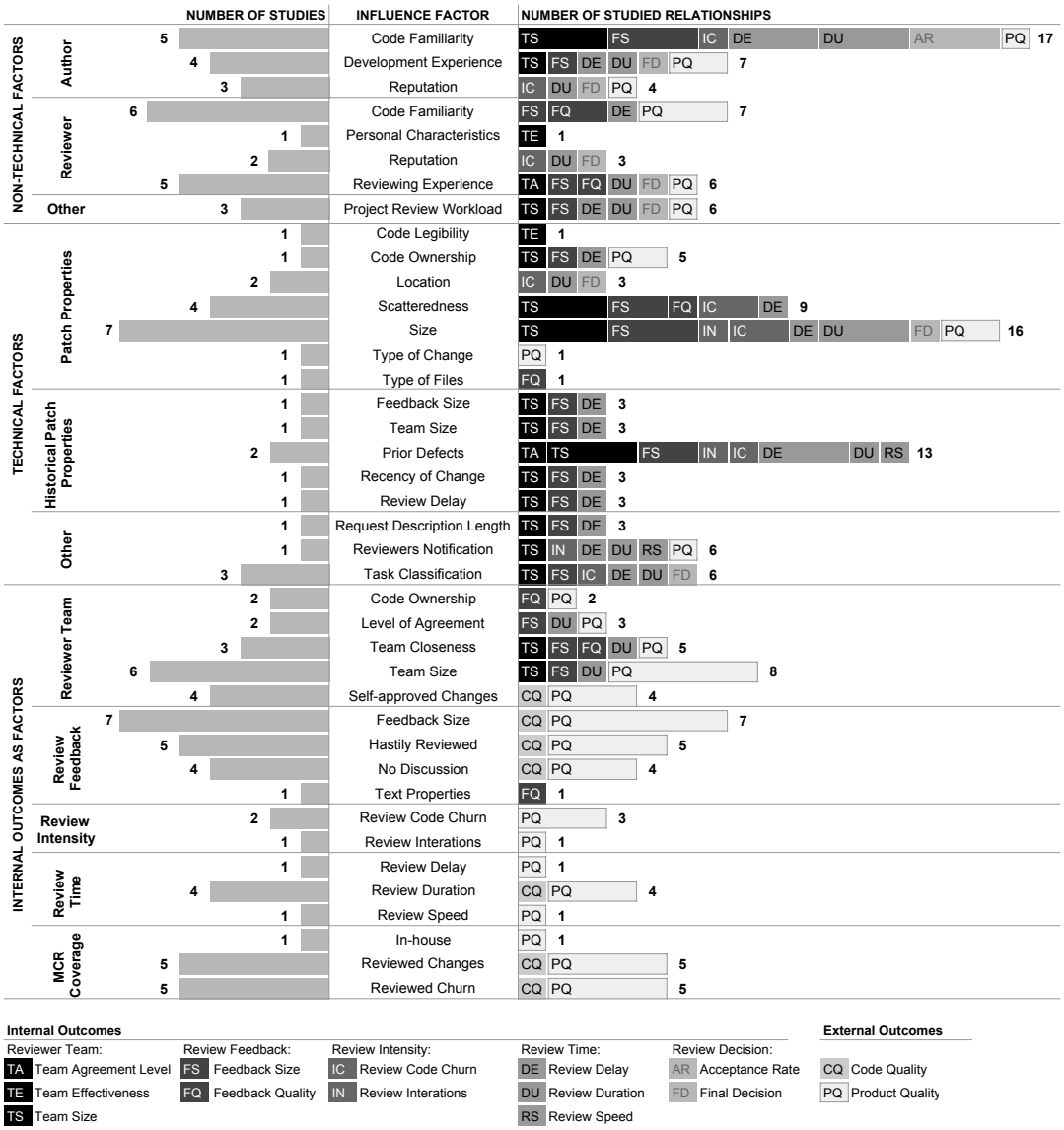
Table 4.3: Influence factors analyzed by the FOUNDATIONAL STUDIES.

Influence Factor		Description	Studies
Non-Technical Factors			
Author	Code Familiarity	Number of contributions made by a developer as code author in the project	Kovalenko and Bacchelli (2018), Lee and Carver (2017), Bosu and Carver (2014), Thongtanunam et al. (2017), Kononenko et al. (2015)
	Development Experience	Number of contributions made by a developer as a code author in general	Thongtanunam et al. (2017), Baysal et al. (2016), Bosu et al. (2014), Kononenko et al. (2015)
	Reputation	Technical characteristics of the developer, such as working company, who authored the code	Beller et al. (2014), Baysal et al. (2016), Bosu et al. (2014)
Reviewer	Code Familiarity	Number of contributions made by a developer as reviewer in the project	Bosu, Greiler and Bird (2015), Thongtanunam et al. (2017), Kononenko et al. (2015), Rahman, Roy and Kula (2017), Meneely et al. (2014), McIntosh et al. (2016)
	Personal Characteristics	Characteristics of the reviewer, such as age	Murakami, Tsunoda and Uwano (2017)
	Reputation	Technical characteristics of the developer, such as working company, who reviewed the code	Beller et al. (2014), Baysal et al. (2016)
	Reviewing Experience	Number of completed reviews of code changes	Hirao et al. (2016), Thongtanunam et al. (2017), Rahman, Roy and Kula (2017), Baysal et al. (2016), Kononenko et al. (2015)
Project's Review Workload		Number of the review requests submitted to the code review tool in a period	Thongtanunam et al. (2017), Baysal et al. (2016), Kononenko et al. (2015)
Technical Factors			
Patch Properties	Code Legibility	Presence of poor programming practices that affect code legibility or maintainability	Albayrak and Davenport (2010)
	Code Ownership	Number of developers who submitted patches that impact the same files as the patch under review	Thongtanunam et al. (2017)
	Location	Location in the code change, such as the module it belongs to	Beller et al. (2014), Baysal et al. (2016)
	Scatteredness	Measure of the dispersion of the change, such as the number of files or directories in a review request	Liang and Mizuno (2011), Bosu, Greiler and Bird (2015), Beller et al. (2014), Thongtanunam et al. (2017)
	Size	Number of lines of code added or modified in the code change under review	Santos and Nunes (2017), Liang and Mizuno (2011), Thongtanunam et al. (2017), Beller et al. (2014), Baysal et al. (2016), Bosu et al. (2014), Meneely et al. (2014)
	Type of Changes	Indication of the new or modified files	Bosu et al. (2014)
	Type of Files	Indication of the type of the file, such as source code, scripts, or configuration	Bosu, Greiler and Bird (2015)
Past Patch Properties	Feedback Size	Number of messages that were posted in the reviews of prior patches that impact the same files as the patch under review	Thongtanunam et al. (2017)
	Number of Reviewers	Number of reviewers who provided feedback in the reviews of prior patches that impact same files as the patch under review	Thongtanunam et al. (2017)
	Prior Defects	Number of prior bug-fixing patches that impact the same files as the patch under review	Thongtanunam et al. (2017), Thongtanunam et al. (2015)
	Recency	Number of days since the last modification of the files	Thongtanunam et al. (2017)
	Review Delay	Feedback delays of the reviewers of prior patches received	Thongtanunam et al. (2017)
Others	Request Description Length	Number of words an author uses to describe a code change in a review request	Thongtanunam et al. (2017)
	Reviewers Notification	Indication of code review using broadcast (visible for all) or unicast (visible for a specific group) communication technology	Armstrong, Khomh and Adams (2017)
	Task Classification	Indication of the change type based on purpose or priority, such as high-level priority bug fixing	Thongtanunam et al. (2017), Beller et al. (2014), Baysal et al. (2016)

Table 4.4: Internal outcomes as influence factors analyzed by the FOUNDATIONAL STUDIES.

Influence Factor		Description	Studies
Review Team	Code Ownership	Number of developers who uploaded a revision for the proposed changes	Rahman, Roy and Kula (2017), Thongtanunam et al. (2015)
	Level of Agreement	The proportion of reviewers that disagreed with the review conclusions	Hirao et al. (2016), Thongtanunam et al. (2015)
	Team Closeness	Number of distinct geographically distributed development sites or number of distinct teams associated with the author and reviewers	Santos and Nunes (2017), Bosu, Greiler and Bird (2015), Meneely et al. (2014)
	Team Size	Number of reviewers that participate in the reviewing process	Santos and Nunes (2017), Thompson and Wagner (2017), Thongtanunam et al. (2015), Kononenko et al. (2015), Bavota and Russo (2015), Meneely et al. (2014)
	Self-approved Changes	The proportion of changes approved for integration only by the original author	Morales, McIntosh and Khomh (2015), McIntosh et al. (2014), Shimagaki et al. (2016), McIntosh et al. (2016)
Review Feedback	Feedback Size	Number of general comments and inline comments written by reviewers	Morales, McIntosh and Khomh (2015), Shimagaki et al. (2016), Thompson and Wagner (2017), Thongtanunam et al. (2015), Bavota and Russo (2015), Kononenko et al. (2015), McIntosh et al. (2016)
	Hastily Review	Number of hastily reviewed commits (changes approved for integration at a rate that is faster than 200 LOC/hour)	Morales, McIntosh and Khomh (2015), McIntosh et al. (2014), Shimagaki et al. (2016), Meneely et al. (2014), McIntosh et al. (2016)
	No Discussion	Number of review requests that are accepted without any review comments	Morales, McIntosh and Khomh (2015), Shimagaki et al. (2016), McIntosh et al. (2014), McIntosh et al. (2016)
	Feedback Properties	Measure of the reading ease, stop word ratio, question ratio, code element ratio, and conceptual similarity	Rahman, Roy and Kula (2017)
Review Intensity	Code Churn	Number of lines added and deleted between revisions	Thongtanunam et al. (2015), Shimagaki et al. (2016)
	Interations	Number of review iterations of a review request prior to its conclusion	Thongtanunam et al. (2015)
Review Time	Review Delay	Time from the first review request submission to the posting of the first reviewer feedback	Thongtanunam et al. (2015)
	Review Duration	Time from the first review request submission to the review conclusion	Morales, McIntosh and Khomh (2015), Thongtanunam et al. (2015), Shimagaki et al. (2016), McIntosh et al. (2016)
	Review Speed	Rate of lines of code by an hour of a review request	Thongtanunam et al. (2015)
MCR Coverage	In-house	Ratio of internal contributions in the project	Shimagaki et al. (2016)
	Reviewed Changes	The proportion of changes committed to a component that are associated with code reviews	Morales, McIntosh and Khomh (2015), McIntosh et al. (2014), Shimagaki et al. (2016), Thompson and Wagner (2017), McIntosh et al. (2016)
	Reviewed Churn	The proportion of code churn reviewed in the past	Morales, McIntosh and Khomh (2015), McIntosh et al. (2014), Shimagaki et al. (2016), Thompson and Wagner (2017), McIntosh et al. (2016)

Figure 4.1: Studies that analyzed the relationship between influence factors and outcomes.



Source: Author

4.4.1.1 Reviewer Team

There are 13 studies that investigated the properties associated with the team of reviewers that are formed in code reviews. Few works (YANG et al., 2017; BOSU; CARVER, 2012; ASUNDI; JAYANT, 2007) collected objective data with the single purpose of characterizing the participation of reviewers. The other remaining works examined the influence of different factors over outcomes related to reviewer teams.

Most of the studies that analyze reviewer teams investigate the number of reviewers that engage in a code review, i.e. the *team size*. Three studies (YANG et al., 2017; BOSU; CARVER, 2012; ASUNDI; JAYANT, 2007) reported the number of reviewers

that responded to a review request in OSS projects, presenting an average of one or two reviewers per request. In addition, another seven studies (THONGTANUNAM et al., 2017; KOVALENKO; BACCHELLI, 2018; LEE; CARVER, 2017; SANTOS; NUNES, 2017; LIANG; MIZUNO, 2011; THONGTANUNAM et al., 2015; ARMSTRONG; KHOMH; ADAMS, 2017) explored factors influencing the team size.

The low number of active reviewers in reviews motivated Thongtanunam et al. (2017) to investigate the characteristics of patches that suffer from a lack of review participation, exploring several factors. As a key finding, they identified that the description length is a strong indicator of whether a patch is likely to be selected by reviewers—long patch descriptions might increase the likelihood of attracting reviewers. Santos and Nunes (2017), in turn, found a negative effect of large patches, indicating that the number of active reviewers decreases when the patch size increases. They analyzed a software project that involved distributed teams and, with respect to distribution, they concluded that the team size decreases when more locations and teams are involved, which should be considered when inviting reviewers from other sites.

Another investigated aspect is the impact of familiarity with the project code on the team size. Three studies (KOVALENKO; BACCHELLI, 2018; LEE; CARVER, 2017; THONGTANUNAM et al., 2017) analyzed the relationship between the number of previous contributions, i.e. changes of an author (as a means of measuring familiarity with the project) and the size of the team of reviewers. While two of the studies (KOVALENKO; BACCHELLI, 2018; THONGTANUNAM et al., 2017) did not find a significant relationship, Lee and Carver (2017) indicated a general trend that the number of active reviewers increases when the author's familiarity decreases, suggesting that newcomers receive more attention from invited reviewers.

Going in another direction, Yang et al. (2017) compared active and inactive reviewers of pull-requests. Their findings indicate that some super active reviewers lead code review, but inviting inactive reviewers would contribute to reducing the burden and speed up the process. In fact, according to Liang and Mizuno (2011), authors prefer to invite more experienced reviewers, considering historical data, corroborating with the idea that there is a group of few reviewers that are overloaded in MCR.

In addition to team size, two other internal outcomes associated with reviewer teams have been explored, namely *reviewer effectiveness* and *reviewer agreement level*. The former was investigated in two empirical studies (ALBAYRAK; DAVENPORT, 2010; MURAKAMI; TSUNODA; UWANO, 2017). One study (MURAKAMI; TSUNODA;

UWANO, 2017) analyzed the effect of reviewer age on the efficiency and correctness of code review, but their findings did not provide evidence of a significant difference. The second study (ALBAYRAK; DAVENPORT, 2010), in turn, examined how maintainability defects present in the code to be reviewed influences the effectiveness of reviewers, concluding that indentation issues have a negative impact on the reviewer performance.

The latter internal outcome, level of agreement among reviewers in review requests, was also investigated in two studies (THONGTANUNAM et al., 2015; HIRAO et al., 2016). One of them (THONGTANUNAM et al., 2015) analyzed the relationship between a file with prior defects and the level of review disagreement, but results did not show that there is a relationship between them. In another study (HIRAO et al., 2016), the influence of reviewers' reviewing experience on the frequency of their votes that disagreed with the review conclusions was analyzed. In this case, the findings suggested that more experienced reviewers are more likely to have a higher level of agreement than the less experienced reviewers.

4.4.1.2 Review Feedback

The internal outcome mostly investigated by the primary studies is review feedback, with a total of 17 papers. These studies explored the comments made by reviewers in code reviews. We identified three aspects that were examined: (i) content type (BACCHELLI; BIRD, 2013; SPADINI et al., 2018; EFSTATHIOU; SPINELLIS, 2018; BOSU et al., 2014; BIASE; BRUNTINK; BACCHELLI, 2016; LI et al., 2017; EBERT et al., 2017); (ii) size, typically in terms of amount of comments (BOSU; CARVER, 2012; LIANG; MIZUNO, 2011; THONGTANUNAM et al., 2017; LEE; CARVER, 2017; KOVALENKO; BACCHELLI, 2018; SANTOS; NUNES, 2017; THONGTANUNAM et al., 2015; HIRAO et al., 2016); and (iii) quality, in terms of, e.g., usefulness (RAHMAN; ROY; KULA, 2017; BOSU; GREILER; BIRD, 2015).

To further understand what has been discussed within code reviews, there is research work that explored the nature of dialogs and the concerns raised by human reviewers. Bacchelli and Bird (2013) manually analyzed and classified 570 MCR comments from Microsoft projects, creating categories for emerged themes. Spadini et al. (2018) reproduced this analysis using 600 comments from the test code review of OSS projects. Both studies identified code improvement, understanding, social communication, and defects as the most frequent discussion topics. In a more specific analysis, two other studies (BIASE; BRUNTINK; BACCHELLI, 2016; BOSU et al., 2014) investigated *security*

concerns raised in the review feedback, leading to identified categories related to the domain and language-specific issues (BIASE; BRUNTINK; BACCHELLI, 2016) and race conditions (BOSU et al., 2014). While the mentioned studies performed a manual analysis of the review comments, Li et al. (2017) proposed a taxonomy of review comments on pull-requests and an automatic classifier based on that taxonomy. They used the classifier to identify the typical review patterns in OSS projects and found that most are about code correction and social interactions. Ebert et al. (2017) also proposed a theoretical approach to support the analysis of review feedback. In this case, the researchers presented a framework for identifying confusion in textual comments. The conclusion is that humans can reasonably recognize confusion in review feedback. They also observed that the identification of confusion in inline comments is more difficult than in general ones.

Focusing on the feedback size, two studies (BOSU; CARVER, 2012; LIANG; MIZUNO, 2011) analyzed the amount of discussion by review in OSS projects and concluded that there is an average of two or three comments per review. This low number, together with the limited reviewer participation in reviews, motivated the investigation of various factors influencing the feedback, which was done in six papers. Three studies (KOVALENKO; BACCHELLI, 2018; LEE; CARVER, 2017; THONGTANUNAM et al., 2017) examined the relationship between the familiarity with the project of the author submitting the change and the feedback size. They concluded that there is a correlation between them. More specifically, their key finding is that the number of comments increases as the familiarity decreases, indicating higher involvement of reviewers in review requests made by novices. Hirao et al. (2016), in turn, provided evidence that a review request with a reviewer with a lower level of agreement is more likely to take a longer discussion length. Similarly to team size, the feedback size is also influenced by the patch size. Santos and Nunes (2017) reported that the larger the patch, the lower the comment density, according to their analysis. Despite this, Thongtanunam et al. (2017) indicated that the more lines changed in the patch, the more likely the patch is discussed. In previous work, Thongtanunam et al. (2015) also had found that risky files, i.e., files with prior defects, tend to undergo reviews that have shorter discussions and more revisions without reviewer feedback than normal files do.

Differently from the studies above, three studies observed the quality and technical aspects of the provided comments. Bosu, Greiler and Bird (2015) and Rahman, Roy and Kula (2017) explored factors that influence the usefulness of code review comments. The first work made this analysis at Microsoft, while the second explored the textual features

and developer experience to ground the proposal of a usefulness predictor. Both studies found that the code familiarity of the reviewer influences the feedback quality. Rahman, Roy and Kula (2017) also found some variation among textual properties between useful and non-useful comments. Motivated by these two studies (BOSU; GREILER; BIRD, 2015; RAHMAN; ROY; KULA, 2017), Efstathiou and Spinellis (2018) presented a preliminary investigation with OSS data to examine facets of language in the comments. They observed a collocation of source code and linguistic coherence in the review messages, suggesting that this might support future research on the analysis of usefulness in review comments.

4.4.1.3 Review Intensity

The works discussed above focus on the content of the code review. Now, we focus on work that targets the review intensity, which refers to analyses of the number of iterations made during code reviews and the code churn (the delta between the submitted and accepted code). Seven papers fall into this category (BELLER et al., 2014; BOSU; CARVER, 2014; BAYSAL et al., 2016; LIANG; MIZUNO, 2011; THONGTANUNAM et al., 2015; ARMSTRONG; KHOMH; ADAMS, 2017; UEDA et al., 2017). Two of them (UEDA et al., 2017; BAYSAL et al., 2016) investigate the content of the code churn, while the remaining papers make an analysis of the correlation between influence factors and this internal outcome type.

The two studies on the content of changes made in the code under review explored different aspects. Ueda et al. (2017) studied how the author of changes fixes `if` statements during the code review, identifying symbolic operators that are typically changed, such as parentheses. In contrast, Beller et al. (2014) explored the problem types fixed during code review, manually classifying the changes into a defect categorization, as well as analyzing what triggers them. The results of this work indicate a 75:25 ratio between evolvability and functional changes, being 10%–22% of these changes not triggered by review feedback.

Considering the code churn, the findings of studies on review intensity suggest that *technical* aspects potentially affect the delta between the submitted and accepted code. Beller et al. (2014) indicated that bug-fixing tasks lead to fewer changes, while patches with more altered files and lines of code lead to higher code churn. Similarly, Liang and Mizuno (2011) also explored technical aspects, but they did not found a strong correlation between patch content and the churn. In another direction, Bosu and Carver

(2014) analyzed the reputation of the author of the code change, i.e. core and peripheral developer, and its relation with several factors, including the code churn. Despite the several identified differences, the result of the analysis related to the number of patches per review requests is inconclusive.

Investigating the number of review iterations, Baysal et al. (2016) found that larger changes have more rounds of revisions. Thongtanunam et al. (2015), in turn, investigated review intensity in risky files, i.e. files with prior defects. Although their findings suggest that risky files tend to undergo reviews that have fewer iterations, the same analysis indicated that these files churn more during MCR. Finally, Armstrong, Khomh and Adams (2017) reported that patches reviewed using unicast technology (when a review request is visible for a targeted group), undergo more iterations than those reviewed using broadcasts technologies (when all those subscribed to a medium can see the review request).

4.4.1.4 Review Time

Time is another aspect of the MCR process that has also been investigated. Studies focused in particular on (i) review duration (the total amount of time taken to reach a decision), (ii) the first response delay, and (iii) review speed. In total, 13 papers targeted this topic. Four of them (YANG et al., 2017; BOSU; CARVER, 2012; THONGTANUNAM et al., 2015; KERZAZI; ASRI, 2016) characterized the review interval in particular contexts. The other nine papers explored the relationship between influence factors and review time aspects.

Yang et al. (2017) aimed to understand why code review is considered a time-consuming process. They analyzed pull-requests from Rails, an open-source project, and found that more than 40% of its pull-requests are closed in more than ten days, being considered a long time for reviewers to complete the review. Bosu and Carver (2012) also assessed a typical review interval in OSS projects together with the delay for the code author to receive the first feedback. The median review interval is 3–4 days, while the first review feedback is received promptly in most cases. Furthermore, also in the OSS context, Kerzazi and Asri (2016) examined both technical and socio-technical interactions among contributors in code review. They identified behavioral patterns among contributors, reporting that core developers are more likely to have a shorter review interval than peripheral developers (KERZAZI; ASRI, 2016). Lastly, Thongtanunam et al. (2015) investigated reviews with code-reviewer assignment problems and the impact in review duration. They analyzed the content of comments and found that: (i) 4%–30% of

the reviews have the assignment problem; and (ii) these reviews required, on average, 12 days longer to approve code changes.

In addition to the discussed findings on review duration, another eight studies (SANTOS; NUNES, 2017; BOSU; CARVER, 2014; THONGTANUNAM et al., 2015; KOVALENKO; BACCHELLI, 2018; LEE; CARVER, 2017; BAYSAL et al., 2016; ARMSTRONG; KHOMH; ADAMS, 2017; HIRAO et al., 2016) explored which factors might be related to the total amount of time taken to reach a review decision. Although some of them did not identify a significant relationship, others provided evidence of factors influencing review duration. The key findings of these studies indicate that the review takes less time when (i) the authors of code changes are more experienced and familiar with the project (BAYSAL et al., 2016; BOSU; CARVER, 2014; LEE; CARVER, 2017); (ii) the involved reviewers have high reviewing experience (BAYSAL et al., 2016); (iii) the review queue is short (BAYSAL et al., 2016); (iv) the patch size is small (SANTOS; NUNES, 2017); (v) there are few prior defects in the reviewed files (THONGTANUNAM et al., 2015); (vi) there are few active reviewers, and they are from the same team and location (SANTOS; NUNES, 2017); and (vii) the historical level of agreement of involved reviewers is high (HIRAO et al., 2016).

Other five studies (BOSU; CARVER, 2014; THONGTANUNAM et al., 2015; KOVALENKO; BACCHELLI, 2018; ARMSTRONG; KHOMH; ADAMS, 2017; THONGTANUNAM et al., 2017) conducted a similar analysis of the factors influencing the first response delay of a review request. Similarly as above, some studies did not find evidence to support this relationship, while others reported both technical and non-technical aspects associated with this outcome. Key findings indicate that authors of code changes that are more familiar with the project receive faster first feedback (BOSU; CARVER, 2014). In contrast, if a patch has files with prior defects (THONGTANUNAM et al., 2015) and these files historically received a slow response (THONGTANUNAM et al., 2017), then the first response also tends to be slower.

Another aspect of MCR that has been studied is the review speed (reviewed lines of codes per hour). Thongtanunam et al. (2015) investigated whether the speed varied depending on the presence of defects in a prior release, while Armstrong, Khomh and Adams (2017) studied the difference in the review speed when using unicast or broadcast as communication technology. A key finding of these studies is that files with prior defects tend to have a faster review rate than the reviews of normal files.

4.4.1.5 Review Decision

Finally, the last group of factors that influence internal outcomes investigates the result of code reviews (accept or reject), that is, the review conclusion. This is the least explored topic, according to our SLR, with seven identified papers (BAYSAL et al., 2016; BOSU; CARVER, 2014; LEE; CARVER, 2017; KOVALENKO; BACCHELLI, 2018; BOSU; CARVER, 2012; BOSU et al., 2014; HIRAO; IHARA; MATSUMOTO, 2015). The observed findings are scattered.

Hirao, Ihara and Matsumoto (2015) investigated how many review requests followed the simple majority method of voting to decide on the acceptance or rejection of code changes. Conducting a case study in an OSS project, the researchers aimed to understand the criteria for integrating a changeset. Their results indicate that only 59.5% of the requests followed the simple majority method and requests with more negative votes than positive votes were likely to be rejected.

Bosu and Carver (2012), based on the examination of the proportion of review requests rejected in Asterisk and MusicBrainz (OSS projects), identified that 7.5% and less than 1% of requests are not accepted, respectively. Bosu et al. (2014) presented a more extensive investigation in OSS projects, but their focus was to uncover the changes containing which types of vulnerabilities are more likely to be abandoned. Their work concluded that MCR leads to the identification of common types of vulnerabilities. Moreover, code is more likely to be vulnerable when it is authored by less experienced contributors, has a high number of lines changed, and consists of modified (as opposed to new) files.

Baysal et al. (2016) explored the influence of some factors on the outcome of a review request and found that the author development experience affects it. Three studies (BOSU; CARVER, 2014; KOVALENKO; BACCHELLI, 2018; LEE; CARVER, 2017) focused on the the author's familiarity with the project, with consistent results indicating that the acceptance rate is lower for newcomers.

4.4.2 External Outcomes

The results discussed in the previous section consists of the analysis of how different factors influence outcomes associated with the code review itself. However, these are not benefits that are externally perceived (e.g., how intense the discussion is). We discuss in this section work that has focused on external outcomes, which are mainly improve-

ment of code quality (design and programming practices) and product quality (reduction of defects).

We identified 12 papers that explored quantitative data collected in repositories, not only with code review data but also the code being reviewed. From these papers, only one (MORALES; MCINTOSH; KHOMH, 2015) focused on code quality, while the others (BOSU et al., 2014; KONONENKO et al., 2015; MENEELY et al., 2014; MCINTOSH et al., 2016; SHIMAGAKI et al., 2016; ARMSTRONG; KHOMH; ADAMS, 2017; THONGTANUNAM et al., 2015; THOMPSON; WAGNER, 2017; BAVOTA; RUSSO, 2015; MCINTOSH et al., 2014; BIASE; BRUNTINK; BACCHELLI, 2016) on product quality.

Morales, McIntosh and Khomh (2015) studied the impact of MCR on software design by examining how the incidence of seven anti-patterns is affected by the review coverage and participation. Their findings indicated that components with low coverage or low review participation are more likely to have occurrences of anti-patterns, but with variances observed across the analyzed projects.

Focusing on a different perspective, four other studies (BOSU et al., 2014; MENEELY et al., 2014; BIASE; BRUNTINK; BACCHELLI, 2016; THOMPSON; WAGNER, 2017) analyzed open-source projects to investigate factors influencing the security aspects of code that went through code review. Biase, Bruntink and Bacchelli (2016) presented a case study of security aspects of the open-source Chromium, analyzing several aspects of code review data. With respect to the MCR coverage, the researchers reported that code reviews tend mostly to miss language-specific and domain-specific issues, such as buffer overflows and Cross-Site Scripting. The other three studies then provide evidence of what might increase the likelihood of a code change to contain a security flaw after being checked in code review. As key findings, the mentioned works indicate that (i) the majority of the vulnerable code is written by the most experienced authors, although the less experienced authors' changes are 1.5 to 24 times more likely to be vulnerable (BOSU et al., 2014); (ii) more lines churned increased the probability of a patch to contain a vulnerability (BOSU et al., 2014; MENEELY et al., 2014); (iii) modified files are more likely to have vulnerabilities than new files (BOSU et al., 2014); (iv) vulnerable files tend to have more involved reviewers, with lower security-experience (MENEELY et al., 2014). Furthermore, the study of Thompson and Wagner (2017) reported that code review appears to reduce the number of issues and security issues, revealing that there is relationship between review coverage (assessed by unreviewed pull requests and

unreviewed churn), and review participation (measured by average commenters, mean discussion comments, and mean review comments).

In addition to the studies on security aspects, the remaining papers related to external outcomes are focused on the overall software quality. Kononenko et al. (2015) identified that 54%–56% of code reviews missed bugs, while Bavota and Russo (2015) indicated that unreviewed code changes have over two times more changes in inducing bugs concerning reviewed changes. They also reported that there is a difference between the quality attributes complexity and readability of code components in reviewed and unreviewed commits. Considering factors influencing the bug proneness of reviewed code changes, the likelihood of post-release defects increased as the involved reviewers have fewer reviewing experience (KONONENKO et al., 2015), and they are less familiar with the project (lack subject matter expertise) (MCINTOSH et al., 2016). Review queue also has an impact on whether reviewers catch bugs, being longer review queues more related to defect-proneness changes (KONONENKO et al., 2015). Additionally, Armstrong, Khomh and Adams (2017) observed that review using unicast communication technology has fewer defects than broadcast communication. Analyzing review practices, Thongtanunam et al. (2015) identified that future-defective files are less intensely scrutinized, having less participation of reviewers, and a faster rate of code checking than files without post-release defects. Moreover, both Thongtanunam et al. (2015) and Kononenko et al. (2015) found a relationship between a small number of reviewers and the increasing likelihood of missing issues. This finding is in contrast with the result presented by the work of Meneely et al. (2014), in which the researchers identified that vulnerable files tend to have more involved reviewers.

Finally, McIntosh et al. (2014) and Shimagaki et al. (2016) studied post-release defects and their relationship with code review coverage and review participation. While McIntosh et al. (2014) analyzed MCR practices in open-source projects, Shimagaki et al. (2016) replicated the study in a proprietary setting at Sony Mobile. Despite the slight differences in metrics for assessing review participation and coverage (metrics added in the more recent study), Shimagaki et al. (2016) reported that the relationship associated with software quality identified in the original research is not consistent with that identified in the proprietary setting. Despite the differences, both studies indicate that code review practices share a strong association with defect-proneness.

4.5 Human Aspects of Reviewers

Code review is mostly based on the subjective human evaluation of code changes and involves intensive human interactions. Therefore, there are researchers that have been exploring the *behavior* of reviewers during code review, how they check the changed code, and their willingness to participate in MCR. Seven studies went to this direction: (i) three (BEGEL; VRZAKOVA, 2018; UWANO et al., 2006; CHANDRIKA; AMUDHA; SUDARSAN, 2017) relied on eye tracking; (ii) two (FLOYD; SANTANDER; WEIMER, 2017; DURAES et al., 2016) relied on functional magnetic resonance imaging (fMRI); and (iii) the remaining two (KITAGAWA et al., 2016; DUNSMORE; ROPER; WOOD, 2000) analyzed the reviewer behavior with simulation and observation.

Studies that used eye tracking (BEGEL; VRZAKOVA, 2018; UWANO et al., 2006; CHANDRIKA; AMUDHA; SUDARSAN, 2017) aimed to understand how reviewers check the code. Uwano et al. (2006) conducted an experiment with professionals to characterize the overall performance of individuals during code review. They found that the subjects are likely to read the whole lines briefly, then concentrate on particular sections. Begel and Vrzakova (2018) are more specific and analyze the eye movements that triggers review comments, presenting a classification of five kinds of code elements that might act as a trigger. Finally, Chandrika, Amudha and Sudarsan (2017) investigated the eye-tracking trait differences of subjects with and without programming skills to understand visual attention. The authors concluded that the key aspect for MCR is attention span on error lines and comments and better code coverage. In summary, these three studies suggest that reviewers first examine all lines of changed code, then focus on specific proportions, which might be influenced by programming skills.

Exploring fMRI, two studies (FLOYD; SANTANDER; WEIMER, 2017; DURAES et al., 2016) aim to understand the brain activity of reviewers in experiments to identify patterns for analysis. One of the studies (FLOYD; SANTANDER; WEIMER, 2017) was conducted with students in an attempt to relate tasks performed by individuals with patterns of brain activation. The authors compared tasks of code review, code comprehension, and English prose review (a snippet of English writing marked up with edits) and identified distinct neural representations. They also found that a programming language is treated more like a natural language when an individual has more expertise. The other research (DURAES et al., 2016) is more focused on brain activity patterns when the reviewer identifies a bug. In this case, the researchers also found specific brain regions

where activation increased during code review, specifically the areas associated with language processing and mathematics. They showed that particular brain activity patterns can be related to the decision-making moment of suspicion/bug detection.

Finally, the remaining studies investigate two particular aspects of the review behavior. One of them (KITAGAWA et al., 2016) aim to understand the participation of reviewers in MCR using simulation. It consists of a model of a reviewing situation based on a snowdrift game. A key finding is that a reviewer cooperates with others when the benefit of a review is higher than its cost. The other (DUNSMORE; ROPER; WOOD, 2000) is an empirical investigation of defect detection in programs of the object-oriented (OO) paradigm, concluding that defects that require information spread throughout the software to be identified are hard to find and the object-oriented code structure favors this type of defect.

4.6 Relationship with MCR

The FOUNDATIONAL STUDIES discussed previously focused solely on the MCR practice. The last group of FOUNDATIONAL STUDIES consists of research that analyzed how MCR is related to other approaches within software development. We discuss identified papers in two groups. The first compares MCR with other verification techniques, and the second analyzes the impact of MCR in other practices of the software development.

4.6.1 Comparison with Verification Techniques

Considering as verification techniques the practices to software quality assurance, we found three experiments comparing MCR with pair programming (MÜLLER, 2005; SWAMIDURAI; DENNIS; KANNAN, 2014) and testing (RUNESON; ANDREWS, 2003). These studies involved only students, and the overall goal is to examine the effectiveness of one technique with respect to another.

Müller (2005) and Swamidurai, Dennis and Kannan (2014) compared code review and pair programming aiming to verify which has a more significant impact in terms of cost. In both studies, participants are divided into the ones using pair programming to execute a task, while those who work individually with the assistance of a code review phase, being the output of such tasks assessed by the researchers. The difference between the

two studies is that in Swamidurai et al.'s experiment (SWAMIDURAI; DENNIS; KANNAN, 2014) the techniques are adopted in the context of the Test-Driven Development (TDD) environment. As a key finding, Müller (2005) indicated that pairs are as cheap as single developers if both are forced to produce code of similar correctness. In contrast, when taking into account programs of different levels of correctness, pairs provide code with fewer failures at a higher expense, although the difference is not statistically significant. Therefore, this study suggests that pair programming and individual review may be interchangeable in terms of cost (MÜLLER, 2005). However, in the context of TDD, Swamidurai, Dennis and Kannan (2014) found evidence that programs with similar quality can be produced using peer review with 28% less cost than using pair programming.

While the comparisons with pair programming focused on the cost, the study that compares testing practices (RUNESON; ANDREWS, 2003) and code review focuses on the capability of detecting defects. Runeson and Andrews (2003) compared unit testing with code review, investigating the detection and isolation of the underlying sources of the defects. As a result, researchers reported differences, being code review more effective in terms of time spend and isolation, and testing finds more failures (RUNESON; ANDREWS, 2003).

Considering the discussed findings, we highlight that some of them were published more than a decade ago (from 2003 to 2014) and, since then, the MCR key goal has shifted from defect detection to problem-solving (RIGBY; BIRD, 2013) and tool-support became popular (BACCHELLI; BIRD, 2013). The expected benefits of practitioners have also been changing, as discussed. In our SLR, we did not identify more recent comparisons of MCR with other verification techniques.

4.6.2 Interaction with Development Practices

The last group of FOUNDATIONAL studies has four analyses involving MCR and its relationship with other software development practices. Two studies investigated traces of code review focusing on quality, verifying its relationship with continuous integration (RAHMAN; ROY, 2017) and static analysis (PANICHELLA et al., 2015). One study explored code review and its association with code ownership (THONGTANUNAM et al., 2016). Finally, the fourth study investigates the intent and awareness of developers when performing changes concerned with architectural aspects (PAIXAO et al., 2017). All these four works examined code review data from open-source projects.

Rahman and Roy (2017) explored the impact of continuous integration on code reviews, examining both automated builds and review comments of pull requests. Their findings indicate a relationship between both practices, identifying that passed builds and frequently built projects are more likely to encourage the reviewers' participation. While Rahman and Roy (2017) focused on a statistical analysis of build entries and review comments, Panichella et al. (2015) examined what has been changed in the code during a review to understand how the static analysis tools could have helped. By analyzing the changes during code reviews, Panichella et al. (2015) found that 6%–22% of the warnings detected by static analysis approaches are removed during code checking. The analysis also indicates a trend of developers to focus on particular kinds of problems during a review, such as imports and regular expressions. Therefore, both studies of Rahman and Roy (2017) and Panichella et al. (2015) suggest that other practices focusing on code quality might be helpful for code review, promoting participation and reducing the burden during the code checking.

Exploring a more specific topic, Paixao et al. (2017) mined review data to investigate the intent and awareness of developers of the architectural impact of their changes. The key findings of this study indicate that only 38% of the examined reviews have a conversation with respect to the architecture, which suggests a lack of awareness when such type of modification is performed in the software. However, Paixao et al. (2017) also found that developers tend to be more often aware of the architecture when the change is related to it.

The fourth study in this group is concerned with code ownership heuristics and whether code review data might complement them. Thongtanunam et al. (2016) analyzed how the code authoring and reviewing contributions differ to investigate whether review activity should be used in the code ownership heuristics. By examining data of two open-source projects, the researchers found that 67%–86% of the developers are review-only contributors, being 18%–50% of them documented as core team members. In addition, there is evidence of an increasing relationship between the proportion of reviewers who have both low traditional and review ownership values with the likelihood of having post-release defects. This suggests that the reviewing activity can be used to refine the code ownership heuristics.

5 PROPOSALS AND THEIR EVALUATION

The work previously discussed provides an understanding of how MCR works, giving insights that are helpful to propose solutions that support MCR. We now in this chapter present 46 primary studies, which consist of the PROPOSAL of a technique, tool, or theory with the goal of improving the practice. We also discuss the 28 EVALUATIONS of proposed approaches to support MCR.

5.1 Proposals to Support MCR

The 46 PROPOSALS identified in our set of primary studies were classified into three groups, as presented in Table 5.1. The first two groups are associated with the two phases of MCR described in our background chapter, namely *Review Planning and Setup* and *Code Review*. The third group, *Process Management and Support* includes approaches that focus on aiding the MCR process by means of the provision of guidance, data analysis or tool support. We next discuss approaches in these groups, highlighting their addressed MCR tasks.

5.1.1 Review Planning and Setup

We identified three kinds of support associated with the *Review Planning and Setup* phase, i.e. before the code review is performed by reviewers. The types of provided support are: (i) *patch documentation*: helping authors to complement the code change with information that is helpful to the review; (ii) *reviewer recommender*: aid in the selection of suitable reviewers; and (iii) *review prioritization*: help reviewers to select code reviews to be performed earlier. We detail approaches that aim at providing such a support next.

5.1.1.1 Patch Documentation

We identified a single approach (HAO et al., 2013) that is dedicated to support patch documentation. It consists of a tool—named Multimedia Commenting Tool (MCT)—integrated with the development environment. MCT allows programmers to include code narration and embedded multimedia resources, as well as support the replay of these com-

Table 5.1: Classification of MCR PROPOSALS.

Category	Approach Goal	# Approaches
Review Planning and Setup		**14**
Patch Documentation	Assist the preparation of the review request	1
Reviewer Recommender	Recommend or automate the selection of reviewers	11
Review Prioritization	Support the prioritization of review requests	2
Code Review		**14**
Code Checking	Support the activity of checking the code performed by reviewers	10
Feedback Provision	Support the activity of providing feedback to authors	2
Review Decision	Support deciding whether there is a need for further review of a patch	2
Process Management and Support		**17**
Grounded Theory	Provide a taxonomy or guidelines to support MCR	6
Review Retrospective	Assess or predict high-level MCR outcomes	6
Tool Support	Presentation of tools to support the MCR lifecycle	7

ments. Thus, reviewers can easily reproduce them, which might help the understanding of code changes.

5.1.1.2 Reviewer Recommender

Reviewer recommenders consist of tools and underlying techniques focused on suggesting a list of the best candidates to review a review request. The motivation of most of the techniques is to reduce of the time taken for a review acceptance. However, there are techniques whose goal is to support newcomers to reach out experienced developers (ZANJANI; KAGDI; BIRD, 2016; RAHMAN; ROY; COLLINS, 2016) or to find the best candidates when the review request involves multiple files and large changes (OUNI; KULA; INOUE, 2016; XIA et al., 2017). These recommender techniques use as input the review request and complementary data from repositories of software development. The rationale behind these algorithms is to assign a score for reviewer candidates based on attributes of the historical databases, presenting as recommendation those with higher ratings. Some techniques also consider a time prioritization factor to give more weight to current than past reviews (YU et al., 2016; JIANG et al., 2017; BALACHANDRAN, 2013; THONGTANUNAM et al., 2014; ZANJANI; KAGDI; BIRD, 2016; XIA et al., 2017; FEJZER; PRZYMUS; STENCEL, 2018).

The identified technique that was first published is called Review Bot (BALACHANDRAN, 2013), which recommends as reviewers those who worked on the same lines of the review request. Thongtanunam et al. (2014) and Thongtanunam et al. (2015) then suggested the use of *file patch similarity*, which takes into account previous changes with similar paths and who reviewed them. Thongtanunam et al. (2015) presented Rev-

Finder, which was adopted as a baseline technique in the evaluation of most of the approaches proposed posteriorly. Other techniques explored a wide range of additional features to improve code recommenders, they are: (i) the *review description* and its similarity with past descriptions; (ii) the *review feedback* written by potential reviewers (YU et al., 2016; ZANJANI; KAGDI; BIRD, 2016); and (iii) common *interests* among reviewers (OUNI; KULA; INOUE, 2016; XIA et al., 2017), and cross-project experience in specialized *technologies* (RAHMAN; ROY; COLLINS, 2016). This information complemented those previously explored. Xia et al. (2015) extended RevFinder, while other approaches (OUNI; KULA; INOUE, 2016; RAHMAN; ROY; COLLINS, 2016) included the expertise with files to be reviewed. Instead of proposing a new technique, other studies investigated the effectiveness of different features to identify the best set of reviewers. Jiang Jiang et al. (2017) compared the performance of the use of activeness, text similarity, file similarity, and social relation, concluding that activeness outperforms the others.

Finally, Fejzer, Przymus and Stencel (2018) suggested building profiles of individual programmers for recommending a reviewer. This profiles can be updated as new reviews are made, so that it is not necessary to process past information for a recommendation.

5.1.1.3 Review Prioritization

Given that reviewers might be invited to many code reviews, there is a need for prioritization. Thus, two approaches provide support by suggesting which requests should be reviewed first. Aman (2013) gives as recommendation a list of source files to be reviewed, using a 0-1 programming model-based method for planning code review using bug-fixing history. The approach estimates the effort needed to review, helping the reviewers to prioritize which files to check based on the likelihood of *bug-proneness* and *review effort*. Fan et al. (2018), instead, suggested the estimation of the *chances for a change to be accepted or rejected*, based on a prediction model using 34 features associated with code changes. The idea is to give higher priority to high-quality changes.

5.1.2 Code Review

A set of approaches focuses on helping reviewers in the manual activity of analyzing a code change and providing comments. These approaches are slit into three

categories—code checking, feedback provision and review decision—which are discussed next.

5.1.2.1 Code Checking

In order to support the manual activity of code checking, there are approaches that provide visualizations of code changes or properties associated with them. Three approaches that provide change visualization assume that a change can be decomposed into clusters. ClusterChanges (BARNETT et al., 2015) clusters diff-regions using static analysis. Instantiated for C# language, its output is the clustering and classification of partitions in trivial and non-trivial. This idea was also instantiated to the Java language as the JClusterChanges (FREIRE; BRUNET; FIGUEIREDO, 2018) approach. Lastly, Tao and Kim (2015) built a heuristic-based approach that identifies and groups two changed lines as related if (i) both are formatting-only changes; or (ii) they are semantically related for having static dependencies, or (iii) they are logically related for having similar change patterns.

Differently, there are approaches that are specific to particular kinds of change or language. ReviewFactor (GE et al., 2017) focuses on separating *refactoring* from non-refactoring changes, based on the code before and after the change and the log files of the refactoring tool usage. They take into account the automatic and manual refactorings, providing as output a visualization of the non-refactoring part and then the refactoring part. CRITICS (ZHANG et al., 2015) targets the inspection of systematic changes, allowing authors to customize a change template to summarize similar changes. This is used to detect potential mistakes. The other more specific approach consists of a differencing algorithm, named Vdiff (DULEY; SPANDIKOW; KIM, 2010), which focuses on a hardware description language, Verilog. As typical diff tools assume sequential execution semantics, they are not suitable to hardware design descriptions, and thus need alternatives to identify changes.

The approaches discussed above focus on *syntactic* aspects of code changes. The last four approaches that aim to support code checking target on the software behavior, quality, and change impact analysis, respectively. Getty (MENARINI; YAN; GRISWOLD, 2017) aims to aid code review with inter-version semantic differential analysis, presenting summaries of both code differences and *behavioral differences*, using invariants extracted from the execution of test cases. Visual Design Inspection (ViDI) (TYMCHUK; MOCCI; LANZA, 2015) gives a city-based code visualization to help reviewers

to inspect the impact of changes on the overall *quality* of the software. This visualization together with critics (broken design rules) are used by reviewers to indicate changes to be made. Similarly, MultiViewer (WANG et al., 2017) provides an assistance tool for change impact analysis. It includes the formal definition of three metrics, which are effort, risk, and impact. MultiViewer presents this information in a Spider Chart and a Coupling Chart to support reviewers to identify coupling relations among related files in the changes. Concerning effort, another study (MISHRA; SUREKA, 2014) provides an estimation model for code review. Mishra and Sureka (2014) defined six variables to measure the size and complexity of the modified files, which might help reviewers to predict the work needed to review a code change.

5.1.2.2 Feedback Provision

In addition to analyze code changes, reviewers must be able to provide feedback to authors, which can be, for example, request for changes, ask questions and clarification, or votes of acceptance or rejection. Two approaches have the goal on supporting this feedback provision. Rich Code Annotation (RCA) (PRIEST; PLIMMER, 2006) is a digital ink tool integrated with the development environment to support annotation during a review, so that reviewers can provide feedback using multimedia resources. The other approach consists of a prediction model—RevHelper (RAHMAN; ROY; KULA, 2017)—to indicate the usefulness of the review comment written by the reviewer during review submission. This model was built on top of studies (BOSU; GREILER; BIRD, 2015; RAHMAN; ROY; KULA, 2017) on the usefulness of reviews comments (one of them published in the same paper in which RevHelper was proposed). These studies were discussed in Chapter 4.

5.1.2.3 Review Decision

After going through a round of review, a code change can be accepted, rejected or may need rework, possibly requiring further reviews. In order to help reviewers make such a decision, there are two approaches that provide reviewers with complementary information that serve as indicators of whether a code change should be accepted. Both approaches have the goal to predict fault proneness. Harel and Kantorowitz (2005), more specifically, estimate the number of faults remaining in code. The method is an adaptation of an estimator, used in the formal software inspection process, to a scenario of iterative

code review, where there are multiple review iterations. Soltanifar, Erdem and Bener (2016), in turn, proposed a prediction model similar to typical bug predictors, which build a model to predict fault proneness based on a set of features. The difference of their approach is that they consider features associated with the review that has been done to predict whether a patch remains defective.

5.1.3 Process Support

The two groups of approaches previously described target specific tasks of MCR. We now detail the last group, which focuses on the MCR process as a whole and is split into three sub-groups: grounded theory, review retrospective, and tool support.

5.1.3.1 Grounded Theory

Existing works classified as grounded theory are those that, based on collected data and previous studies, propose a taxonomy or guidelines to improve MCR. Taxonomies were proposed by Baum et al. (2016a) and Li et al. (2017). The former presented a faceted classification scheme for industrial code review processes, including variations on how the process is embedded, reviewers aspects, code checking, feedback, and overarching facets, while the latter consists of a taxonomy of topics in the review feedback, with four main categories (code correctness, pull-request decision-making, project management and social interaction).

Rigby et al. (2012) discussed a series of lessons learned and recommendations based on the experience of code review in OSS that could be transferred to proprietary projects. They point out as lessons learned: (i) asynchronous, frequent and incremental reviews; (ii) invested experience reviewers; and (iii) empowerment of expert reviewers. They advocate the use of lightweight review tools and nonintrusive metrics, and the implementation of a review process. The analysis of several case studies, led Rigby and Bird (2013) to identify common best practices on the use of MCR also in OSS. These practices indicated that code review (i) is a lightweight, flexible process; (ii) happens quickly, frequently, and early; (iii) change sizes are small; (iv) usually involves two reviewers; and (v) has changed from defect identification to a group problem-solving activity, in which reviewers prefer discussion and fixing code than reporting defects. Moreover, the authors suggested that the tool-supported review provides the benefits of traceability, and

its increased adoption is an indicator of success.

In addition to these two works that proposed guidelines, there are two more specific studies. The first (BAUM; SCHNEIDER, 2016) makes recommendations associated with tool support, suggesting improvements in code review tools to increase review efficiency and effectiveness. The second study (BAUM; SCHNEIDER; BACCHELLI, 2017) is a middle-range theory to indicate an optimal order to read the code, deriving six principles that define a proper order of changes and how they should be presented for reviewers.

5.1.3.2 Review Retrospective

As MCR is performed, there is a repository with code review data that can be explored to improve the MCR process in particular projects. Six works went to this direction. Uwano et al. (2006) presented an integrated environment to measure and record the eye movements during the code review, the Crescent tool. The solution uses an eye mark tracker, associating this information with a line of the code. This data is then available for further analysis, which allows developers to examine individual performance objectively.

The other proposals explored the comments written by reviewers, presenting solutions to categorize them by means of machine learning algorithms. The studies aim to inform comment usefulness, confusion content, sentiment analysis, and identification of review topics. Pangsakulyanont et al. (2014) proposed a semantic similarity classification of comment usefulness, in which the approach computed its semantic similarity with the review request description and observing if it satisfies threshold values. Bosu, Greiler and Bird (2015) then proposed a classification based on eight comment attributes, such as the number of participants in one thread, the number of comments in the thread, and the number of iterations in the review. These attributes were defined based on the findings of a preliminary exploratory study, in which researchers interviewed developers about their perception of usefulness. As the usefulness classifiers, the approach proposed by Ebert et al. (2017) aimed to understand the content of review feedback. However, in this case, the researchers focused on the presence of confusion, grounded on the assumption that confusion negatively affects the effectiveness of code review. Based on an existing theoretical framework for categorizing expressions of confusion, eight different classifiers were trained with manually labeled the data, allowing an automatic confusion identification. Performing sentiment analysis in review comments, SentiCR (AHMED et al., 2017) is a supervised sentiment analysis tool designed explicitly for code review. It uses a sentiment oracle built empirically. Finally, Li et al. (2017) developed a two-stage

Table 5.2: Summary of the proposed approaches of tool support for MCR.

Approach	Main Goal	Tool Style
Support Code Review		
IBIS (LANUBILE; MALLARDO, 2002)	Support to distributed code inspections	Web-based tool
HyperCode (PERRY et al., 2002)	Support to distributed code inspections	Web-based tool
Nagoya, Liu and Chen (2005)	Support to the function-path review method	Desktop-based tool
Java Sniper (ZHANG et al., 2011)	Promotion of collaborative code review	Web-based tool
SmellTagger (MÜLLER et al., 2012)	Improvement in desirability and collaboration	Tool for tablet
Fistbump (KALYAN et al., 2016)	Overcoming of limitations of existing tools	Web-based tool integrated with GitHub
Support the Development of Review Tools		
Sripada, Reddy and Khandelwal (2016)	Increase in the developers' interest	Extensible framework for gamification

hybrid classification of review topics. Grounded on a taxonomy, the approach classifies the contents of review comments, allowing them to identify what reviewers are talking about. The intent, in this case, is to organize the process and optimize the review tasks.

5.1.3.3 Tool Support

MCR is supported by widely used tools, such as Gerrit and CodeFlow. However, there are different tools that have been developed with similar purpose (MÜLLER et al., 2012; KALYAN et al., 2016; NAGOYA; LIU; CHEN, 2005; LANUBILE; MALLARDO, 2002; ZHANG et al., 2011; PERRY et al., 2002; SRIPADA; REDDY; KHANDELWAL, 2016). Table 5.2 summarizes the approaches identified in our SLR (ordered by their publication date), their ultimate goal, and which strategy the researchers adopted to achieve it. We list proposals to support the code review itself, as well as to support the development of code review tools. The latter includes a single work that consists of an extensible framework to the gamification of review systems, aiming to increase developers' interest.

Older supporting tools refer to code inspections, proposing process changes to lead to a more lightweight practice. Thus, although IBIS (LANUBILE; MALLARDO, 2002) and HyperCode (PERRY et al., 2002) did not mention MCR, they can be used within a flexible and asynchronous review process. In contrast, recent proposals of tool support have been focusing on more specific goals, aiming to address specific concerns of MCR, such as SmellTagger for tablets (MÜLLER et al., 2012).

5.2 Evaluations of MCR Approaches

Having discussed the approaches proposed to support MCR tasks, we now focus on how these approaches have been evaluated either individually or compared to a baseline. We first introduce the types of evaluation that appeared either in PROPOSAL or EVALUATION studies. Then we detail design aspects of performed evaluations, followed by a discussion of their key findings.

5.2.1 Evaluations Types

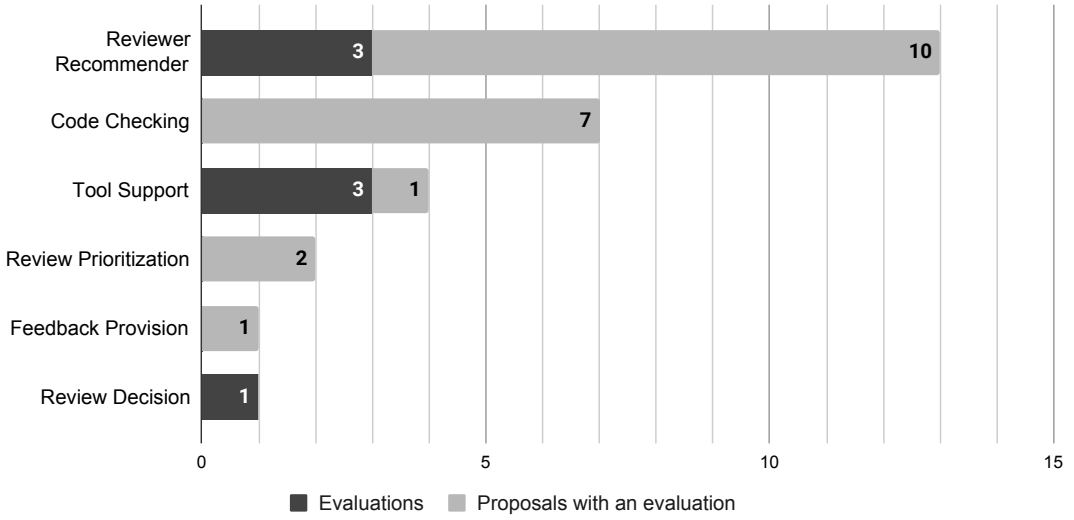
Considering our set of PROPOSALS, we investigated whether MCR approaches were evaluated in the paper that they were proposed, leading us to 21 studies. We analyzed any form of evaluation of proposed MCR solutions to support this practice. However, we did not consider as an evaluation cases in which there is solely the description of a scenario to illustrate the use or the benefits of a PROPOSAL, that is, when there is only an example of its use made by its authors; or the report of gathered informal feedback about the approach. This only provides anecdotal evidence of the effectiveness of the proposed approach. As our review identified 46 PROPOSALS, this indicates that less than a half (46%) present an evaluation, which can be considered a low number. We further identified seven papers presenting one or more EVALUATIONS of MCR approaches.

Figure 5.1 shows the number of publications containing a PROPOSAL with an accompanying evaluation and EVALUATIONS. Most of these studies aim to assess a reviewer recommender technique, which is the most common type of approach according to our SLR. Typically, a newer approach is compared to the current state-of-the-art approach to demonstrate that it supersedes the state of the art at least in one aspect.

We do not include in this section, studies comparing MCR outcomes with other techniques, e.g. pair programming, given that such studies are considered FOUNDATIONAL STUDIES, covered in Chapter 4. In addition, we also do not include studies that consist of an analysis of different sets of features (i.e. feature selection) or learning algorithm to predict MCR outcomes, e.g. Jiang et al. (2017). These studies are classified as PROPOSALS, as their contribution is an identified set of features/algorithm. The process of assessing the accuracy of different alternatives is considered part of the development of the approach.

In total, we identified 28 papers that include evaluations. Analyzing their adopted research methods and procedures, we classified them into four main groups, as described

Figure 5.1: Number of papers reporting an evaluation per MCR approach type.



Source: Author

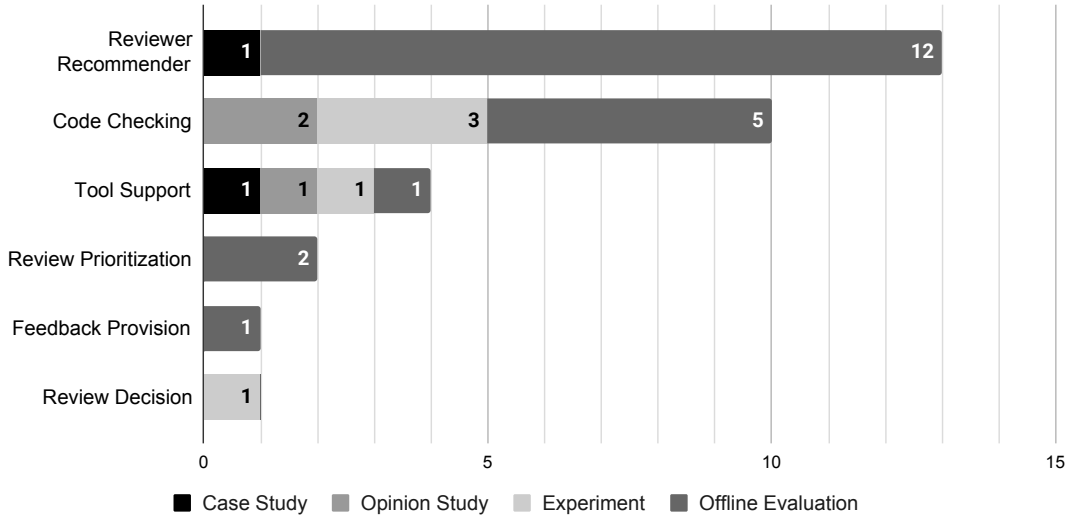
Table 5.3: Classification of MCR EVALUATIONS.

Category	**Evaluation Description**	**# Studies**
Offline Evaluation	Evaluation of an MCR approach (with or without baseline) using historical data from software projects to validate the output of the approach.	21
Experiment	Empirical study in controlled settings to observe the effects of an MCR approach.	5
Opinion Study	Subjective (qualitative or quantitative) evaluation of an MCR approach by subjects, after introducing the approach and allowing participants to experiment it.	3
Case Study	Observation and collection of data from the instantiation of the approach in real settings, possibly using mixed research methods.	2

in Table 5.3. The sum of the studies per type is higher than the number of the papers because there are papers that include more than one type of evaluation. The most common evaluation type is *offline evaluation*, being adopted in more than half (75%) of the cases. This type of evaluation refers to studies in which researchers execute a technique or tool using existing data from MCR repositories as ground truth, and also as input of the approach, when it is the case. These offline evaluations do not involve human subjects. In Figure 5.2, we further detail the adopted evaluation type by the categories of MCR approaches. As can be seen, almost all review recommenders have been evaluated offline.

Possibly due to the time and effort required to conduct user studies, evaluations involving human subjects, either professionals or students, were the choice in only a few studies. *Experiments* are the second most frequent evaluation type in our review, presented in five papers (17.86%). These studies are characterized by a controlled environment, which involves participants and measured variables to analyze the effect of an interven-

Figure 5.2: Number of studies categorized by evaluation type and MCR approach type.



Source: Author

tion in the code review process. In contrast to experiments, *opinion studies* have none or limited control, being participants invited to a hands-on trial to experiment with the proposed tool or technique. From this interaction with the proposed solution, researchers collect user perception using interviews or questionnaires. Therefore, the evaluation is based on the collected subjective data. Studies of this type are present in three (10.71%) EVALUATION papers. From these, two (BARNETT et al., 2015; ZHANG et al., 2015) were used to complement the results of another study detailed in the same publication. The third opinion study (MÜLLER et al., 2012) consists of a preliminary evaluation in which the subjects interact with a prototype of the proposed approach. Lastly, *case studies* are studies that use mixed research methods to collect and analyze data from a particular non-controlled environment. Two studies fall into this category, both based on open-source projects. In one of them, Peng et al. (PENG et al., 2018) evaluated both the usage and perception of the developers on a reviewer recommender in GitHub, collecting quantitative data and interviewing developers. In the other case study, Mizuno and Liang (MIZUNO; LIANG, 2015) assessed the evolution of Gerrit using multiple sources, such as an interview with a developer of the tool and a comparison between Gerrit and Rietveld regarding features and code review logs.

We next investigate in-depth the two most common types of evaluation, namely offline evaluations and experiments, detailing their designs and reached conclusions.

5.2.2 Offline Evaluations

We now investigate the identified offline evaluations. We first discuss their study design in terms of (i) object of the study, i.e. evaluated approach; (ii) number and type of target projects; and (iii) metrics collected for the evaluation. We then examine the conclusions reached by these studies.

5.2.2.1 Study Design

Study Object. From the 21 offline evaluations, most of them (57%) focused on reviewer recommenders. Approaches to support code checking are the second most common (25%) (BARNETT et al., 2015; TAO; KIM, 2015; DULEY; SPANDIKOW; KIM, 2010; GE et al., 2017; FREIRE; BRUNET; FIGUEIREDO, 2018). From the remaining works, two target review prioritization support (AMAN, 2013; FAN et al., 2018), and the last two studies evaluate feedback provision assistance (RAHMAN; ROY; KULA, 2017) and a variation in the MCR process (BAUM et al., 2016) .

Reviewer recommenders are usually evaluated using historical datasets, which are used to train and test generated models. The main key steps typically followed in this kind of evaluation are: (i) retrieving review requests with closed status; (ii) cleaning and sorting the collected data in chronological order; (iii) using part of the data to build a model using a learning technique; and (iv) using this model to make predictions in the test set and evaluating the results with a particular metric, such as precision, recall, and mean squared error (MSE).

Approaches to support code checking, usually by means of visualizations, are also evaluated using historical datasets. In this case, the approaches are applied to existing changesets, and through a manual inspection the correctness of the output is verified, e.g., whether generated clusters of change are acceptable. The manual inspection of the output followed two strategies, namely with and without a ground-truth. Two approaches (TAO; KIM, 2015; DULEY; SPANDIKOW; KIM, 2010) to manipulate changeset visualization were evaluated using a baseline created by human evaluators. Tao and Kim (2015) included the first author and two external students as evaluators, while Duley, Spandikow and Kim (2010) did not mention an external member in this step. In the three other studies, the output visualization was manually scrutiny by researchers without a ground-truth, using existing data of review requests as support. For example, Barnett et al. (2015) examined commit messages to verify the proposed partition of a changeset. Two of these

Table 5.4: Descriptive Statistics of the Target of Offline Evaluations and Experiments.

Evaluation Type	Study Target	# Studies	Mean	SD	Median	Min	Max
Offline Evaluation	Open-Source Projects	17	8.47	19.56	4	1	84
	Proprietary Projects	5	3.6	3.78	2	1	10
Experiment	Subjects	5	47.8	67.70	18	8	183

papers (BARNETT et al., 2015; TAO; KIM, 2015) also presented studies with subjects to complement the offline evaluation of their proposed approaches.

The remaining four offline evaluations followed different procedures. Aman (2013) produced review plans by the so called "conventional method" and the proposed method, analyzing both recommendations by the number of buggy files included in the suggested list. In contrast, Fan et al. (2018) and Rahman, Roy and Kula (2017) proposed prediction models. Consequently, their evaluations consisted of comparisons with other baselines. Moreover, Rahman, Roy and Kula (2017) manually built a ground truth to evaluate the feedback assistance approach, while Fan et al. (2018) used the stored data of a change request as a baseline for analysis. Finally, Baum et al. (2016) used a simulation model to analyze the differences between pre commit review and post commit review in terms of quality, efficiency and cycle time.

Target Projects. Offline evaluations, in our context, use data from existing software projects. We detail in Table 5.4 descriptive statistics of the number of the projects analyzed in each study and whether these projects are open source or proprietary. Two studies (RAHMAN; ROY; COLLINS, 2016; ZANJANI; KAGDI; BIRD, 2016) are taken into account in both table rows as they used data from both types of projects. One study (BAUM et al., 2016) is not considered in this table because the paper only mentions the use of data from the industry, without detailing information from which projects data was collected.

Considering project types, as said, two studies used both open-source and proprietary projects. However, the majority of the studies collected data only from open-source repositories. The most frequently used projects are Android and OpenStack with seven occurrences each, followed by Qt and LibreOffice, which were adopted in six and four evaluations, respectively. Two offline evaluations (YU et al., 2016; PENG et al., 2018) using open-source data did not inform which repositories they mined. Only three studies (BALACHANDRAN, 2013; BARNETT et al., 2015; RAHMAN; ROY; KULA, 2017) used data solely from proprietary projects.

With respect to the number of projects, most of the studies had as target only a few projects—the median is 4 and 2 projects that are open source and proprietary, respectively. As an outlier, the work of Yu et al. (2016) used data from 84 projects. Note that some of the projects are large scale and, therefore, contain a large amount of code review data, with many review requests and contributors, which can justify the low number of projects in some of the studies.

Metrics. Approaches that rely on learning techniques use the metrics that are typically used in this context. Usually, the reported metrics are accuracy, precision, and recall. Particular studies also consider F-measure (YU et al., 2016; ZANJANI; KAGDI; BIRD, 2016; FEJZER; PRZYMUS; STENCEL, 2018) or effectiveness ratio (FAN et al., 2018), for example. As reviewer recommendation can also be seen as a raking problem, another frequent evaluation metric is Mean Reciprocal Rank (MRR) (RAHMAN; ROY; COLLINS, 2016; THONGTANUNAM et al., 2015; OUNI; KULA; INOUE, 2016; XIA et al., 2015; FEJZER; PRZYMUS; STENCEL, 2018).

Approaches that have a specific purpose elaborate custom metrics: (i) Aman (2013) analyzed the number of buggy files in the generated output; (ii) Ge et al. (2017) considered the refactoring ratio detected by the approach to assessing its impact; (iii) Fejzer, Przymus and Stencel (2018) examined the memory footprint; and (iv) Baum et al. (2016) used specific heuristics for quality, efficiency and cycle time.

5.2.2.2 Findings

Offline evaluations are based on quantitative data analysis. Therefore, reached conclusions indicate how an MCR approach performs and whether it outperforms an existing solution. We discuss key findings of the these evaluations by study object as follows.

Reviewer Recommenders. Most of the papers on reviewer recommenders (12 out of 13) report results of an offline comparison between a proposed approach and selected baselines. Consequently, the main result is an evidence that indicates if the proposed approach is better than an existing one according to a selected metric. Typically, the studies consider the top-k recommendations (where k is the number of recommended reviewers).

The papers in which a new approach is proposed present an evaluation with results

Table 5.5: Results of comparisons of code reviewer recommenders.

Approach	Outperformed Reviewer Recommenders*						
	Review Bot	FPS	RevFinder	TIE	cHRev	IR+CN	Activeness
FPS (THONGTANUNAM et al., 2014)	ACC						
RevFinder (THONGTANUNAM et al., 2015)	ACC						
TIE (XIA et al., 2015)			ACC				
Correct (RAHMAN; ROY; COLLINS, 2016)			ACC, P&R, MRR				
cHRev (ZANJANI; KAGDI; BIRD, 2016)			P&R				
RevRec (OUNI; KULA; INOUE, 2016)	P&R		P&R		P&R		
WRC (HANNEBAUER et al., 2016)		ACC					
PR-CF (XIA et al., 2017)		P&R		P&R		P&R	P&R
Fejzer, Przymus and Stencel (2018)	P&R		P&R				

* Table cells indicate when an approach (rows) outperformed a baseline listed in its corresponding column, with respect to a particular metric. The metrics are accuracy (ACC), mean reciprocal rank (MRR), and precision and recall (P&R).

Table 5.6: Results of comparisons between a reviewer recomender and a baseline technique.

Approach	Measurement*	Outperformed Baselines
Review Bot (BALACHANDRAN, 2013)	ACC	RevHistRECO
IR+CN (YU et al., 2016)	P&R and F1	SVM-based, IR-based, FL-based, IR-based+CN-based, and FL-based+CN-based
cHRev (ZANJANI; KAGDI; BIRD, 2016)	P&R, F1, and MRR	xFinder and RevCom
WRC Algorithm (HANNEBAUER et al., 2016)	ACC	Line 10 Rule, Number of Changes, Expertise Recommender, Code Ownership, Expertise Cloud, and Degree-of-Authorship

* The metrics are accuracy (ACC), F-measure (F1), mean reciprocal rank (MRR), and precision and recall (P&R).

that indicate that the proposed approach is better than the selected baseline. We detail in Table 5.5 the reported results when a new approach is compared with another algorithm of reviewer recommendation, showing which approach outperformed which baseline according to which metric.

Other studies—presented in Table 5.6—compared a reviewer recommender with alternative baselines, such as a simple heuristic or a standard learning technique. For instance, Balachandran (2013) developed RevHistRECO, which is a simple heuristic inspired by the observed manual process of the reviewer assignment, to be used as a baseline for his proposed Review Bot. Yu et al. (2016), in turn, extended and implemented as baselines recommenders based on existing techniques, such as information retrieval (IR). Some of the baseline approaches in this table, e.g., xFinder (KAGDI; HAMMAD; MALETIC, 2008), were not retrieved by our SLR, because their purpose is not to recommend code reviewers and are not in the context of MCR.

Code Checking. While reviewer recommenders are usually compared with similar approaches, all of the five offline studies (BARNETT et al., 2015; TAO; KIM, 2015; DULEY; SPANDIKOW; KIM, 2010; GE et al., 2017; FREIRE; BRUNET; FIGUEIREDO,

2018) that evaluate work on support to code checking use a manually built ground truth. It is used to identify false positives and false negatives given as output of the approaches to support code checking. The identified false positives are then analyzed by the researchers, who examine their cause so that the proposed approach can be improved. For instance, Barnett et al. (2015) and Freire, Brunet and Figueiredo (2018) highlighted which type of code change was not considered by their approach to distinguish trivial and non-trivial changes.

5.2.3 Experiments

A smaller amount of MCR approaches (in comparison with offline evaluations) have been evaluated by means of experiments. They are performed in controlled environments and involve subjects, being them students, professionals, or both. We next discuss the following aspects of the design of these studies: (i) independent and dependent variables; and (ii) participants. As in the previous section, we then summarize their findings.

5.2.3.1 Study Design

Variables of the experiments. The experimental studies to evaluate an MCR approach relied on various variables. Two of these studies adopted a within-subjects design, in which all participants performed review tasks using both the proposed approach and another selected for comparison. In the experiment of Zhang et al. (2015), the analysis of systematic changes in the code was performed using the proposed CRITICS and also, as baseline, the diff and search features of Eclipse. In Tao and Kim (2015), participants reviewed a code change with and without partitions. In both cases, the researchers measured the correctness of the review task output and the time spent.

In contrast with these studies, two evaluations followed a between-subjects design. Khandelwal, Sripada and Reddy (2017) evaluated the effect of gamification on code review, organizing the participants into five groups, each of them using either a gamified or a non-gamified review tool. The researchers then measured the subject's interest by the number of review comments, the usefulness of review comments, the number of identified bugs, the number of identified code smells, and the time spent. In Menarini, Yan and Griswold (2017)'s experiment, the intervention group used the proposed Getty approach, while the control group used GitHub resources. From these interactions, the code review

process resulting from the used supporting approach has been assessed.

Finally, Runeson and Wohlin (1998) performed an experiment to compare three alternative capture-recapture methods, which are used to estimate the number of bugs in a code after going through review. Participants had to review a target code and point out bugs. Based on this, the researchers analyzed the identified bugs and inspected the errors in the estimation of the evaluated methods.

Sample size. The experiments performed to evaluate MCR approaches considered varying numbers of subjects to participate in the studies. In Table 5.4, where we show the number of projects used in offline evaluations, we also detail the descriptive statistics of the number of subjects in experiments. The study that involved the highest number of subjects (183) is that conducted by Khandelwal, Sripada and Reddy (2017), while Runeson and Wohlin (1998) experiment involved the smallest sample, with only 8 subjects. Moreover, considering the background of subjects, three of the experiments (KHANDELWAL; SRIPADA; REDDY, 2017; ZHANG et al., 2015; TAO; KIM, 2015) were conducted solely with students; the other two experiments (MENARINI; YAN; GRISWOLD, 2017; RUNESON; WOHLIN, 1998) involved professionals in addition to students.

5.2.3.2 Findings

Experiments performed in the context of MCR focused on evaluating particular aspects of the proposed approaches. Thus, the key findings are focused on the value promoted by each approach. Two experiments (ZHANG et al., 2015; TAO; KIM, 2015) provided evidence of a positive impact in the review process due to the proposed approach, such as by improving the correctness and time spent in the reviewing activity. In contrast, Khandelwal, Sripada and Reddy (2017) found no evidence that gamified tools promote a positive impact on the subjects' interest. Additionally, grounded on observations of the experiment execution, Menarini, Yan and Griswold (2017) indicated that semantically-assisted code review is feasible and effective.

In addition to the analysis of the results of experiments, three studies also include a follow-up study with the participants to collect their perceptions about the proposed approach. By means of a survey (KHANDELWAL; SRIPADA; REDDY, 2017; ZHANG et al., 2015) or an interview (MENARINI; YAN; GRISWOLD, 2017), collected subjective opinion of the participants suggested that the approaches might indeed help the code review process.

6 DISCUSSION

The reviewed literature on MCR in our SLR allowed us to identify and analyze the researched aspects of the practice, classify the existing solutions to support it, and understand how these supporting approaches have been evaluated. Thus, we presented and discussed in previous chapters a structured body of knowledge of the MCR practice, which is helpful for both researchers and practitioners. In this chapter, we discuss insights derived from our findings, highlighting lessons learned.

6.1 Foundational Studies of MCR

Modern code review is a popular approach in the context of open source development and large companies, with most of the foundational studies of MCR based on data from real settings. Our set of primary studies contains examples of the tailored adoption of the practice and one work identified what factors shape this review process in the industry. These studies provide evidence of the adoption and flexibility of MCR. However, the effect of the process variations on the effectiveness of the code review has been little explored.

Besides the variations in the MCR process, the largest part of the reviewed literature presents studies to better understand the practitioners' perceptive and the perceived outcomes of the practice. On the one hand, researchers explored the challenges and expectations among developers, which motivate the proposal of novel approaches to support authors and reviewers. On the other hand, the data stored in code review repositories have been largely explored through data mining techniques to provide an overview of the current practice. Usually, these studies focus on OSS or the industry, with few occurrences exploring both scenarios and discussing their similarities and differences.

6.1.1 MCR Dynamics and Perceived Benefits

Modern code review is usually conducted by one or two reviewers, who (in OSS projects) write an average of two or three feedback comments. The most frequent tools to support this practice are Gerrit and pull-request systems, and proprietary tools, e.g., Microsoft uses CodeFlow and Google adopts Critique. In several contexts, understanding

is a challenge faced by practitioners, especially among reviewers. For a reviewer, it is challenging to gain code familiarity and to comprehend the motivation (or purpose) of the change.

Concerning the expected benefits of practitioners, the findings indicate that code improvement, defect identification, knowledge sharing, and learning are frequent motivations to conduct code review. While the latter two have been little explored, studies have been providing evidence that the practice has a positive impact on software quality. By examing the content of the review feedback, researchers identified that code improvement is a frequent topic approached by reviewers. Futhermore, most of the code changes during a review are triggered by such comments, usually leading to evolutionary changes. Regarding defects, there is also evidence that both reviewer participation and review coverage might improve software issues. Thus, submitting code changes to be reviewed and encouraging the participation of reviewers help reduce the likelihood of post-release defects.

6.1.2 Factors that Influence the MCR Practice

From the foundational studies of MCR, we also learn about the factors influencing the internal outcomes of the practice. Usually adopting a correlational analysis, studies explored technical and non-technical factors, as well as the impact of some internal outcomes in other outcomes. Table 6.1 summarizes influence factors that have a direct or inverse relationship with internal outcomes of MCR, as discussed in Chapter 4. We represent a direct relationship with an up arrow, indicating that as the factor increases, the outcome also increases. In contrast, an inverse relationship is described with a down arrow.

6.2 Developed Approaches and their Evaluations

Solutions to support MCR are grounded on multiple purposes and adopted different strategies. Their development is usually based on existing findings from the foundational studies of MCR, aiming to help authors and reviewers do their work more effectively.

As we discussed in the previous chapter, reviewer recommenders are the most

Table 6.1: Influence on the MCR practice

Influence Factors	Reviewer Agreement Level	Reviewer Effectiveness	Reviewer Team Size	Feedback Size	Feedback Quality	Review Code Churn	Review Interations	Review Delay	Review Duration	Review Speed	Acceptance Rate
Author Code Familiarity			↓	↓				↓	↓		↑
Author Development Experience									↓		
Reviewer Code Familiarity					↑						
Reviewer Reviewing Experience	↑								↓		
Project Review Workload									↑		
Patch Code Legibility		↑									
Patch Size			↓	↓		↑	↑		↑		
Patch Prior Defects				↓		↑	↓	↑	↑	↑	
Request Description Length			↑								
Reviewer Agreement Level				↓					↓		
Reviewer Team Closeness			↑						↓		

common type of MCR approach. There are 11 proposed algorithms with this goal, exploring multiple strategies to elaborate a list of best candidates given the code change, the author, and historical information. The main purpose is to reduce time to find suitable reviewers and support decision-making, although some studies also focused on aiding newcomers, for example. Nevertheless, the adoption of such recommenders and their impact on the internal outcomes of MCR in real settings has been little explored.

Another common goal of MCR supporting approaches is to help reviewers' understanding, in particular when they perform code checking. Thus, the existing approaches explored the change documentation and visualization to increase and facilitate the gained of awareness about the artifact under review. In this case, there is evidence that a reviewer with more familiarity provides more useful review feedback, as well as the duration of a review is shorter. Despite the effort to show the feasibility of such proposals, there is also a lack of evidence about their adoption in real settings and their impact on code review outcomes.

7 CONCLUSION AND FUTURE WORK

Code review is a well-known practice of quality assurance in software development that evolved from a structured and rigid form (i.e. software inspection) to a flexible, tool-based, and asynchronous process, namely modern code review (MCR). As MCR gained increasing popularity in recent years, the practice has been largely investigated in academia. Therefore, to have a comprehensive view of what has been done in this field, we presented in this thesis the results of a systematic literature review on MCR, which includes 110 primary studies.

We identified three main categories of studies, namely foundational studies, proposed novel approaches to support the practice, and empirical studies to evaluate existing MCR work. Each paper category was systematically analyzed observing aspects relevant for each type of work. Most of the investigated work consists of foundational studies that have been conducted to better understand the motivations for the adoption of MCR, its challenges and benefits, and analysis of which influence factors lead to which MCR outcomes. From the novel approaches to support MCR, the most common is code reviewer recommenders. Evaluations of MCR approaches have been done mostly offline and few studies involving human subjects have been conducted. The analysis of the existing literature on MCR allowed us to identify concerns that remain unaddressed in this context, which are discussed as follows.

- **MCR process improvement.** Empirical studies on MCR demonstrate the feasibility of extracting information from the history of review activity stored in tools, such as Gerrit and CodeFlow. These studies provide findings useful for practitioners and researchers, such as that the submission of small patches leads to better outcomes in MCR. However, how developers can use this data to improve the MCR process in particular projects is an unexplored question. Moreover, this information can also be used to support strategic decision making to improve this process.
- **Code Improvement.** Differently from inspections, MCR has as one of its main benefits source code improvement. Nevertheless, changes based on comments are made in an individual basis. Consequently, the same comments might be made in different code reviews. Natural language processing (NLP) techniques can be used to extract recurrent bad practices in particular projects based on comments to avoid them to occur again by means of knowledge dissemination.
- **Exploration of Non-technical MCR Benefits.** Differently from code inspections,

whose primary focus was bug detection, MCR brings various other benefits, such as knowledge transfer, collective code ownership, and learning. However, these non-technical benefits have been little explored in foundational studies and existing approaches do not focus on improving them.

- **User studies on MCR.** Only few experiments involving human subjects have been conducted in the context of MCR. MCR is essentially a human-based activity, supported by tools. Consequently, further user studies must be done. Most of the evaluations of code reviewer recommenders rely only on accuracy metrics. However, as known in the recommender systems research area, other aspects, such as novelty and transparency are key for the adoption of recommenders.

We performed a *systematic* literature review in order to reduce the bias in the selection and analysis of a large number of primary studies on MCR. Although this search is large, it is not complete and, therefore, our review may have left out other existing studies on MCR. In order to mitigate the limitations of SLRs, we selected widely used digital libraries as sources, assuming that they would contain the largest number of relevant studies. In addition, to identify further studies, a snowballing approach can be used. This approach has not been followed in the present work because a preliminary analysis of obtained results indicated that the most relevant studies have been retrieved by searching our selected digital libraries. Future SLRs on MCR may target other digital libraries or gray literature as well as cover future years given that the research on MCR has been active to a great extent in the recent years.

In summary, this thesis consists of an overview of the current state-of-the-art research on MCR with an in-depth analysis of researched aspects in this context. The survey can be used by developers to learn aspects of MCR, so that they can improve their practice, and by researchers and practitioners to have a solid foundation on this topic for developing future approaches, evaluations, and foundational studies. Therefore, our work consists of a significant step towards understanding the MCR knowledge body.

REFERENCES

AHMED, T. et al. Senticr: A customized sentiment analysis tool for code review interactions. In: **Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering**. [S.l.]: IEEE Press, 2017. (ASE 2017), p. 106–111. ISBN 9781538626849.

ALBAYRAK, O.; DAVENPORT, D. Impact of maintainability defects on code inspections. In: **Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement**. [S.l.: s.n.], 2010. (ESEM '10), p. 50:1–50:4. ISBN 978-1-4503-0039-1.

AMAN, H. 0-1 programming model-based method for planning code review using bug fix history. **2013 20th Asia-Pacific Software Engineering Conference (APSEC)**, v. 2, p. 37–42, 2013.

ARMSTRONG, F.; KHOMH, F.; ADAMS, B. Broadcast vs. unicast review technology: Does it matter? In: **2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)**. [S.l.]: IEEE, 2017. p. 219–229.

ASUNDI, J.; JAYANT, R. Patch review processes in open source software development communities: A comparative case study. In: . USA: IEEE Computer Society, 2007. ISBN 0769527558.

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**. [S.l.]: IEEE Press, 2013. (ICSE'13), p. 712–721. ISBN 9781467330763.

BADAMPUDI, D.; BRITTO, R.; UNTERKALMSTEINER, M. Modern code reviews - preliminary results of a systematic mapping study. In: **Proceedings of the Evaluation and Assessment on Software Engineering**. [S.l.: s.n.], 2019. (EASE '19), p. 340–345. ISBN 978-1-4503-7145-2.

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**. [S.l.]: IEEE Press, 2013. (ICSE '13), p. 931–940. ISBN 9781467330763.

BARNETT, M. et al. Helping developers help themselves: Automatic decomposition of code review changesets. In: **Proceedings of the 37th International Conference on Software Engineering - Volume 1**. [S.l.]: IEEE Press, 2015. (ICSE '15), p. 134–144. ISBN 9781479919345.

BAUM, T. et al. Comparing pre commit reviews and post commit reviews using process simulation. In: **Proceedings of the International Conference on Software and Systems Process**. [S.l.: s.n.], 2016. (ICSSP '16), p. 26–35. ISBN 978-1-4503-4188-2.

BAUM, T.; LESSMANN, H.; SCHNEIDER, K. The choice of code review process: A survey on the state of the practice. In: FELDERER, M. et al. (Ed.). **Product-Focused Software Process Improvement**. Cham: Springer International Publishing, 2017. p. 111–127. ISBN 978-3-319-69926-4.

BAUM, T. et al. A faceted classification scheme for change-based industrial code review processes. In: **2016 IEEE International Conference on Software Quality, Reliability and Security (QRS)**. [S.l.: s.n.], 2016. p. 74–85.

BAUM, T. et al. Factors influencing code review processes in industry. In: **Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering**. [S.l.: s.n.], 2016. (FSE 2016), p. 85–96. ISBN 978-1-4503-4218-6.

BAUM, T.; SCHNEIDER, K. On the need for a new generation of code review tools. In: ABRAHAMSSON, P. et al. (Ed.). **Product-Focused Software Process Improvement**. Cham: Springer International Publishing, 2016. p. 301–308. ISBN 978-3-319-49094-6.

BAUM, T.; SCHNEIDER, K.; BACCHELLI, A. On the optimal order of reading source code changes for review. In: **2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)**. Shanghai, China: IEEE, 2017. p. 329–340.

BAVOTA, G.; RUSSO, B. Four eyes are better than two: On the impact of code reviews on software quality. In: **2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)**. [S.l.]: IEEE, 2015. p. 81–90.

BAYSAL, O. et al. The secret life of patches: A firefox case study. In: **2012 19th Working Conference on Reverse Engineering**. [S.l.]: IEEE Computer Society, 2012. p. 447–455. ISSN 2375-5369.

BAYSAL, O. et al. Investigating technical and non-technical factors influencing modern code review. **Empirical Software Engineering**, v. 21, n. 3, p. 932–959, Jun 2016. ISSN 1573-7616.

BEGEL, A.; VRZAKOVA, H. Eye movements in code review. In: **Proceedings of the Workshop on Eye Movements in Programming**. [S.l.: s.n.], 2018. (EMIP '18), p. 5:1–5:5. ISBN 978-1-4503-5792-0.

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**. [S.l.: s.n.], 2014. (MSR 2014), p. 202–211. ISBN 978-1-4503-2863-0.

BERNHART, M.; GRECHENIG, T. On the understanding of programs with continuous code reviews. In: **2013 21st International Conference on Program Comprehension (ICPC)**. [S.l.: s.n.], 2013. p. 192–198. ISSN 1092-8138.

BIASE, M. di; BRUNTINK, M.; BACCHELLI, A. A security perspective on code review: The case of chromium. In: **2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)**. [S.l.]: IEEE, 2016. p. 21–30. ISSN 2470-6892.

BIRD, C.; CARNAHAN, T.; GREILER, M. Lessons learned from building and deploying a code review analytics platform. In: **2015 IEEE/ACM 12th Working**

Conference on Mining Software Repositories. [S.l.]: IEEE Press, 2015. (MSR '15), p. 191–201. ISSN 2160-1852.

BOSU, A.; CARVER, J. C. Peer code review in open source communities using reviewboard. In: **Proceedings of the ACM 4th Annual Workshop on Evaluation and Usability of Programming Languages and Tools**. [S.l.: s.n.], 2012. (PLATEAU '12), p. 17–24. ISBN 978-1-4503-1631-6.

BOSU, A.; CARVER, J. C. Impact of peer code review on peer impression formation: A survey. In: **2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement**. [S.l.]: IEEE, 2013. p. 133–142. ISSN 1949-3770.

BOSU, A.; CARVER, J. C. Impact of developer reputation on code review outcomes in oss projects: An empirical investigation. In: **Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement**. [S.l.: s.n.], 2014. (ESEM '14), p. 33:1–33:10. ISBN 978-1-4503-2774-9.

BOSU, A. et al. Process aspects and social dynamics of contemporary code review: Insights from open source development and industrial practice at microsoft. **IEEE Transactions on Software Engineering**, v. 43, n. 1, p. 56–75, Jan 2017. ISSN 0098-5589.

BOSU, A. et al. Identifying the characteristics of vulnerable code changes: An empirical study. In: **Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering**. [S.l.: s.n.], 2014. (FSE 2014), p. 257–268. ISBN 978-1-4503-3056-5.

BOSU, A.; GREILER, M.; BIRD, C. Characteristics of useful code reviews: An empirical study at microsoft. In: **Proceedings of the 12th Working Conference on Mining Software Repositories**. [S.l.: s.n.], 2015. (MSR '15), p. 146–156. ISBN 978-0-7695-5594-2.

BRYKCZYNSKI, B. A survey of software inspection checklists. **SIGSOFT Softw. Eng. Notes**, v. 24, n. 1, p. 82–, jan. 1999. ISSN 0163-5948.

CHANDRIKA, K. R.; AMUDHA, J.; SUDARSAN, S. D. Recognizing eye tracking traits for source code review. In: **2017 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)**. [S.l.]: IEEE, 2017. p. 1–8. ISSN 1946-0759.

COHEN, J. Modern code review. **Making Software: What Really Works, and Why We Believe It**, " O'Reilly Media, Inc.", p. 329–336, 2010.

DULEY, A.; SPANDIKOW, C.; KIM, M. A program differencing algorithm for verilog hdl. In: **Proceedings of the IEEE/ACM International Conference on Automated Software Engineering**. [S.l.: s.n.], 2010. (ASE '10), p. 477–486. ISBN 978-1-4503-0116-9.

DUNSMORE, A.; ROPER, M.; WOOD, M. Object-oriented inspection in the face of delocalisation. In: **Proceedings of the 22Nd International Conference on Software Engineering**. [S.l.: s.n.], 2000. (ICSE '00), p. 467–476. ISBN 1-58113-206-9.

DURAES, J. et al. Wap: Understanding the brain at software debugging. In: **2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)**. [S.l.]: IEEE, 2016. p. 87–92. ISSN 2332-6549.

EBERT, F. et al. Confusion detection in code reviews. In: **2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)**. [S.l.]: IEEE, 2017. p. 549–553.

EFSTATHIOU, V.; SPINELLIS, D. Code review comments: Language matters. In: **Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results**. [S.l.: s.n.], 2018. (ICSE-NIER '18), p. 69–72. ISBN 978-1-4503-5662-6.

FAGAN, M. E. Design and code inspections to reduce errors in program development. **IBM Syst. J.**, v. 15, n. 3, p. 182–211, sep. 1976. ISSN 0018-8670.

FAN, Y. et al. Early prediction of merged code changes to prioritize reviewing tasks. **Empirical Software Engineering**, v. 23, n. 6, p. 3346–3393, Dec 2018. ISSN 1573-7616.

FEJZER, M.; PRZYMUS, P.; STENCEL, K. Profile based recommendation of code reviewers. **Journal of Intelligent Information Systems**, v. 50, n. 3, p. 597–619, Jun 2018. ISSN 1573-7675.

FLOYD, B.; SANTANDER, T.; WEIMER, W. Decoding the representation of code in the brain: An fmri study of code review and expertise. In: **Proceedings of the 39th International Conference on Software Engineering**. [S.l.: s.n.], 2017. (ICSE '17), p. 175–186. ISBN 978-1-5386-3868-2.

FREIRE, V. d. C. L.; BRUNET, J.; FIGUEIREDO, J. C. A. de. Automatic decomposition of java open source pull requests: A replication study. In: TJOA, A. M. et al. (Ed.). **SOFSEM 2018: Theory and Practice of Computer Science**. Cham: Springer International Publishing, 2018. p. 255–268. ISBN 978-3-319-73117-9.

FRONZA, I. et al. Code reviews, software inspections, and code walkthroughs: Systematic mapping study of research topics. In: WINKLER, D. et al. (Ed.). **Software Quality: Quality Intelligence in Software and Systems Engineering**. Cham: Springer International Publishing, 2020. p. 121–133. ISBN 978-3-030-35510-4.

GE, X. et al. Refactoring-aware code review. In: **2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)**. [S.l.]: IEEE, 2017. p. 71–79. ISSN 1943-6106.

GERMAN, D. M. et al. "was my contribution fairly reviewed?": A framework to study the perception of fairness in modern code reviews. In: **Proceedings of the 40th International Conference on Software Engineering**. [S.l.: s.n.], 2018. (ICSE '18), p. 523–534. ISBN 978-1-4503-5638-1.

GIBBS, G. R. Thematic coding and categorizing. In: **Analyzing Qualitative Data**. London: SAGE Publications Ltd., 2007.

HALLING, M. et al. Using reading techniques to focus inspection performance. In: **Proceedings 27th EUROMICRO Conference. 2001: A Net Odyssey**. [S.l.: s.n.], 2001. p. 248–257.

HANNEBAUER, C. et al. Automatically recommending code reviewers based on their expertise: An empirical comparison. In: **Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering**. [S.l.: s.n.], 2016. (ASE 2016), p. 99–110. ISBN 978-1-4503-3845-5.

HAO, Y. et al. Mct: A tool for commenting programs by multimedia comments. In: **Proceedings of the 2013 International Conference on Software Engineering**. [S.l.: s.n.], 2013. (ICSE '13), p. 1339–1342. ISBN 978-1-4673-3076-3.

HAREL, A.; KANTOROWITZ, E. Estimating the number of faults remaining in software code documents inspected with iterative code reviews. In: **IEEE International Conference on Software - Science, Technology Engineering (SwSTE'05)**. [S.l.]: IEEE Computer Society, 2005. p. 151–160.

HERNANDES, E. M.; BELGAMO, A.; FABBRI, S. An overview of experimental studies on software inspection process. In: HAMMOUDI, S. et al. (Ed.). **Enterprise Information Systems**. Cham: Springer International Publishing, 2014. p. 118–134. ISBN 978-3-319-09492-2.

HIRAO, T.; IHARA, A.; MATSUMOTO, K.-i. Pilot study of collective decision-making in the code review process. In: **Proceedings of the 25th Annual International Conference on Computer Science and Software Engineering**. [S.l.: s.n.], 2015. (CASCON '15), p. 248–251.

HIRAO, T. et al. The impact of a low level of agreement among reviewers in a code review process. In: CROWSTON, K. et al. (Ed.). **Open Source Systems: Integrating Communities**. Cham: Springer International Publishing, 2016. p. 97–110. ISBN 978-3-319-39225-7.

HUNDHAUSEN, C. D.; AGARWAL, P.; TREVISAN, M. Online vs. face-to-face pedagogical code reviews: An empirical comparison. In: **Proceedings of the 42Nd ACM Technical Symposium on Computer Science Education**. [S.l.: s.n.], 2011. (SIGCSE '11), p. 117–122. ISBN 978-1-4503-0500-6.

IZQUIERDO-CORTAZAR, D. et al. Using metrics to track code review performance. In: **Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering**. [S.l.: s.n.], 2017. (EASE'17), p. 214–223. ISBN 978-1-4503-4804-1.

JIANG, J. et al. Who should comment on this pull request? analyzing attributes for more accurate commenter recommendation in pull-based development. **Information and Software Technology**, v. 84, p. 48 – 62, 2017. ISSN 0950-5849.

KAGDI, H.; HAMMAD, M.; MALETIC, J. I. Who can help me with this source code change? In: **2008 IEEE International Conference on Software Maintenance**. [S.l.: s.n.], 2008. p. 157–166. ISSN 1063-6773.

KALYAN, A. et al. A collaborative code review platform for github. In: **2016 21st International Conference on Engineering of Complex Computer Systems (ICECCS)**. [S.l.]: IEEE, 2016. p. 191–196.

KASSE, T. Reviews and testing. In: **Practical Insight into CMMI**. Norwood: Artech House, 2008. chp. 13, p. 223–247.

KAWAHARA, A. E. B. . H. A. . M. Examination of coding violations focusing on their change patterns over releases. In: **2016 23rd Asia-Pacific Software Engineering Conference (APSEC)**. Hamilton, New Zealand: IEEE, 2016. p. 121–128. ISSN 1530-1362.

KERZAZI, N.; ASRI, I. E. Who can help to review this piece of code? In: AFSARMANESH, H.; CAMARINHA-MATOS, L. M.; SOARES, A. L. (Ed.). **Collaboration in a Hyperconnected World**. Cham: Springer International Publishing, 2016. p. 289–301. ISBN 978-3-319-45390-3.

KHANDELWAL, S.; SRIPADA, S. K.; REDDY, Y. R. Impact of gamification on code review process: An experimental study. In: **Proceedings of the 10th Innovations in Software Engineering Conference**. [S.l.: s.n.], 2017. (ISEC '17), p. 122–126. ISBN 978-1-4503-4856-0.

KITAGAWA, N. et al. Code review participation: Game theoretical modeling of reviewers in gerrit datasets. In: **Proceedings of the 9th International Workshop on Cooperative and Human Aspects of Software Engineering**. [S.l.: s.n.], 2016. (CHASE '16), p. 64–67. ISBN 978-1-4503-4155-4.

KITCHENHAM, B.; CHARTERS, S. **Guidelines for performing systematic literature reviews in software engineering**. [S.l.], 2007.

KOLLANUS, S.; KOSKINEN, J. Survey of software inspection research: 1991-2005. 2007.

KONONENKO, O.; BAYSAL, O.; GODFREY, M. W. Code review quality: How developers see it. In: **Proceedings of the 38th International Conference on Software Engineering**. [S.l.: s.n.], 2016. (ICSE '16), p. 1028–1038. ISBN 978-1-4503-3900-1.

KONONENKO, O. et al. Investigating code review quality: Do people and participation matter? In: **2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)**. [S.l.]: IEEE Computer Society, 2015. p. 111–120.

KOVALENKO, V.; BACCHELLI, A. Code review for newcomers: Is it different? In: **Proceedings of the 11th International Workshop on Cooperative and Human Aspects of Software Engineering**. [S.l.: s.n.], 2018. (CHASE '18), p. 29–32. ISBN 978-1-4503-5725-8.

LAITENBERGER, O.; DEBAUD, J.-M. An encompassing life cycle centric survey of software inspection. **J. Syst. Softw.**, v. 50, n. 1, p. 5–31, jan. 2000. ISSN 0164-1212.

LANUBILE, F.; MALLARDO, T. Tool support for distributed inspection. In: **Proceedings of the 26th International Computer Software and Applications Conference on Prolonging Software Life: Development and Redevelopment**. [S.l.: s.n.], 2002. (COMPSAC '02), p. 1071–1076. ISBN 0-7695-1727-7.

LEE, A.; CARVER, J. C. Are one-time contributors different?: A comparison to core and periphery developers in floss repositories. In: **2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)**. [S.l.]: IEEE Press, 2017. p. 1–10. ISBN 978-1-5090-4039-1.

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.

LIANG, J.; MIZUNO, O. Analyzing involvements of reviewers through mining a code review repository. In: **2011 Joint Conference of the 21st International Workshop on Software Measurement and the 6th International Conference on Software Process and Product Measurement**. [S.l.]: IEEE, 2011. p. 126–132.

MACDONALD, F.; MILLER, J. A comparison of computer support systems for software inspection. **Automated Software Engineering**, v. 6, n. 3, p. 291–313, Jul 1999. ISSN 1573-7535.

MACDONALD, F. et al. A review of tool support for software inspection. In: **Proceedings Seventh International Workshop on Computer-Aided Software Engineering**. Toronto, Ontario, Canada: IEEE, 1995. p. 340–349.

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 0740-7459.

MASHAYEKHI, V. et al. Distributed, collaborative software inspection. **IEEE Software**, v. 10, n. 5, p. 66–75, Sep. 1993.

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**. [S.l.: s.n.], 2014. (MSR 2014), p. 192–201. ISBN 978-1-4503-2863-0.

MCINTOSH, S. et al. An empirical study of the impact of modern code review practices on software quality. **Empirical Software Engineering**, v. 21, n. 5, p. 2146–2189, Oct 2016. ISSN 1573-7616.

MENARINI, M.; YAN, Y.; GRISWOLD, W. G. Semantics-assisted code review: An efficient tool chain and a user study. In: **2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)**. [S.l.]: IEEE Press, 2017. p. 554–565.

MENEELY, A. et al. An empirical investigation of socio-technical code review metrics and security vulnerabilities. In: **Proceedings of the 6th International Workshop on Social Software Engineering**. [S.l.: s.n.], 2014. (SSE 2014), p. 37–44. ISBN 978-1-4503-3227-9.

MILLER, J.; YIN, Z. Adding diversity to software inspections. In: **The Second IEEE International Conference on Cognitive Informatics, 2003. Proceedings.** London, UK: IEEE, 2003. p. 81–86.

MISHRA, R.; SUREKA, A. Mining peer code review system for computing effort and contribution metrics for patch reviewers. In: **2014 IEEE 4th Workshop on Mining Unstructured Data**. [S.l.]: IEEE Computer Society, 2014. p. 11–15.

MIZUNO, O.; LIANG, J. Does a code review tool evolve as the developer intended? In: ____. **Software Engineering Research, Management and Applications**. Cham: Springer International Publishing, 2015. p. 59–74. ISBN 978-3-319-11265-7.

MORALES, R.; MCINTOSH, S.; KHOMH, F. Do code review practices impact design quality? a case study of the qt, vtk, and itk projects. In: **2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)**. [S.l.]: IEEE Press, 2015. p. 171–180. ISSN 1534-5351.

MÜLLER, M. M. Two controlled experiments concerning the comparison of pair programming to peer review. **Journal of Systems and Software**, v. 78, n. 2, p. 166 – 179, 2005. ISSN 0164-1212.

MÜLLER, S. et al. An approach for collaborative code reviews using multi-touch technology. In: **2012 5th International Workshop on Co-operative and Human Aspects of Software Engineering (CHASE)**. [S.l.]: IEEE Press, 2012. p. 93–99.

MURAKAMI, Y.; TSUNODA, M.; UWANO, H. Wap: Does reviewer age affect code review performance? In: **2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)**. [S.l.]: IEEE, 2017. p. 164–169. ISSN 2332-6549.

NAGOYA, F.; LIU, S.; CHEN, Y. A tool and case study for specification-based program review. In: **29th Annual International Computer Software and Applications Conference (COMPSAC'05)**. [S.l.]: IEEE Press, 2005. v. 1, p. 375–380 Vol. 2. ISSN 0730-3157.

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

PAIXAO, M. et al. Are developers aware of the architectural impact of their changes? In: **Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering**. [S.l.: s.n.], 2017. (ASE 2017), p. 95–105. ISBN 978-1-5386-2684-9.

PANGSAKULYANONT, T. et al. Assessing mcr discussion usefulness using semantic similarity. In: **2014 6th International Workshop on Empirical Software Engineering in Practice**. [S.l.: s.n.], 2014. p. 49–54.

PANICHELLA, S. et al. Would static analysis tools help developers with code reviews? In: **2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)**. [S.l.]: IEEE, 2015. p. 161–170. ISSN 1534-5351.

PENG, Z. et al. Exploring how software developers work with mention bot in github. In: **Proceedings of the Sixth International Symposium of Chinese CHI**. [S.l.: s.n.], 2018. (ChineseCHI '18), p. 152–155. ISBN 978-1-4503-6508-6.

PERRY, D. E. et al. Reducing inspection interval in large-scale software development. **IEEE Transactions on Software Engineering**, v. 28, n. 7, p. 695–705, July 2002. ISSN 0098-5589.

PETERSEN, A.; ZINGARO, D. Code reviews in large, first-year courses. In: **Proceedings of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education**. [S.l.: s.n.], 2018. (ITiCSE 2018), p. 354–355. ISBN 978-1-4503-5707-4.

PETERSSON, H. et al. Capture-recapture in software inspections after 10 years research–theory, evaluation and application. **Journal of Systems and Software**, v. 72, n. 2, p. 249 – 264, 2004. ISSN 0164-1212.

PRIEST, R.; PLIMMER, B. Rca: Experiences with an ide annotation tool. In: **Proceedings of the 7th ACM SIGCHI New Zealand Chapter's International Conference on Computer-human Interaction: Design Centered HCI**. [S.l.: s.n.], 2006. (CHINZ '06), p. 53–60. ISBN 1-59593-473-1.

RAHMAN, M. M.; ROY, C. K. Impact of continuous integration on code reviews. In: **Proceedings of the 14th International Conference on Mining Software Repositories**. [S.l.: s.n.], 2017. (MSR '17), p. 499–502. ISBN 978-1-5386-1544-7.

RAHMAN, M. M.; ROY, C. K.; COLLINS, J. A. Correct: Code reviewer recommendation in github based on cross-project and technology experience. In: **2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)**. [S.l.]: IEEE Press, 2016. p. 222–231.

RAHMAN, M. M.; ROY, C. K.; KULA, R. G. Predicting usefulness of code review comments using textual features and developer experience. In: **2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)**. [S.l.]: IEEE Press, 2017. p. 215–226.

RIGBY, P.; BIRD, C. Convergent contemporary software peer review practices. In: **Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering**. [S.l.: s.n.], 2013. (ESEC/FSE 2013), p. 202–212. ISBN 978-1-4503-2237-9.

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

RUNESON, P.; ANDREWS, A. Detection or isolation of defects? an experimental comparison of unit testing and code inspection. In: **14th International Symposium on Software Reliability Engineering, 2003. ISSRE 2003.** [S.l.]: IEEE Computer Society, 2003. p. 3–13. ISSN 1071-9458.

RUNESON, P.; WOHLIN, C. An experimental evaluation of an experience-based capture-recapture method in software code inspections. **Empirical Software Engineering**, v. 3, n. 4, p. 381–406, Dec 1998. ISSN 1573-7616.

SADOWSKI, C. et al. Modern code review: A case study at google. In: **Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice**. [S.l.: s.n.], 2018. (ICSE-SEIP '18), p. 181–190. ISBN 978-1-4503-5659-6.

SANTOS, E. W. dos; NUNES, I. Investigating the effectiveness of peer code review in distributed software development. In: **Proceedings of the 31st Brazilian Symposium on Software Engineering**. [S.l.: s.n.], 2017. (SBES'17), p. 84–93. ISBN 978-1-4503-5326-7.

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**. [S.l.: s.n.], 2016. (ICSE '16), p. 212–221. ISBN 978-1-4503-4205-6.

SHULL, F.; SEAMAN, C. Inspecting the history of inspections: An example of evidence-based technology diffusion. **IEEE Software**, v. 25, n. 1, p. 88–90, Jan 2008. ISSN 0740-7459.

SINGH, D. et al. Evaluating how static analysis tools can reduce code review effort. In: **2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)**. [S.l.]: IEEE, 2017. p. 101–105. ISSN 1943-6106.

SOLTANIFAR, B.; ERDEM, A.; BENER, A. Predicting defectiveness of software patches. In: **Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement**. [S.l.: s.n.], 2016. (ESEM '16), p. 22:1–22:10. ISBN 978-1-4503-4427-2.

SPADINI, D. et al. When testing meets code review: Why and how developers review tests. In: **Proceedings of the 40th International Conference on Software Engineering**. [S.l.: s.n.], 2018. (ICSE '18), p. 677–687. ISBN 978-1-4503-5638-1.

SRIPADA, S. K.; REDDY, Y. R.; KHANDELWAL, S. Architecting an extensible framework for gamifying software engineering concepts. In: **Proceedings of the 9th India Software Engineering Conference**. [S.l.: s.n.], 2016. (ISEC '16), p. 119–130. ISBN 978-1-4503-4018-2.

SUTHERLAND, A.; VENOLIA, G. Can peer code reviews be exploited for later information needs? In: **2009 31st International Conference on Software Engineering - Companion Volume**. [S.l.]: IEEE, 2009. p. 259–262.

SWAMIDURAI, R.; DENNIS, B.; KANNAN, U. Investigating the impact of peer code review and pair programming on test-driven development. In: **IEEE SOUTHEASTCON 2014**. [S.l.]: IEEE, 2014. p. 1–5. ISSN 1558-058X.

TAO, Y.; KIM, S. Partitioning composite code changes to facilitate code review. In: **2015 IEEE/ACM 12th Working Conference on Mining Software Repositories**. [S.l.]: IEEE Press, 2015. (MSR '15), p. 180–190. ISSN 2160-1852.

THELIN, T. et al. Evaluation of usage-based reading—conclusions after three experiments. **Empirical Software Engineering**, v. 9, n. 1, p. 77–110, Mar 2004.

THOMPSON, C.; WAGNER, D. A large-scale study of modern code review and security in open source projects. In: **Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering**. [S.l.: s.n.], 2017. (PROMISE), p. 83–92. ISBN 978-1-4503-5305-2.

THONGTANUNAM, P. et al. Improving code review effectiveness through reviewer recommendations. In: **Proceedings of the 7th International Workshop on Cooperative and Human Aspects of Software Engineering**. [S.l.: s.n.], 2014. (CHASE 2014), p. 119–122. ISBN 978-1-4503-2860-9.

THONGTANUNAM, P. et al. Investigating code review practices in defective files: An empirical study of the qt system. In: **2015 IEEE/ACM 12th Working Conference on Mining Software Repositories**. [S.l.]: IEEE Press, 2015. (MSR '15), p. 168–179. ISBN 978-0-7695-5594-2.

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**. [S.l.: s.n.], 2016. (ICSE '16), p. 1039–1050. ISBN 978-1-4503-3900-1.

THONGTANUNAM, P. et al. Review participation in modern code review. **Empirical Software Engineering**, v. 22, n. 2, p. 768–817, Apr 2017. ISSN 1573-7616.

THONGTANUNAM, P. et al. Who should review my code? a file location-based code-reviewer recommendation approach for modern code review. In: **2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)**. [S.l.]: IEEE, 2015. p. 141–150. ISSN 1534-5351.

TYMCHUK, Y.; MOCCI, A.; LANZA, M. Code review: Veni, vidi, vici. In: **2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)**. [S.l.]: IEEE, 2015. p. 151–160. ISSN 1534-5351.

UEDA, Y. et al. How is if statement fixed through code review? a case study of qt project. In: **2017 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)**. [S.l.]: IEEE Press, 2017. p. 207–213.

UWANO, H. et al. Analyzing individual performance of source code review using reviewers' eye movement. In: **Proceedings of the 2006 Symposium on Eye Tracking Research &Amp; Applications**. [S.l.: s.n.], 2006. (ETRA '06), p. 133–140. ISBN 1-59593-305-0.

VOTTA JR., L. G. Does every inspection need a meeting? In: **Proceedings of the 1st ACM SIGSOFT Symposium on Foundations of Software ngineering**. [S.l.: s.n.], 1993. (SIGSOFT '93), p. 107–114. ISBN 0-89791-625-5.

WANG, C. et al. Multi-perspective visualization to assist code change review. In: **2017 24th Asia-Pacific Software Engineering Conference (APSEC)**. [S.l.]: IEEE Press, 2017. p. 564–569.

WEINBERG, G. M.; FREEDMAN, D. P. Reviews, walkthroughs, and inspections. **IEEE Transactions on Software Engineering**, SE-10, n. 1, p. 68–72, Jan 1984. ISSN 0098-5589.

WILLIAMS, L. Integrating pair programming into a software development process. In: **Proceedings of the 14th Conference on Software Engineering Education and Training**. [S.l.: s.n.], 2001. (CSEET '01), p. 27–. ISBN 0-7695-1059-0.

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

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

YANG, C. et al. An empirical study of reviewer recommendation in pull-based development model. In: **Proceedings of the 9th Asia-Pacific Symposium on Internetware**. [S.l.: s.n.], 2017. (Internetware'17), p. 14:1–14:6. ISBN 978-1-4503-5313-7.

YU, Y. et al. Reviewer recommendation for pull-requests in github: What can we learn from code review and bug assignment? **Information and Software Technology**, v. 74, p. 204 – 218, 2016. ISSN 0950-5849.

ZANJANI, M. B.; KAGDI, H.; BIRD, C. Automatically recommending peer reviewers in modern code review. **IEEE Transactions on Software Engineering**, v. 42, n. 6, p. 530–543, June 2016. ISSN 0098-5589.

ZHANG, T. et al. Interactive code review for systematic changes. In: **2015 IEEE/ACM 37th IEEE International Conference on Software Engineering**. [S.l.]: IEEE Press, 2015. (ICSE '15, v. 1), p. 111–122. ISBN 978-1-4799-1934-5.

ZHANG, X. et al. Design and implementation of java sniper: A community-based software code review web solution. In: **2011 44th Hawaii International Conference on System Sciences**. [S.l.]: IEEE Computer Society, 2011. p. 1–10. ISSN 1530-1605.

APPENDIX A — RESUMO ESTENDIDO

A revisão de código é uma prática amplamente conhecida de garantia de qualidade no desenvolvimento de software, consistindo em um esforço manual dos desenvolvedores para verificar o código-fonte antes do lançamento em um produto. Desde a primeira descrição da prática em 1976, na forma de inspeção de software, diversas melhorias, novas maneiras de executá-la e abordagens para apoiar a revisão de código foram propostas e estudadas. Atualmente, o formato popular de revisão é conhecido como revisão de código moderna (modern code review ou MCR, em Inglês), uma variante menos formal quando comparada à inspeção de software.

MCR é uma prática caracterizada pelo processo flexível, baseado em ferramentas e assíncrono, com foco em pequenas alterações de código e ocorrendo cedo, rapidamente e frequentemente durante o desenvolvimento do software. Embora o termo MCR tenha se tornado popular a partir de um estudo de 2013, a revisão de código com características de MCR foi praticada e estudada anos antes. Atualmente, múltiplas empresas e comunidades de código aberto adotam a prática, o que tem motivado muitos estudos a entender a atividade ou propor abordagens de suporte, por exemplo.

Dado o crescente interesse e a quantidade de pesquisa que vem sendo realizada sobre MCR, o objetivo principal deste trabalho é identificar e analisar o estado da arte da pesquisa em MCR. Com a condução de uma revisão sistemática da literatura sobre a prática, agregando o trabalho existente e apresentando uma visão geral do corpo de conhecimento sobre MCR, a intenção é ajudar pesquisadores a entender o que já foi explorado e questões em aberto. Tal visão geral também pode contribuir com aqueles que praticam MCR, os quais podem aprender como alcançar melhores resultados com a prática e ficarem cientes das soluções existentes de suporte a sua execução. De forma similar, outros trabalhos também objetivam agregar o conhecimento existente de MCR na literatura por meio de mapeamentos sistemáticos, contribuindo com a identificação dos temas gerais de pesquisa em MCR. O presente estudo difere desses trabalhos ao fornecer uma análise aprofundada dos resultados existentes sobre MCR.

Considerando o objetivo principal de identificar o estado da arte sobre MCR, questões de pesquisa (QP) específicas foram formuladas para delimitar o escopo da investigação: que conjunto de conhecimentos fundamentais foi construído com base nos estudos de MCR? (QP-1); quais abordagens foram desenvolvidas para dar suporte ao MCR? (QP-2); e como as abordagens MCR foram avaliadas e quais as conclusões obtidas? (QP-3). Para

responder tais questões de pesquisa foi definido e executado um procedimento sistemático de busca, seleção e análise de trabalhos científicos sobre MCR.

A pesquisa por publicações sobre MCR foi realizada nos mecanismos de busca da ACM Digital Library, IEEE Explore, Science Direct e Springer Link, retornando 702 resultados (sem duplicações). Destes resultados foram selecionados aqueles que apresentavam estudos sobre aspectos de MCR ou relatos de experiência sobre o uso da prática, propostas de solução para dar suporte a execução do MCR, e avaliações dessas soluções. No total, 110 artigos foram incluídos no conjunto de estudos primários a serem examinados nesta revisão sistemática. Identificamos três categorias principais de estudos primários: ESTUDOS FUNDAMENTAIS, PROPOSTAS de novas abordagens para apoiar a prática e estudos empíricos de AVALIAÇÃO dos soluções existentes de MCR. Cada categoria foi analisada sistematicamente, observando aspectos relevantes para cada tipo de trabalho.

A maior parte do corpo de conhecimento sobre MCR (QP-1) investigado consiste em ESTUDOS FUNDAMENTAIS que foram realizados para entender melhor a dinâmica do MCR, seus desafios e benefícios, e a análise de quais fatores de influência levam a quais resultados do MCR. A literatura então apresenta evidência da flexibilidade do processo de revisão, o qual é conduzido em média por um ou dois revisores, os quais (em projetos de código aberto) escrevem em média dois ou três comentários. Gerrit e sistemas de pull-request são as ferramentas de suporte mais comuns, além dos sistemas proprietários e internos de empresas. A compreensão é o desafio mais frequente, seja para ganhar familiaridade com o código ou para entender o que motivou a alteração. Quanto aos benefícios esperados, a motivação para realizar a revisão de código está relacionada principalmente a melhoria de código, identificação de defeitos e compartilhamento de conhecimento.

O corpo de conhecimento sobre MCR também apresenta evidência do impacto positivo da prática na qualidade do produto, principalmente na redução de defeitos pós-entrega quando há maior cobertura e participação na revisão de código. Outro aspecto investigado é relacionado aos fatores que influenciam a prática. Há evidência da correlação entre fatores técnicos e não técnicos com resultados internos da prática, como a relação entre a familiaridade do autor com o código da alteração e a revisão dessa alteração, entre outros. Assim, revisões de trechos modificados por desenvolvedores novatos tendem a envolver mais revisores, receber mais feedback de revisão e durar mais tempo, por exemplo. Outras descobertas indicam fatores que influenciam o nível de concordância e efetividade dos revisores, a quantidade de revisores, o tamanho do feedback de revisão

e sua qualidade, o volume de código alterado durante a revisão, o número de interações, o atraso para começar a revisão, a duração total do processo, a velocidade e a taxa de aceitação da revisão de código.

Entre as PROPOSTAS de novas abordagens de suporte ao MCR (QP-2), o mais comum são os recomendadores de revisores de código para reduzir o tempo e apoiar a seleção dos revisores considerando o trecho de código alterado. Outro tipo de abordagem frequentemente apresentada é a decomposição do código alterado, técnica que modifica a visualização para auxiliar os revisores a compreender o que foi feito, um dos desafios percebidos nos estudos fundamentais. Entre as AVALIAÇÕES das abordagens de apoio ao MCR (QP-3), a maior parte dos estudos foram feitos principalmente offline (execução da solução com dados históricos para verificar a saída) e poucos estudos envolvendo seres humanos foram realizados. Assim, o foco das avaliações tem sido a validação da saída da técnica ou ferramenta quanto a sua efetividade, em geral com propostas mais recentes obtendo melhores resultados.

A análise da literatura existente sobre MCR também permitiu identificar questões que ainda não foram endereçadas nesse contexto. Assim, há oportunidades de investigação sobre: (i) como os profissionais podem utilizar indicadores do processo de MCR para melhorar o processo de revisão; (ii) quais lições e diretrizes relacionadas a melhoria de código podem ser extraídas dos comentários de revisão para auxiliar desenvolvedores; (iii) como benefícios não técnicos do MCR podem ser mapeamos e melhorados; e a (iv) realização de mais estudos com usuários para avaliar soluções, não limitando a validação das saídas e da viabilidade das propostas.

Em síntese, esta tese consiste em uma visão geral da atual pesquisa de ponta em MCR, com uma análise aprofundada dos aspectos pesquisados nesse contexto. A pesquisa pode ser usada pelos desenvolvedores para aprender aspectos do MCR, para que eles possam melhorar suas práticas, e por pesquisadores e profissionais para ter uma base sólida sobre esse tópico para o desenvolvimento de futuras abordagens, avaliações e estudos fundamentais. Portanto, este trabalho consiste em uma contribuição significativa para a compreensão do corpo de conhecimento do MCR.