A systematic literature review and taxonomy of modern code review ☆

Nicole Davila *, Ingrid Nunes

Universidade Federal do Rio Grande do Sul (UFRGS), Instituto de Informática, Porto Alegre, Brazil

ARTICLE INFO

ABSTRACT

Context: Modern Code Review (MCR) is a widely known practice of software quality assurance. However, the existing body of knowledge of MCR is currently not understood as a whole.
Objective: Our goal is to identify the state of the art on MCR, providing a structured overview and an in-depth analysis of the research done in this field.
Methods: We performed a systematic literature review, selecting publications from four digital libraries.
Results: A total of 139 papers were selected and analyzed in three main categories. FOUNDATIONAL STUDIES are those that analyze existing or collected data from the adoption of MCR. PROPOSALS consist of techniques and tools to support MCR, while EVALUATIONS are studies to assess an approach or compare a set of them.
Conclusion: The most represented category is FOUNDATIONAL STUDIES, mainly aiming to understand the motivations for adopting MCR, its challenges and benefits, and which influence factors lead to which MCR outcomes. The most common types of PROPOSALS are code reviewer recommender and support to code checking. EVALUATIONS of MCR-supporting approaches have been done mostly offline, without involving human subjects. Five main research gaps have been identified, which point out directions for future work in the area.

1. Introduction

Code review is a widely known practice of software quality assurance. It consists of developers, other than the author, manually checking code changes before they are integrated into the main code repository. The goal is to look for defects or improvement opportunities without the software execution and before the product delivery, thus reducing the costs of fixing them later (Baum et al., 2016b). Due to the adoption of agile methods and distributed software development, code reviews are currently done in a less formal way than in the past, reducing the inefficiencies of its early form, namely software inspections (Fagan, 1976). The lightweight variant of code review has been referred to as *modern code review* (MCR) (Bacchelli and Bird, 2013), which is a flexible, tool-based, and asynchronous process. 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 and Bird, 2013). Moreover, because MCR is mediated by tools, it is possible to leverage MCR databases to learn from them and build complementary tools (Thongtanunam et al., 2015b; Ouni et al., 2016).

The increasing popularity of MCR in open source and industrial projects motivated studies on the practice, resulting in a large and also increasing body of knowledge in the literature. Due to this, it is a challenge to identify the main contributions, difficulties, and research opportunities related to MCR. Therefore, it is important to gather and review the existing contributions to be able to understand the state of the art and guide future efforts.

Our goal in this study is thus to obtain a comprehensive overview of the literature on MCR, which is helpful for researchers and practitioners to understand what has already been explored, open questions, and lessons learned about the practice. To achieve this goal, we present the results of a broad and in-depth systematic literature review of research work on MCR. Following a systematic method to search, select, and analyze relevant studies, 139 (out of 825 retrieved) papers have been investigated. We focus on answering three research questions, each related to a different type of study. Each type corresponds to one of the three main typical types of research contributions in this context. FOUNDATIONAL STUDIES are those that analyze existing or collected data related to MCR, exploring various aspects of the practice and deriving knowledge to improve it. From these studies, we discuss their key findings, which consist mainly of evidence about the practice in real settings, such as common characteristics, influencing factors, process variants, and impact of the practice. PROPOSALS consist of proposed approaches, such

as techniques and tools, to support MCR. Learning the proposals that are currently available allows researchers to understand and address limitations of existing support to MCR and practitioners to know what is available for them to improve the MCR practice in particular projects. Lastly, EVALUATIONS are studies to assess a proposed approach or compare a set of them. We compile how existing approaches compare to each other, thus providing evidence of their effectiveness. Moreover, this analysis also provides guidance for researchers to understand how to evaluate their work on MCR. Furthermore, based on our collected and analyzed data, we derive a new taxonomy of MCR research and identify issues that remain unaddressed. This paves the way for future work on MCR, which is a practice that has been largely used to improve the quality of software systems.

There are previous secondary studies on MCR. Three systematic mapping studies were done on this topic. Two (Fronza et al., 2020; Badampudi et al., 2019) aimed at identifying themes that are typically target of research, while that made by Coelho et al. (2019) focused on refactoring-aware code review. In addition, there are systematic literature reviews on MCR with a narrower scope than ours. They cover the MCR benefits (Nazir et al., 2020), the individual elements that impact on knowledge sharing (Fatima et al., 2019a,b), and the situational variables influencing sustainability in MCR (Nazir et al., 2019b,a). In addition, Doğan et al. (2019) conducted a narrative literature review to investigate the validity of ground truth data in reviewer recommender studies. Although these secondary studies are relevant contributions to the field, our review takes a step further by making a broader and in-depth analysis of the literature and capturing a structured view of their key findings.

We next provide background on MCR. The methodology adopted to perform our systematic literature review is presented in Section 3, followed by the analysis of its results that are described in Sections 4–6. Insights derived from our study and its limitations are discussed in Section 7. In Section 8, we conclude.

2. Background on modern code review

The term *modern code review* (MCR) was first used by Cohen (2010) and became popular by the study of Bacchelli and Bird (2013). 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—i.e. the reviewers—assess code changes to identify both defects and quality problems and decide whether the changes will be discarded or integrated into the main project repository. MCR is mainly characterized by being asynchronous and supported by tools, occurring regularly integrated with the routine of developers (Bacchelli and Bird, 2013; Sadowski et al., 2018).

To understand how MCR generally works, we summarize in Fig. 1 a typical MCR process, capturing tasks performed to review a code change using well-known code review tools, such as Gerrit and the pull-request system of GitHub. Each organization or project can have a customized MCR process with particular tools, policies, and culture. Fig. 1 splits the MCR tasks into two phases, *Review Planning and Setup* and *Code 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 changes the source code and creates 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 and Nunes, 2018) (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).
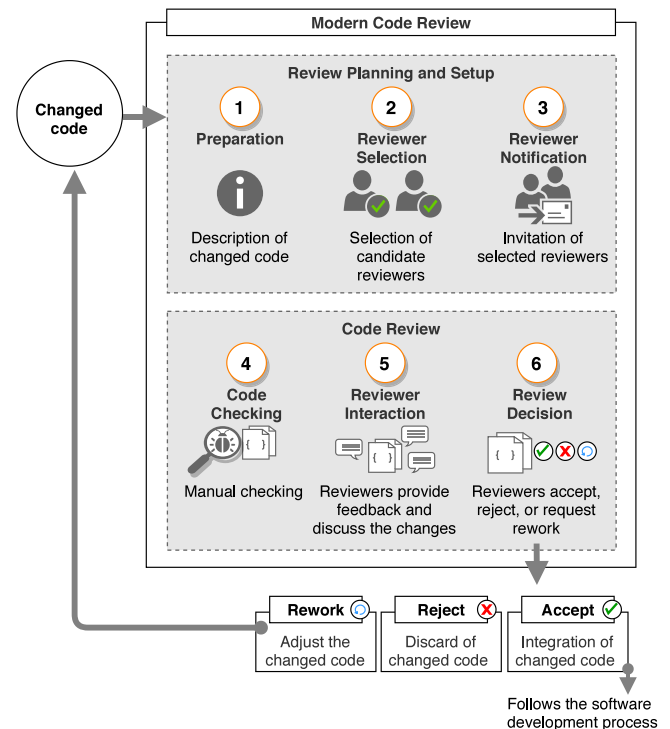

Fig. 1. An overview of the tasks of modern code review.

The first phase, *Review Planning and Setup*, consists of three main tasks. In the first (*Preparation*), the author, who has a unit of work to be reviewed, prepares a review package composed of the changed code accompanied by a description providing additional details of the change. In the *Reviewer Selection* task, suitable reviewers that likely can or should inspect the review package are selected. Finally, in the *Reviewer Notification* task, reviewers receive a notification inviting them to perform the review.

In the second phase, *Code Review*, the process of analyzing the code for defects and improvement opportunities takes place. Each reviewer individually performs *Code Checking*, assessing changes and comparing it with previous versions. They also 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 *accept*, *reject*, and *rework*. The last results in a new review cycle to take place after the code is updated according to the reviewer feedback.

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 motivated research to identify and understand this practice as well as improve it. This led to many studies, which are surveyed in this work.
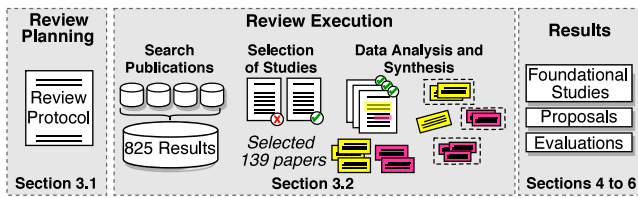
Fig. 2. Main steps of our systematic literature review.

3. Systematic literature review

To perform our systematic literature review (SLR), we followed the guidelines proposed by Kitchenham and Charters (2007). We executed three main steps—Review Planning, Review Execution, and Data Analysis and Synthesis—which are illustrated in Fig. 2. This section details the protocol followed to conduct our SLR and the results of the literature search and selection.

3.1. Systematic review planning

3.1.1. Research goals

Our goal is to obtain a comprehensive overview of the literature on MCR. Over the past years, this topic received significant research interest, which led to many studies. Our goal is to gather, structure, and compare studies' findings. Such an in-depth analysis of research on MCR 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 by identifying how to adjust the MCR practice in particular settings and becoming aware of existing approaches that can be adopted to support the practice.

3.1.2. Research questions

Based on our goal, our main research question is: *what is the state of the art of research on modern code review?* Considering this broad research question, we focus on different types of research work, covering three typical research directions: FOUNDATIONAL STUDIES, which are those that analyze existing or collected data associated with MCR to gather knowledge about the practice that provide practitioners with knowledge to improve the adoption of the practice or researchers with findings to build further research on them; PROPOSALS, which consist of novel techniques and tools to support MCR, and EVALUATIONS, which are studies to assess a proposed approach or compare a set of them. This leads us to 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?

3.1.3. Search strategy

To find the studies that are relevant to our work, we selected four databases, shown in Table 1, which store papers published in many key software engineering conferences and journals. Our search does not include other databases, e.g. Google Scholar and arXiv, because they provide pointers to papers already stored in our searched databases or include non-peer-reviewed papers. We focus on papers that are peer reviewed as a means to obtain evidence of the quality of their research.

To retrieve the publications from the selected databases, we used as keyword the term *code review* and the following synonyms: *code inspection*, *software inspection*, and *formal inspection*.

Table 1
Search results by source.

Databases	URL	#Studies
ACM DL	http://portal.acm.org	291
IEEE Xplore	http://ieeexplore.ieee.org	394
ScienceDirect	http://www.sciencedirect.com	122
SpringerLink	http://link.springer.com	76
Duplicates		58
Total (including duplicates)		**883**
Total (excluding duplicates)		**825**

Table 2
Inclusion criteria (IC) and exclusion criteria (EC).

Criteria	
IC-1	The paper presents a study of foundational aspects of MCR.
IC-2	The paper proposes an approach to support MCR.
IC-3	The paper presents an evaluation of an MCR approach.
EC-1	The paper does not focus on MCR as a reviewing practice made by peers within the software development.
EC-2	The paper is not written in English.
EC-3	The content of the paper was also published in another more complete paper, which is already included.
EC-4	We have no access to the full paper.
EC-5	The content is not a scientific paper of at least four pages.

Though our goal is to identify work on MCR, some authors use the term inspection but refer to a lightweight process. We, therefore, identified whether papers focus on MCR not by the adopted term but by their description of the practice. The term *code review* is a substring of other synonyms, e.g. modern code review, changed-based code review, contemporary peer code review, and peer code review. Thus, there is no need to include them. The only term not covered, which was used by a few authors, is *peer review* because it leads to a huge amount of papers related neither to MCR nor to Computer Science and would make the SLR infeasible to be done in a timely manner. Our resulting search string is as follows.

> **Search String**: "code review" OR "code inspection" OR "software inspection" OR "formal inspection"

3.1.4. Selection criteria

The papers retrieved by querying search databases are filtered using selection criteria, summarized in Table 2. To be selected, a primary study must satisfy at least one inclusion criterion and no exclusion criterion.

Our EC-1 excludes studies that use MCR as a motivation or as an example of context where an approach 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.

3.2. Review execution

The execution of the review protocol detailed in the previous section led to the results described as follows.

Table 3
Selection of studies based on inclusion and exclusion criteria. In the full analysis, 33 papers matched two IC and two papers matched three IC.

Step/Criterion	IC-1	IC-2	IC-3	Total
Abstract analysis	228	146	45	378
Full analysis	98	60	37	158
EC-1				0
EC-2				0
EC-3	3	5		8
EC-4				0
EC-5	9	2		11
Selected studies	**86**	**53**	**37**	**139**

3.2.1. Search execution

Our search string was customized according to the specific syntax of each search engine. We searched within the abstracts of the publications. Due to limitations in the Springer Link database, which does not allow searches in abstracts, we considered paper keywords instead. We searched for papers published before or in 2019. The number of retrieved papers is shown in Table 1.

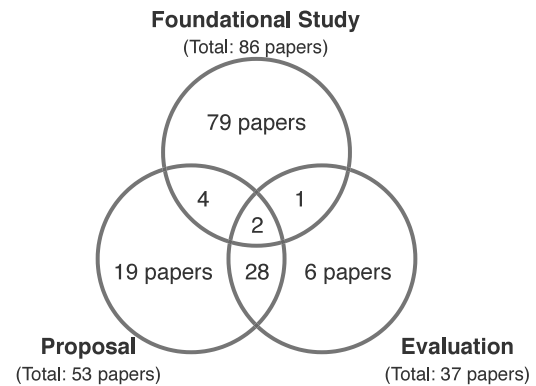### 3.2.2. Selection of primary studies

To select primary studies, we first selected papers that matched any inclusion criterion, based on their title and abstract. Then, in a second step, we evaluated all inclusion and exclusion criteria based on the full text of the paper. Table 3 depicts the results of the execution of these steps. After a precise specification of each IC and EC and the joint analysis of papers for verifying a common understanding of their meaning, each primary study was analyzed by a single researcher. If the researcher could not evaluate the satisfaction of any of the criteria, the opinion of the second researcher was requested to minimize the potential researcher bias. If there was no agreement, both researchers discussed until they converged to a decision.

Papers that do not focus on MCR do not match any IC. For example, there are studies excluded because they focus on software inspection as a synchronous process with an inspection meeting. Moreover, there are papers satisfying multiple IC, e.g. (Rahman et al., 2017; Thongtanunam et al., 2014), which describe a combination of foundational studies (IC-1), proposed approach (IC-2), and evaluation (IC-3). In addition, there are papers discarded due to the satisfaction of ECs. Due to EC-1, we excluded studies of pedagogical code review (Hundhausen et al., 2011; Petersen and Zingaro, 2018) and static analysis tools (Burhandenny et al., 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.

Our final set of selected primary studies has a total of 139 publications. Papers are categorized according to the IC that they satisfy: (i) FOUNDATIONAL STUDIES investigate aspects of MCR or an experience report of the use of MCR (IC-1); (ii) PROPOSALS describe novel approach to support MCR, such as a tool, technique or method (IC-2); and (iii) EVALUATIONS of MCR approaches (IC-3). Throughout the paper, the highlighted terms are used to refer to these study types. Although the majority of papers present a single type of contribution, 35 include two or three types of studies and are therefore in multiple categories, as shown in the Venn diagram in Fig. 3. Table 4 shows the papers selected for analysis grouped by category.

3.2.3. Data extraction and information labeling

To answer each research question, we extracted information from each study according to the facets indicated in Fig. 4. If a paper includes multiple types of study, each study of the paper was analyzed separately.

Fig. 3. Distribution of primary studies in three main categories and their intersections.
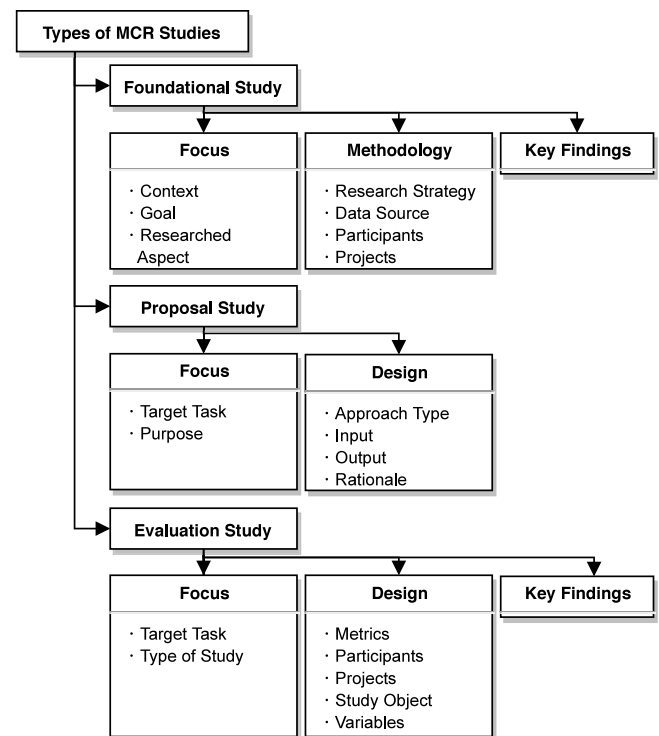
Fig. 4. Facets analyzed in each primary study.

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. A classification was jointly elaborated by the two authors of this paper, with definitions as precise as possible of each classification to be made. To maintain consistency in the classification, the studies of each main category, i.e., FOUNDATIONAL STUDIES, PROPOSALS, and EVALUATIONS, were carefully read, analyzed, and coded by a single researcher. In cases in which there was no certainty while classifying the paper content based on the definitions, the two researchers analyzed the study and discussed until they converged on the codes that should be used. From the 139 primary studies, there were 27 (approximately 20%) cases in which the

Table 4

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), Rigby and Bird (2013), German et al. (2018), Rahman and Roy (2017), Sadowski et al. (2018), Bosu et al. (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. (2016c), Bird et al. (2015), Kononenko et al. (2016), Bosu and Carver (2014), Thongtanunam et al. (2015a), Kovalenko and Bacchelli (2018), Efstathiou and Spinellis (2018), Bosu et al. (2014), Rahman et al. (2017), Begel and Vrzakova (2018), Dunsmore et al. (2000), Hirao et al. (2015), Thongtanunam et al. (2016), Floyd et al. (2017), Uwano et al. (2006), Lee and Carver (2017), Meneely et al. (2014), Bosu et al. (2017), Sutherland and Venolia (2009), Chandrika et al. (2017), Asundi and Jayant (2007), MacLeod et al. (2018), Ueda et al. (2018), Armstrong et al. (2017), Thongtanunam et al. (2015b), Ebert et al. (2017), Kononenko et al. (2015), Bavota and Russo (2015), Runeson and Andrews (2003), Bosu and Carver (2013), Liang and Mizuno (2011), di Biase et al. (2016), Panichella et al. (2015), Swamidurai et al. (2014), Morales et al. (2015), Baysal et al. (2012), Rigby et al. (2012), Bernhart and Grechenig (2013), Duraes et al. (2016), Murakami et al. (2017), Li et al. (2017), Thongtanunam et al. (2017), Baum et al. (2017a), McIntosh et al. (2016), Baysal et al. (2016), Hirao et al. (2016), Alami et al. (2019), Pascarella et al. (2018), Spadini et al. (2019), Zanaty et al. (2018), Hirao et al. (2019), Ram et al. (2018), Ebert et al. (2018), An et al. (2018), Ueda et al. (2018), Norikane et al. (2018), Ueda et al. (2019), Zampetti et al. (2019), Ebert et al. (2019), Jiang et al. (2019b), Paixao and Maia (2019), Paul et al. (2019), Paixao et al. (2019), El Asri et al. (2019), Wang et al. (2019b), Baum et al. (2019), Ruangwan et al. (2019), Santos and Nunes (2018)

PROPOSALS
Yu et al. (2016), Jiang et al. (2017), Bosu et al. (2015), Müller et al. (2012), Balachandran (2013), Barnett et al. (2015), Rahman et al. (2016), Zhang et al. (2015), Sripada et al. (2016), Rahman et al. (2017), Thongtanunam et al. (2014), Hao et al. (2013), Tao and Kim (2015), Priest and Plimmer (2006), Soltanifar et al. (2016), Duley et al. (2010), Uwano et al. (2006), Kalyan et al. (2016), Ge et al. (2017), Zanjani et al. (2016), Tymchuk et al. (2015), Ahmed et al. (2017), Baum et al. (2017b, 2016b), Nagoya et al. (2005), Lanubile and Mallardo (2002), Harel and Kantorowitz (2005), Thongtanunam et al. (2015b), Ebert et al. (2017), Pangsakulyanont et al. (2014), Xia et al. (2017), Zhang et al. (2011), Mishra and Sureka (2014), Ouni et al. (2016), Menarini et al. (2017), Perry et al. (2002), Wang et al. (2017), Xia et al. (2015), Aman (2013), Fejzer et al. (2018), Fan et al. (2018), Luna Freire et al. (2018), Li et al. (2017), Baum and Schneider (2016), Asthana et al. (2019), Huang et al. (2018b), Wang et al. (2019a), Huang et al. (2018a), Wen et al. (2018), Hanam et al. (2019), Liao et al. (2019), Jiang et al. (2019a), Guo et al. (2019)

EVALUATIONS
Yu et al. (2016), Khandelwal et al. (2017), Müller et al. (2012), Balachandran (2013), Yang et al. (2017), Barnett et al. (2015), Rahman et al. (2016), Zhang et al. (2015), Rahman et al. (2017), Thongtanunam et al. (2014), Tao and Kim (2015), Baum et al. (2016a), Duley et al. (2010), Peng et al. (2018), Hannebauer et al. (2016), Ge et al. (2017), Zanjani et al. (2016), Thongtanunam et al. (2015b), Xia et al. (2017), Ouni et al. (2016), Menarini et al. (2017), Xia et al. (2015), Aman (2013), Fejzer et al. (2018), Fan et al. (2018), Luna Freire et al. (2018), Mizuno and Liang (2015), Runeson and Wohlin (1998), Asthana et al. (2019), Huang et al. (2018b), Wang et al. (2019a), Huang et al. (2018a), Wen et al. (2018), Hanam et al. (2019), Liao et al. (2019), Jiang et al. (2019a), Guo et al. (2019)

first author faced uncertainty in the classification. Considering the FOUNDATIONAL STUDIES, the researchers had five interactions until they reached a final version of the codes. There were four refinement interactions for PROPOSALS, and for EVALUATIONS, there was the need for only two interactions. For example, considering de refinement of aspects related to FOUNDATIONAL STUDIES, we had in the set of codes the labels *Analysis of the Impact on Internal Outcomes* and *Analysis of Activities Outcomes*, which after the refinements were merged into *Analysis of Internal Outcomes* because both included studies targeting internal outcomes.

The coding process was performed at different levels in each main category, as summarized in Fig. 4. We used coding to categorize information of the study focus, methodology, and key findings of FOUNDATIONAL STUDIES. In the PROPOSALS category, we coded the approach focus and its design. Then, studies in the EVALUATION group were labeled considering their focus, design, and key findings.

3.2.4. Findings of the systematic literature review

The next three sections present our findings by groups of studies (FOUNDATIONAL STUDIES, PROPOSALS, and EVALUATIONS), which are associated with our three research questions, respectively.[1] We first provide an overview of the category and then we discuss the subcategories in detail. Throughout the discussion, we highlighted the key findings of each category using frames. At the end of each section we answer the respective research question.

4. FOUNDATIONAL STUDIES

We classified the 86 FOUNDATIONAL STUDIES present in the selected papers into seven categories, as presented in Table 5.

Most of the studies focus solely on MCR, while eight investigate the relationship of the practice with other approaches of software development. Studies classified as *practitioner perception of the practice* and *analysis of innerworkings of MCR process* analyze the state of the practice, exploring multiple aspects and sharing lessons learning from studies in real settings. Studies focusing on *cross-patch analysis* search for links and patterns related to multiple reviewed patches. The other two categories—*analysis of internal outcomes* and *analysis of external outcomes*—investigate specific characteristics of the MCR process by means of data analysis. A smaller set of studies explores simulations or human body data (e.g., brain and eye activity) to understand *human aspects of reviewers*. Finally, there are papers that focus on the relationship of MCR with other practices, which are classified as *external aspects of MCR*. Given this overview of the categories of FOUNDATIONAL STUDIES, we next analyze each of them.

4.1. Practitioner perception of the practice

We identified in the literature 18 FOUNDATIONAL STUDIES that explore multiple perspectives of the MCR practice collecting the perceptions of the practitioners. In most studies, the opinion and thoughts of practitioners were collected by means of questionnaires (German et al., 2018; Kononenko et al., 2016; Bosu et al., 2017; Sutherland and Venolia, 2009; Bosu and Carver, 2013; Bernhart and Grechenig, 2013; Baum et al., 2017a; Ebert et al., 2019; Santos and Nunes, 2018), semi-structured interviews (Bosu et al., 2015; Spadini et al., 2018; Baum et al., 2016c; Alami et al., 2019; Ram et al., 2018), or both (Sadowski et al., 2018; Spadini et al., 2019). Moreover, some studies complemented these data using observation (Bacchelli and Bird, 2013; MacLeod et al., 2018) or meeting with practitioners (Alami et al., 2019), or deployed a specific tool to ask professionals for feedback (Ram et al., 2018). Half of these FOUNDATIONAL STUDIES involve participants from

---

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

Table 5

Taxonomy of topics researched in FOUNDATIONAL STUDIES.

Category	Main goal	Typical research method	#Studies
Practitioner perception of the practice	Learn from a developers' perspective the state of the practice of MCR	Collection and analysis of subjective or qualitative data with the opinion and thoughts of the practitioners	18
Analysis of the innerworkings of the MCR process	Analyze the reviewing process in real settings, sharing of insights and lessons learned of the practice adoption	Contributions based on quantitative analysis, experience or the analysis of a case study	6
Cross-patch analysis	Analyze the relationship among reviewed patches and identify recurring patterns	Collection and analysis of data from code review traces and software development history	3
Analysis of internal outcomes	Understand key properties and metrics associated with the MCR process and what influences them	Collection and analysis of data of code in repositories or collected in experiments	39
Analysis of external outcomes	Understand how MCR influences outcomes external to the review process (code and product quality)	Collection and analysis of data from repositories of code review, version control systems and issue trackers	13
Human aspects of reviewers	Understand human behavior and individual aspects of the reviewers	Simulations or experiments to collect human body data (e.g., brain and eye activity)	9
External aspects of MCR	Analyze of the relationship between MCR and other software development practices	Experiments or data analysis from software development history	8

both industry and open-source projects. The remaining studies on perceptions of the practice involved only participants from open-source software (OSS) projects (German et al., 2018; Kononenko et al., 2016; Bosu and Carver, 2013; Alami et al., 2019) or only participants from the industry (Bacchelli and Bird, 2013; Sadowski et al., 2018; Bosu et al., 2015; Baum et al., 2016c, 2017a; Santos and Nunes, 2018).

The studies that capture the practitioner perception of the practice focus on four different MCR aspects, namely (i) *reviewing practices*; (ii) *social aspects*; (iii) *expected benefits*; and (iv) *challenges and difficulties*. The majority of the studies target only one aspect, while six of them (Bacchelli and Bird, 2013; Sadowski et al., 2018; Baum et al., 2016c; Kononenko et al., 2016; Bosu et al., 2017; MacLeod et al., 2018) study multiple aspects. We next detail the findings provided by these studies.

> **Finding 1**: The perception of the MCR is collected mainly through questionnaires (13 of 18 studies, approximately 72%) and provides evidence of practitioners' opinions from industry (50% of the studies), open-source projects (22%), or both (28%).

Reviewing practices. More than a half of the studies that collected the opinion of practitioners (Bosu et al., 2015; Baum et al., 2016c; Kononenko et al., 2016; Bosu et al., 2017; Sutherland and Venolia, 2009; MacLeod et al., 2018; Bernhart and Grechenig, 2013; Baum et al., 2017a; Alami et al., 2019; Ram et al., 2018; Ebert et al., 2019; Spadini et al., 2019; Santos and Nunes, 2018) aim to better understand the MCR adoption. The goal is to identify the process variations and how developers perform code review.

Six studies gathered perceptions of reviewing practices in proprietary settings. To understand why code reviews are used, Baum et al. (2016c) interviewed professionals to identify factors influencing the adoption 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. In a posterior study, Baum et al. (2017a) refined this finding surveying professionals from commercial teams. They concluded that MCR (i.e. change-based code review) is the most

common type of review and the risk of review fade away increases when there are no rules or conventions, which supports previous findings. In addition to these findings, Bernhart and Grechenig (2013) reported a positive effect of continuous review practice on the understandability and collective ownership in a particular project. Two works focused on investigating communication in MCR (Sutherland and Venolia, 2009; Bosu et al., 2015). There is evidence of face-to-face and digital communication, not restricted to a tool that is dedicated to support MCR (Sutherland and Venolia, 2009). Overall, this communication is perceived as useful and helps developers understand the design rationale. Specifically, the review feedback is useful when it helps the author improve the code quality and triggers a code change (Bosu et al., 2015). Nevertheless, review comments with questions for clarifications are not considered useful by authors, but they may be useful to the reviewers. Moreover, another study (Santos and Nunes, 2018) explored the opinion of professionals of how different factors (such as the patch size) influence internal outcomes of MCR (such as its duration and provided feedback). According to the study participants, the large number of active reviewers has a positive influence on the number of comments, while a large patch size and having more teams involved might have a negative influence on the review duration.

The perception of the practitioners of reviewing practices was also investigated in OSS, to understand why code review works in this context. Alami et al. (2019) identified that in OSS the practice is more related to human and social aspects, particularly the hacker ethics. This study also observed that rejections and negative feedbacks are common in such a context, but there are professionals who perceive it as an opportunity for learning and technical growth due to the iterative improvement cycle. Concerning the practice quality, review feedback is also pointed out as a key aspect by Kononenko et al. (2016). The thoroughness of the feedback is associated with the review quality by practitioners as well as the reviewer's familiarity with the code, which is also observed in studies relying on objective data (discussed in later sections).

Lastly, the perception of practitioners on code review are also explored by research work that consider both contexts, i.e. OSS and industry. Ebert et al. (2019) focused on exploring the main causes of confusion during a review and found as frequent reasons the missing rationale, discussion of non-functional requirements, and lack of familiarity with code. To cope with

this confusion, 13 strategies were identified, such as information requests, improved familiarity with the existing code, and off-line discussions. Ram et al. (2018), in turn, identified what makes a change easier to review. They concluded that reviewability can be defined through several factors, such as change description and size. Moreover, Spadini et al. (2019) explored the opinion of practitioners on test-driven code review (TDR), a variation of MCR when test-code is reviewed before the production code. They examined the perception of both problems and advantages of TDR, concluding that developers prefer to review production code as they consider it more critical and tests should follow from it.

> **Finding 2**: Multiple factors influence the adoption of MCR. In the industry, its adoption is influenced by culture, product, tools, development process, and team. In open-source projects, the influence is more related to human and social aspects, particularly hacker ethics.

> **Finding 3**: The review feedback is perceived as valuable when it provides an opportunity to learn or improve the code.

Social aspects. MCR is a collaborative activity that relies on intensive human interaction. This motivated the researchers to investigate the social issues that emerge from the interaction among peers. There are three studies that focus on social aspects associated with MCR based on the developers' perceptions. German et al. (2018) investigate *fairness*, analyzing how practitioners perceive the treatment given and received in code review. The results indicate that a significant proportion of participants perceives unfairness in MCR. This observation is more common among authors than reviewers. The other two studies (Bosu et al., 2017; Bosu and Carver, 2013) explore 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.

Expected benefits. The *key* benefit expected of software inspection was the early defect identification. Though the analyzed studies also reported this as an expected benefit of MCR, only one survey (Bacchelli and Bird, 2013) points out defect identification as the *primary* expected benefit of MCR, being other benefits higher ranked. Most surveys report code improvement as the main desired benefit of MCR (Baum et al., 2016c; Bosu et al., 2017; MacLeod et al., 2018), which means obtaining a better internal code quality, readability, and maintainability. Knowledge sharing and learning also emerged as key expectations Sadowski et al. (2018), Baum et al. (2016c), Bosu et al. (2017). In this case, people involved in a review desire to gain knowledge about the code, module, or coding style, for example. Moreover, practitioners also expect to discuss alternative solutions during the reviewing process, collaboratively developing new and better ideas Bacchelli and Bird (2013), Baum et al. (2016c). In summary, (1) the two main expected benefits are *code quality improvement* and *defect identification*; and (2) MCR promotes additional benefits, such as *knowledge sharing* and *learning*.

Challenges and difficulties. Five studies (Sadowski et al., 2018; Spadini et al., 2018; Baum et al., 2016c; Kononenko et al., 2016; MacLeod et al., 2018) investigate what difficulties developers face when performing the code review activity, especially in an industrial environment. Understanding the motivation and purpose of a change has been pointed out as a challenge for practitioners of MCR (Bacchelli and Bird, 2013; MacLeod et al., 2018; Kononenko et al., 2016; Sadowski et al., 2018). There are studies (Kononenko et al., 2016; Sadowski et al., 2018) that indicate that gaining familiarity with the code is a technical challenge faced by reviewers because reviewing an unfamiliar code might result in misunderstandings. Moreover, in the context of reviewing a test code (Spadini et al., 2018), test files require developers to understand not only the code being reviewed (test files) but also the associated production files.

The other MCR challenges reported in the reviewed literature are more scattered. Two of the five studies (Kononenko et al., 2016; MacLeod et al., 2018) indicated time management as a challenge. 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 two studies (Sadowski et al., 2018; Kononenko et al., 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.

> **Finding 4**: The practitioners expect as effect of MCR the improvement of code quality and defect detection. Additionally, there are expectations related to knowledge sharing and learning. The main perceived challenge is to understand the code change, its purpose, and motivation.

4.2. Analysis of the innerworkings of the MCR process

Six studies focus on understanding the internal mechanisms (i.e. innerworkings) of the MCR process, sharing findings, experiences and lessons learned from the adoption of MCR in real settings. Three of them share insights on the use of MCR in particular projects (Bird et al., 2015; Izquierdo-Cortazar et al., 2017; Baysal et al., 2012), while three studies report findings, lessons learned and recurrent practices of MCR in more than one setting (Rigby et al., 2012; Rigby and Bird, 2013; Paixao and Maia, 2019).

Particular projects were investigated to understand the effects of process changes or the adoption of a particular tool on MCR, resulting in scattered findings. Izquierdo-Cortazar et al. (2017) worked together with the developers of the Xen Project in the analysis of internal aspects of MCR to understand how the performance of code review evolves. In another study (Baysal et al., 2012), data of Mozilla Firefox was examined to verify the differences between pre- and post-rapid release development, assuming that this change affects how the code is reviewed. Finally, Bird et al. (2015) described the development and use of CodeFlow Analytics (CFA), a Microsoft's internal platform, which allows developers to explore the historical data of code review. Despite the specific findings of each study, they contribute with insights from real environments. The work targeting the Xen Project contributes with the approach used for analysis, while the

report of CFA provides evidence of the positive impact of a tool for code review analytics. Moreover, there is evidence that post-rapid release is a successful approach to reduce the time patches wait for review, mainly considering patches from casual contributors, which are more likely to be abandoned.

In another study focusing on specific aspects of MCR, Paixao and Maia (2019) analyzed data from multiple open-source systems. In this case, the study focuses on rebasing operations and their relationship with code review. As key contributions, the authors observed that rebasing occurs in most of reviews and also tend to tamper with the reviewing process, which might negatively affect the practice.

Lastly, two studies discuss more general insights. Rigby et al. (2012) introduced various 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. In another work (Rigby and Bird, 2013), the analysis of several case studies led to the identification of 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. Tool-supported review is suggested to provide the benefit of traceability, and its increased adoption is an indicator of success.

4.3. Cross-patch analysis

We identified in our set of FOUNDATIONAL STUDIES research work (Hirao et al., 2019; Ueda et al., 2018, 2019) that explore patterns that occur across different patches of a project. Hirao et al. (2019) investigated the impact of review linkage on MCR analytics. They extracted review linkage graphs of six projects and found that linkage rates range from 3% to 25%. They identified 16 types of review links, which are distributed into five categories: patch dependency, broader context, alternative solution, version control issues, and feedback related. Moreover, there is evidence that exploiting review linkage can improve the performance of reviewer recommendation approaches. Differently, two studies aimed to identify behavioral patterns in the review feedback. Ueda et al. (2018) compared how authors handle issues raised by automated checkers and manually indicated by reviewers. As a result, they found that authors repeatedly introduce the same types of problems despite the reviewer feedback, which does not occur with issues found by checkers. Then, Ueda et al. (2019) explored source code improvement patterns. They identified several patterns and grouped the eight most frequent into three categories, namely project-specific, readability-improvement, and language-specific. In these studies, the researchers aimed to reduce the cost of review, identifying issues that might be addressed by authors before code review.

4.4. Analysis of MCR outcomes

The majority of FOUNDATIONAL STUDIES focus on objective and quantitative data from review practice, being this the most common type of study 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 MCR outcomes into two groups: (i) *internal outcomes*, which are associated with the MCR process (e.g. number of reviewers and number of comments); and (ii) *external outcomes*, which are observed in the software product (code quality and defects).

From the 48 works that focus on the analysis of MCR outcomes, we identified 27 studies exploring the correlation between 42 influence factors and 14 outcomes. We overview the studies of the relationship between influence factors and outcomes in Fig. 5. On the left-hand side, there are influence factors. On the right-hand side, there is the number of times that the influence of a factor over an MCR outcome was investigated. Table 6 describes both non-technical and technical influence factors, along with the studies that explored their influence. Additionally, there are studies that investigate the influence of internal outcomes over external outcomes. These internal outcomes are detailed in Table 7.

The majority of the studies on MCR outcomes used data from OSS repositories, while five (Shimagaki et al., 2016; Bacchelli and Bird, 2013; Bosu et al., 2015; Rahman et al., 2017; Santos and Nunes, 2018) of them collected information from commercial projects and two (Albayrak and Davenport, 2010; Murakami et al., 2017) from experiments. The results of these studies are discussed as follows.

Finding 5: Most of the foundational knowledge on outcomes of MCR is based on data from open-source projects. From 48 studies that focus on analyzing internal or external outcomes of MCR, 41 (85.4%) used data from OSS repositories. There are few studies in the industry (10.4%) and experiments (4.2%).

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, assessing and analyzing the values of these outcomes. Others inspect the relationship between influence factors and outcomes. From the 39 works that focus on internal outcomes, 33 used data from OSS projects, while the remaining ones used data from proprietary projects (Bacchelli and Bird, 2013; Bosu et al., 2015; Santos and Nunes, 2018; Rahman et al., 2017). Moreover, two of the studies collected and used data produced by participants of an experiment (Rahman et al., 2017; Murakami et al., 2017). We grouped the investigated internal outcomes into five groups: *reviewer team*, *review feedback*, *review intensity*, *review time*, and *review decision*. We next discuss the findings of each group.

Reviewer team. There are 14 studies that investigate the properties associated with the team of reviewers that are formed in code reviews. Few works (Yang et al., 2017; Bosu and Carver, 2012; Asundi and 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*. In OSS projects, an average of one or two reviewers per request responded to a review request (Yang et al., 2017; Bosu and Carver, 2012; Asundi and Jayant, 2007). Also in the context of OSS, 16%–66% of the patches have at least one invited reviewer who did not respond (Ruangwan et al., 2019). As we discuss later, the participation of reviewers in code review share a positive relationship with product quality.

Table 6

Influence factors analyzed by the FOUNDATIONAL STUDIES.

Group	Influence factor	Description	Studies
Non-technical factors			
Author	Code familiarity	Number of contributions authored by a developer in the project	Kovalenko and Bacchelli (2018), Bosu et al. (2014), Lee and Carver (2017), Bosu and Carver (2014), Thongtanunam et al. (2017), Kononenko et al. (2015), Ruangwan et al. (2019)
	Development experience	Number of contributions authored by a developer in general	Baysal et al. (2016), 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)
	Reviewing experience	Number of completed reviews of code changes	Ruangwan et al. (2019)
Reviewer	Code familiarity	Number of review contributions made by a developer in the project	Bosu et al. (2015), Thongtanunam et al. (2017), Kononenko et al. (2015), Rahman et al. (2017), Meneely et al. (2014), McIntosh et al. (2016), Ruangwan et al. (2019)
	Personal characteristics	Characteristics of the reviewer, such as age	Murakami et al. (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), Rahman et al. (2017), Baysal et al. (2016), Kononenko et al. (2015), Ruangwan et al. (2019)
	Review participation rate	Number of review invitations responded	Ruangwan et al. (2019)
Other	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), Ruangwan et al. (2019)
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	McIntosh et al. (2016), 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 et al. (2015), Beller et al. (2014), Thongtanunam et al. (2017)
	Size	Number of added or modified lines of code under review	Santos and Nunes (2018), Liang and Mizuno (2011), Thongtanunam et al. (2017), Beller et al. (2014), Baysal et al. (2016), Bosu et al. (2014), Meneely et al. (2014), Ruangwan et al. (2019)
	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 or scripts	Bosu et al. (2015)
Historical 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, 2015a)
	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)
Other	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 et al. (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)

Eight studies (Thongtanunam et al., 2017; Kovalenko and Bacchelli, 2018; Lee and Carver, 2017; Santos and Nunes, 2018; Liang and Mizuno, 2011; Thongtanunam et al., 2015a; Armstrong et al., 2017; Ruangwan et al., 2019) explored factors influencing the team size. As key findings, these studies indicate that both the description length (Thongtanunam et al., 2017) and patch size (Santos and Nunes, 2018) influence the number of reviewers participating in a review—long patch descriptions and small patch size might increase the likelihood of attracting reviewers. Moreover, considering a proprietary project developed

Table 7

Internal outcomes as influence factors analyzed by the FOUNDATIONAL STUDIES.

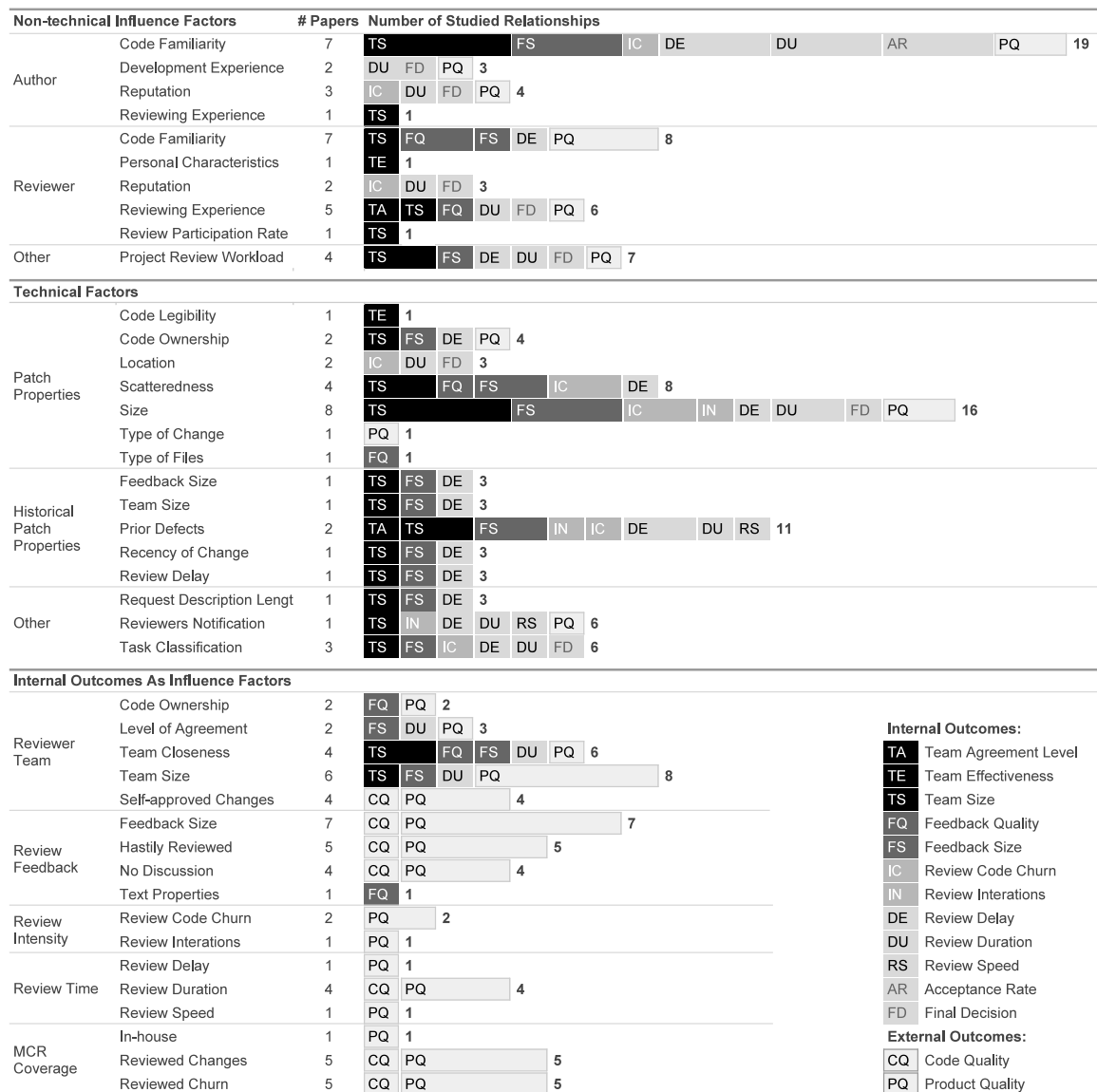| Group	Influence factor	Description	Studies
Reviewer team	Code ownership	Number of developers who uploaded a revision for the proposed changes	Rahman et al. (2017), Thongtanunam et al. (2015a)
	Level of agreement	The proportion of reviewers that disagreed with the review conclusions	Hirao et al. (2016), Thongtanunam et al. (2015a)
	Team closeness	Number of distinct geographically distributed development sites or number of distinct teams associated with the author and reviewers	Santos and Nunes (2018), Bosu et al. (2015), Meneely et al. (2014)
	Team size	Number of reviewers that participate in the reviewing process	Santos and Nunes (2018), Thompson and Wagner (2017), Thongtanunam et al. (2015a), 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 et al. (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 et al. (2015), Shimagaki et al. (2016), Thompson and Wagner (2017), Thongtanunam et al. (2015a), 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 et al. (2015), McIntosh et al. (2014), Shimagaki et al. (2016), Meneely et al. (2014), McIntosh et al. (2016)
	No discussion	Number of accepted review requests without any review comments	Morales et al. (2015), Shimagaki et al. (2016), McIntosh et al. (2014, 2016)
	Text properties	Measure of textual features of review comments, such as stop word ratio	Rahman et al. (2017)
Review intensity	Code churn	Number of lines added and deleted between revisions	Thongtanunam et al. (2015a), Shimagaki et al. (2016)
	Interactions	Number of review iterations of a review request prior to its conclusion	Thongtanunam et al. (2015a)
Review time	Review delay	Time from the first review request submission to the first reviewer feedback	Thongtanunam et al. (2015a)
	Review duration	Time from the first review request submission to the review conclusion	Morales et al. (2015), Thongtanunam et al. (2015a), 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. (2015a)
MCR coverage	In-house	Ratio of internal contributions in the project	Shimagaki et al. (2016)
	Reviewed changes	The proportion of committed changes associated with code reviews	Morales et al. (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 et al. (2015), McIntosh et al. (2014), Shimagaki et al. (2016), Thompson and Wagner (2017), McIntosh et al. (2016)

with distributed teams, Santos and Nunes (2018) concluded that the team size decreases when more locations and teams are involved. Ruangwan et al. (2019) also found that an experienced reviewer with higher review participation rate is more likely to respond a review invitation, i.e. a reviewer who has been actively responding to a review invitation in the past is more likely to respond to a new invitation. Another investigated aspect is the impact of familiarity with the project code on the team size. Three studies (Kovalenko and Bacchelli, 2018; Lee and Carver, 2017; Thongtanunam et al., 2017) analyze 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 and 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 speeding 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 and Davenport, 2010; Murakami et al., 2017). One study (Murakami et al., 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 and 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 level of agreement among reviewers is investigated in two studies. Thongtanunam et al. (2015a) analyze the relationship between a file with prior defects and the level of review disagreement, but the results do not show that there is a relationship

Non-technical Influence Factors		# Papers	Number of Studied Relationships	
Author	Code Familiarity	7	TS FS IC DE DU AR PQ	19
	Development Experience	2	DU FD PQ	3
	Reputation	3	IC DU FD PQ	4
	Reviewing Experience	1	TS	1
Reviewer	Code Familiarity	7	TS FQ FS DE PQ	8
	Personal Characteristics	1	TE	1
	Reputation	2	IC DU FD	3
	Reviewing Experience	5	TA TS FQ DU FD PQ	6
	Review Participation Rate	1	TS	1
Other	Project Review Workload	4	TS FS DE DU FD PQ	7

Technical Factors				
Patch Properties	Code Legibility	1	TE	1
	Code Ownership	2	TS FS DE PQ	4
	Location	2	IC DU FD	3
	Scatteredness	4	TS FQ FS IC DE	8
	Size	8	TS FS IC IN DE DU FD PQ	16
	Type of Change	1	PQ	1
	Type of Files	1	FQ	1
Historical Patch Properties	Feedback Size	1	TS FS DE	3
	Team Size	1	TS FS DE	3
	Prior Defects	2	TA TS FS IN IC DE DU RS	11
	Recency of Change	1	TS FS DE	3
	Review Delay	1	TS FS DE	3
Other	Request Description Lengt	1	TS FS DE	3
	Reviewers Notification	1	TS IN DE DU RS PQ	6
	Task Classification	3	TS FS IC DE DU FD	6

Internal Outcomes As Influence Factors				
Reviewer Team	Code Ownership	2	FQ PQ	2
	Level of Agreement	2	FS DU PQ	3
	Team Closeness	4	TS FQ FS DU PQ	6
	Team Size	6	TS FS DU PQ	8
	Self-approved Changes	4	CQ PQ	4
Review Feedback	Feedback Size	7	CQ PQ	7
	Hastily Reviewed	5	CQ PQ	5
	No Discussion	4	CQ PQ	4
	Text Properties	1	FQ	1
Review Intensity	Review Code Churn	2	PQ	2
	Review Interations	1	PQ	1
Review Time	Review Delay	1	PQ	1
	Review Duration	4	CQ PQ	4
	Review Speed	1	PQ	1
MCR Coverage	In-house	1	PQ	1
	Reviewed Changes	5	CQ PQ	5
	Reviewed Churn	5	CQ PQ	5

Internal Outcomes:
TA Team Agreement Level
TE Team Effectiveness
TS Team Size
FQ Feedback Quality
FS Feedback Size
IC Review Code Churn
IN Review Interations
DE Review Delay
DU Review Duration
RS Review Speed
AR Acceptance Rate
FD Final Decision

External Outcomes:
CQ Code Quality
PQ Product Quality

Fig. 5. Studies that analyzed the relationship between influence factors and outcomes. On the left-hand side, there is the number of published studies that investigated the effects of an influence factor. On the right-hand side, there is the number of studies that investigated the effects of an influence factor over a particular outcome. A study can investigate the effect of an influence factor over more than one outcome.

between them. Hirao et al. (2016), in turn, examined the influence of reviewers' reviewing experience on the frequency of their votes that disagreed with the review conclusions. The findings suggest that more experienced reviewers are more likely to have a higher level of agreement than less experienced reviewers.

> **Finding 6**: There is an average of one or two reviewers by review request in open-source projects. Overall, small code changes and long descriptions of review requests are more likely to attract reviewers. Developers prefer to invite experienced reviewers.

Review feedback. The most investigated internal outcome is review feedback, with a total of 26 papers. These studies explored the comments made by reviewers. We identified three examined aspects: (i) content information (Bacchelli and Bird, 2013; Spadini et al., 2018; Efstathiou and Spinellis, 2018; Bosu et al., 2014; di Biase et al., 2016; Li et al., 2017; Ebert et al., 2017; Ebert et al., 2019; Paul et al., 2019; El Asri et al., 2019; Norikane et al.,

2018; Wang et al., 2019b; Pascarella et al., 2018; Zanaty et al., 2018; Ebert et al., 2018); (ii) size, typically in terms of amount of comments (Liang and Mizuno, 2011; Bosu and Carver, 2012; Thongtanunam et al., 2017; Lee and Carver, 2017; Kovalenko and Bacchelli, 2018; Santos and Nunes, 2018; Thongtanunam et al., 2015a; Hirao et al., 2016; Norikane et al., 2018; Jiang et al., 2019b); and (iii) quality, in terms of, e.g., usefulness (Rahman et al., 2017; Bosu et al., 2015).

To further understand what has been discussed within code reviews, there is research work that explored the nature of dialogues and the concerns raised by human reviewers. Bacchelli and Bird (2013) manually analyzed and classified MCR comments from Microsoft projects, creating categories for emerged themes. Spadini et al. (2018) reproduced this analysis using 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. Two studies (di Biase et al., 2016; Bosu et al., 2014) examined security concerns raised in the review feedback, leading to identified categories related to the domain and language-specific issues (di

Biase et al., 2016) and race conditions (Bosu et al., 2014). While the mentioned studies performed a manual analysis of the 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.

Zanaty et al. (2018) investigated design-related discussions in code reviews, concluding that this aspect is not commonly discussed. However, when design issues are raised in the review feedback, they are considered constructive, offering alternative solutions. Moreover, two studies (Pascarella et al., 2018; Ebert et al., 2018) specifically analyzed the questions and answers in the review feedback to identify the information that reviewers need and their communicative intentions. As contributions, both studies found that questions are used to ask an action of the author related to a suggestion of an alternative solution. The request of confirmation or clarification of the correct understanding were also topics found in these studies.

We also identified research work focused on confusion (Ebert et al., 2017; Ebert et al., 2019) and sentiments (Paul et al., 2019; El Asri et al., 2019) expressed by reviewers. With respect to confusion, Ebert et al. (2017) initially studied the feasibility of analyzing the confusion using linguistic feature, they then complement this investigation by exploring the reasons for the confusion and on how developers cope with it (Ebert et al., 2019). Several reasons were found for confusion, being most frequent the missing of rationale, discussion of non-functional requirements, and lack of familiarity with code. Concerning the topic of sentiments, El Asri et al. (2019) conducted a broad empirical study, while Paul et al. (2019) focused on the differences in expressions between male and female developers during code review. These studies observed differences depending on the position in the collaboration network and the gender of reviewers. While peripheral contributors have more outliers in expressing positive and negative sentiments, the core developers are neutral when commenting a review (El Asri et al., 2019). The results also suggest that females are less likely to express sentiments than males (Paul et al., 2019).

In addition to the discussed studies, two other works investigate specific aspects using the review feedback. Wang et al. (2019b) manually analyzed reviewed changes and classified the reasons for abandoning them in 12 categories. The top three categories are: (i) duplicate, in which are included changes that were abandoned because they were similar to others; (ii) lack of feedback, when authors did not respond the review feedback or when nobody responded the review request; and (iii) contributor operation, when occurs erroneous operations. Nevertheless, most changes are abandoned due to duplication. Norikane et al. (2018), in turn, analyzed the review feedback to understand what encourages a contributor to continue with an open-source project. As a result, the study observed similarities in the early contributions of those who become long-term contributors (LTC) and those who become short-time contributors (STC). In summary, those who become an LTC submitted more code changes and received more review feedback than STCs. Moreover, there are more similarities between the reasons for rejecting LTCs, while STCs received more scattered motives for rejection.

Focusing on the feedback size, two studies (Bosu and Carver, 2012; Liang and Mizuno, 2011) analyzed the amount of discussion by review in OSS projects and observed an average of two or three comments per review. When examining the review data of reopened pull-requests, Jiang et al. (2019b) found an average of two comments for non-reopened pull requests and seven comments for reopened requests. 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 (Kovalenko and Bacchelli, 2018; Lee and Carver, 2017; Thongtanunam et al., 2017; Hirao et al., 2016; Santos and Nunes, 2018; Thongtanunam et al., 2015a).

Three studies (Kovalenko and Bacchelli, 2018; Lee and Carver, 2017; Thongtanunam et al., 2017) examine the relationship between the familiarity with the project of the author submitting the change and the feedback size. 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 have a longer discussion length. Similarly to team size, the feedback size is also influenced by the patch size. Santos and Nunes (2018) 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. (2015a) also 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 et al. (2015) and Rahman et al. (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 et al. (2017) also found some variation among textual properties between useful and non-useful comments. Motivated by these two studies (Bosu et al., 2015; Rahman et al., 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.

> **Finding 7**: The most frequent discussion topics in the review feedback are related to code improvement, understanding, and social interactions. The reviewers also discuss alternative solutions related to design, and most of the suggestions are made using questions. Moreover, in the feedback, reviewers express confusion, often due to missing rationale, discussion of non-functional requirements, and the lack of familiarity with code.

> **Finding 8**: In open-source projects there is an average of two or three review comments per request. Newcomers tend to receive more attention. The reviewer more familiar with the code is more likely to provide useful feedback.

Review intensity. We now 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 and Carver, 2014; Baysal et al., 2016; Liang and Mizuno, 2011; Thongtanunam et al., 2015a; Armstrong et al., 2017; Ueda et al., 2017). Two (Ueda et al., 2017; Baysal et al., 2016) investigate the content of the code churn, while those remaining make an analysis of the correlation between influence factors and the review intensity.

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 code churn, the 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) explored technical aspects but did not find a strong correlation between patch content and 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 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. (2015a), 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 et al. (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).

Finding 9: Most of the problems fixed during code review are related to evolvability and are triggered by review feedback. Small code changes and files with prior defects tend to undergo few rounds of code review.

Review time. *Time* is another aspect of the MCR process that has been investigated. Studies focus 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, 14 papers targeted this topic. Four of them (Yang et al., 2017; Bosu and Carver, 2012; Thongtanunam et al., 2015b; Kerzazi and El Asri, 2016) characterize the review interval in particular contexts. The other papers explore 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 El Asri (2016) examined both technical and socio-technical interactions among contributors in code review. They identified behavioral patterns, reporting that core developers are more likely to have a shorter review interval than peripheral developers (Kerzazi and El Asri, 2016). Lastly, Thongtanunam et al. (2015b) 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 require, on average, 12 days longer to approve code changes.

In addition to the discussed findings on review duration, many studies (Santos and Nunes, 2018; Bosu and Carver, 2014; Thongtanunam et al., 2015a; Kovalenko and Bacchelli, 2018; Lee and Carver, 2017; Armstrong et al., 2017; Baysal et al., 2016; Hirao et al., 2016) explore which factors might be related to the total amount of time taken to reach a review decision. Although some 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 and Carver, 2014; Lee and 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 and Nunes, 2018); (v) there are few prior defects in the reviewed files (Thongtanunam et al., 2015a); (vi) there are few active reviewers, and they are from the same team and location (Santos and Nunes, 2018); (vii) the historical level of agreement of involved reviewers is high (Hirao et al., 2016); and (viii) the review feedback contains positive sentiments (El Asri et al., 2019). Moreover, Jiang et al. (2019b) analyzed characteristics of reopened pull-requests and their impact on code review, reporting that these requests take more review time to be concluded.

Other five studies (Bosu and Carver, 2014; Thongtanunam et al., 2015a; Kovalenko and Bacchelli, 2018; Armstrong et al., 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. The key findings are that authors of code changes that are more familiar with the project receive faster first feedback (Bosu and Carver, 2014). In contrast, if a patch has files with prior defects (Thongtanunam et al., 2015a) 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. (2015a) investigated whether the speed varies depending on the presence of defects in a prior release, while Armstrong et al. (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.

Finding 10: The review request takes less time to be completed when involving experienced authors familiarized with the code, experienced and active reviewers with a high level of agreement. Moreover, the review duration reduces when there is a short review queue, the code change is small, the files have few prior defects, and the review feedback contains positive sentiments.

Review decision. The last group of influence factors refers to the result of code reviews (accept or reject), that is, the review conclusion. This is one of the least explored topics with nine identified papers (Baysal et al., 2016; Bosu and Carver, 2014; Lee and Carver, 2017; Kovalenko and Bacchelli, 2018; Bosu and Carver, 2012; Bosu et al., 2014; Hirao et al., 2015; Jiang et al., 2019b; El Asri et al., 2019). The observed findings are scattered.

Hirao et al. (2015) investigated how many review requests followed the simple majority method of voting to decide on the acceptance or rejection of code changes. With 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 the requests are not accepted, respectively. Bosu et al. (2014), in contrast, focused on uncovering the changes containing which types of vulnerabilities are more likely to be abandoned. They 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.

Lastly, there are few studies exploring the factors influencing the review outcome and acceptance rate. Baysal et al. (2016) analyzed multiple factors and concluded that the review outcome is most affected by author development experience. The findings of El Asri et al. (2019) also indicate that negative review comments share a relationship with the likelihood of an unsuccessful review. Similarly to Baysal et al. (2016), three studies (Bosu and Carver, 2014; Kovalenko and Bacchelli, 2018; Lee and Carver, 2017) focus on the author's familiarity with the project, with results indicating that the acceptance rate is lower for newcomers. In addition, Jiang et al. (2019b) found that reopened pull requests have lower acceptance rates in the code review.

> **Finding 11**: The accepted code changes are usually authored by experienced developers and contain review feedback with positive sentiments. Considering the review feedback analysis, the main reasons to abandon a code change are duplication and lack of feedback.

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 process. However, these are not properties that are externally perceived (e.g., how intense the discussion is). We now discuss work that focus on external outcomes, which are mainly improvement of code quality (design and programming practices) and product quality (reduction of defects). We identified 13 papers that explored quantitative data collected from repositories, not only with code review data but also the code being reviewed. From these papers, only one (Morales et al., 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 et al., 2017; Thongtanunam et al., 2015a; Thompson and Wagner, 2017; Bavota and Russo, 2015; McIntosh et al., 2014; di Biase et al., 2016; An et al., 2018) on product quality.

Morales et al. (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 indicate 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; Thompson and Wagner, 2017; di Biase et al., 2016) analyze OSS projects to investigate factors influencing security aspects of code that went through code review. di

Biase et al. (2016) presented a case study of security aspects of 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) reports 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. Bavota and Russo (2015) identified that unreviewed code changes have over two times more chances of inducing bugs than 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. Despite this positive impact of reviewing practices on software quality, Kononenko et al. (2015) identified that 54%–56% of code reviews missed bugs.

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). Analyzing review practices, Thongtanunam et al. (2015a) 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. (2015a) 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 in the work of Meneely et al. (2014), which shows that vulnerable files tend to have more involved reviewers. Additionally, Armstrong et al. (2017) observed that review using unicast communication technology has fewer defects than broadcast communication.

An et al. (2018) focused on crashed reviewed code, a type of defect with severe implications. The goal is to understand why reviewed change still crashes. As result, they found that those reviewed changes are usually related to performance, refactoring, fix previous crashes, and new functionality. Moreover, they found that the crashes are mainly motivated by memory and semantic errors, which were not detected during code review.

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 OSS 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.

> **Finding 12**: The product quality tends to increase when code changes are reviewed. More participation in code reviews, more experience with the reviewing activity, and reviewers more familiar with the project are also factors that reduce the likelihood of post-release defects.

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, some researchers explored the *behavior* of reviewers during code review, how they check the changed code, and their willingness to participate in MCR. Nine studies go to this direction: (i) three (Begel and Vrzakova, 2018; Uwano et al., 2006; Chandrika et al., 2017) rely on eye tracking; (ii) two (Floyd et al., 2017; Duraes et al., 2016) rely on functional magnetic resonance imaging (fMRI); and (iii) the remaining four (Kitagawa et al., 2016; Dunsmore et al., 2000; Baum et al., 2019; Spadini et al., 2019) analyze the reviewer behavior with simulation and experimental data.

Studies that used eye tracking (Begel and Vrzakova, 2018; Uwano et al., 2006; Chandrika et al., 2017) aim 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 et al. (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 et al., 2017; Duraes et al., 2016) aim to understand the brain activity of reviewers to identify patterns for analysis. One of the studies (Floyd et al., 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) focuses on brain activity patterns when the reviewer identifies a bug. 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 particular aspects of the review behavior. Kitagawa et al. (2016) aimed to understand the reviewer participation 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 study (Dunsmore et al., 2000) is an empirical investigation of defect detection in object-oriented (OO) programs, concluding that defects that require information spread throughout the software to be identified are hard to find and the OO code structure favors this type of defect. Baum et al. (2019) then experimentally analyzed the association between aspects of the cognitive load of reviewers and their performance. As key findings, they found a correlation between working memory capacity and the reviewer's effectiveness of finding delocalized defects, as well as evidence of the negative impact of larger and complex code changes on review performance. Finally, Spadini et al. (2019) explored the influence of the order that the test code is presented to the reviewer, i.e. test then production code or production then test code. Based on a controlled experiment, they observed that reviewers in a test-first review find the same proportion of defects in production file and more defects in test code.

> **Finding 13**: The programming experience might influence reviewers' attention and brain work during the code checking. Specific brain regions associated with language processing and mathematics have increased activity when reviewing a code, and experienced developers tend to perceive programming languages like a natural language.

4.6. Relationship with MCR

The FOUNDATIONAL STUDIES discussed previously focus 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

There are experiments that compare MCR with verification techniques, namely pair programming (Müller, 2005; Swamidurai et al., 2014) and testing (Runeson and Andrews, 2003). They involved only students, and the overall goal is to examine the effectiveness of one technique with respect to another.

Müller (2005) and Swamidurai et al. (2014) compared code review and pair programming aiming to verify which has a higher impact in terms of cost. In both studies, the participants are divided into the ones using pair programming to execute a task, and those who work individually with the assistance of a code review phase. The difference between the two studies is that in Swamidurai et al.'s experiment (Swamidurai et al., 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 et al. (2014) found evidence that programs with similar quality can be produced using peer review with 28% lower cost than using pair programming.

While the comparisons with pair programming focused on the cost, there is a study that compares testing practices 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, they reported differences, being code review more effective in terms of time spend and isolation, and testing finds more failures (Runeson and Andrews, 2003).

Considering the discussed findings, we highlight that some 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 and Bird, 2013) and tool-support became popular (Bacchelli and 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.

> **Finding 14**: MCR and pair programming are interchangeable in terms of cost, except when the latter is adopted within test-driven development—in this case, MCR has lower cost. Unit testing finds more failures than MCR, but the latter requires less time in the detection and isolation of the underlying sources of the defects.

4.6.2. Interaction with development practices

The last group of FOUNDATIONAL studies has five analyses involving MCR and its relationship with other software development practices. Three studies focus on quality, verifying its relationship with continuous integration (Rahman and Roy, 2017; Zampetti et al., 2019) and static analysis (Panichella et al., 2015). One study investigates the intent and awareness of developers when performing changes concerned with architectural aspects (Paixao et al., 2019). Finally, the fifth study explores code review and its association with code ownership (Thongtanunam et al., 2016). All these five works examine code review data from OSS projects.

The interplay between reviewing practices and continuous integration (CI) has been investigated in two studies (Rahman and Roy, 2017; Zampetti et al., 2019), both exploring builds from the CI process. Rahman and Roy (2017) focused on the influence of the status and frequency of these builds on code review, while Zampetti et al. (2019) targeted build failures and how these builds are discussed during code review. The key findings of these investigations indicate that passed builds and frequently built projects are more likely to encourage the reviewers' participation. In general, the status of a build influences the acceptance of a review request, but there are exceptions, and failed builds are sometimes accepted. Moreover, they found many review discussions on difficulties in configuring the CI pipeline.

Panichella et al. (2015) examined what has been changed in the code during a review to understand how 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. (2019) mined review data to investigate the intent and awareness of developers of the architectural impact of their changes, and also how architectural changes evolve during code review. As result, only 31% of the examined reviews with a noticeable impact on the architecture have a conversation related to such impact, which suggests

a lack of awareness when such type of modification is performed. However, Paixao et al. (2019) also found that developers tend to be more often aware of the architecture when the change is related to it. In reviews in which the architecture is discussed, there is a trend to have larger improvements in cohesion and coupling.

The fifth 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 the review activity should be used in the code ownership heuristics. By examining data of two OSS 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.

> **Finding 15**: Few warnings raised by static analysis tools are removed during code review. Contexts relying on continuous integration, with frequently built projects and automated builds with the passed status, are related to increased reviewers' participation.

4.7. RQ-1 - What foundational body of knowledge has been built based on studies of MCR?

The foundational body of knowledge of MCR consists mainly of evidence about the practice in real settings. The adoption and execution of MCR are influenced by multiple factors, being frequent findings related to developers' experience. The experience with other teams, projects, products, tools, and familiarity with the source code is reflected in the decisions that shape the review process, internal outcomes, and how reviewers behave when checking the code. MCR is adopted using different tools and rules in both industry and OSS. Despite the MCR variants, there are convergent refinements that remain over the years, such as the small number of reviewers and review comments per review request. As expectations, the review practitioners desire a positive impact on the code quality, knowledge sharing, and learning. The FOUNDATIONAL STUDIES provide evidence that MCR increases the code quality and promotes discussions about quality aspects, security, and architecture. Moreover, MCR influences the peers' perception, the developers' role in open-source projects, and code ownership. The foundational body of knowledge of MCR is commonly based on data analysis from historical information of open-source repositories, but there are also studies in large companies.

5. PROPOSALS

The previously discussed studies provide an understanding of how MCR works, deriving knowledge that is helpful to develop novel approaches to support MCR. This section introduces 53 PROPOSALS of a technique, tool, or theory that aim at improving the MCR practice. These are grouped into three categories, shown in Table 8. The first two categories are related to the two phases of MCR described in Section 2, namely *Review Planning and Setup* and *Code Review*. The third, *Process Management and Support*, includes approaches that focus on aiding the MCR process by providing guidance, data analysis or tool support. We next discuss the identified PROPOSALS.

Table 8
Classification of MCR Proposals.

Category	Proposal goal	#Proposals
Review planning and setup		**18**
Patch documentation	Assist the preparation of the review request	1
Reviewer recommender	Recommend or automate the selection of reviewers	14
Review prioritization	Support the prioritization of review requests	3
Code review		**19**
Code checking	Support the activity of checking the code performed by reviewers	15
Feedback Provision	Support the activity of providing feedback to authors	2
Review decision	Support deciding whether there is a need for further review	2
Process management and support		**16**
Methodology and guidelines	Provide a taxonomy or guidelines to support MCR	4
Review retrospective	Assess or predict high-level MCR outcomes	6
Tool support	Presentation of tools to support the MCR lifecycle	7

5.1. Review planning and setup

We identified three kinds of support associated with the *Review Planning and Setup* phase, i.e. before reviewers review the code. 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*: aiding the selection of suitable reviewers; and (iii) *review prioritization*: helping reviewers to select code reviews to be performed earlier.

5.1.1. Patch documentation

A single approach (Hao et al., 2013) is dedicated to support patch documentation. It consists of a tool—named Multimedia Commenting Tool (MCT)—implemented as an Eclipse plug-in. MCT allows programmers to include code narration and embedded multimedia resources, as well as support the replay of these comments. Thus, reviewers can reproduce them, which might help the understanding of code changes. MCT works with multimedia comments that contain audio or a video clip, a file recording mouse movements, a copy of the source file when the comment is created, and optionally extra files due to code changes.

5.1.2. Reviewer recommender

Reviewer recommenders consist of tools and underlying techniques suggesting a list of the best candidates to review a review request. The motivation of most techniques is to reduce of the time taken for a review acceptance. There are also techniques whose goal is to support newcomers to reach out experienced developers (Zanjani et al., 2016; Rahman et al., 2016), to find the best candidates when the review request involves multiple files and large changes (Ouni et al., 2016; Xia et al., 2017), or to quickly perform recommendations at scale (Asthana et al., 2019). 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 et al., 2016; Xia et al., 2017; Fejzer et al., 2018; Jiang et al., 2019a). A single study proposes a recommender with load balancing to mitigate the effect of unbalanced recommendations (Asthana et al., 2019).

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, 2015b) then suggested the use of *file patch similarity*, which takes into account previous changes with similar paths and who reviewed them. Their second work presented RevFinder (Thongtanunam et al., 2015b) was adopted as a baseline technique in the evaluation of most of the approaches proposed posteriorly. Other techniques explore a wide range of additional features to improve code recommenders, they are: (i) the *review description* and its similarity with past descriptions (Xia et al., 2015); (ii) the *review feedback* written by potential reviewers (Yu et al., 2016; Zanjani et al., 2016); (iii) common *interests* among reviewers (Ouni et al., 2016; Xia et al., 2017; Liao et al., 2019); (iv) cross-project experience in specialized *technologies* (Rahman et al., 2016); and (v) *topic models* using review request information and reviewers' influence (Liao et al., 2019). Part of this information complemented those previously explored. Xia et al. (2015) extended RevFinder, while other approaches (Ouni et al., 2016; Rahman et al., 2016) include the expertise with files to be reviewed. Asthana et al. (2019) then prioritized who either committed changes or reviewed the files in the review request of a project. Instead of proposing a new technique, other studies investigated the effectiveness of different features to identify the best set of reviewers. 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 et al. (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.

5.1.3. Review prioritization

Given that reviewers might be invited to many code reviews, there is a need for prioritization. Thus, the approaches in this group 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. The approach estimates the bug-proneness and the cost required for review each file. The goal is to find an optimal selection of files to be reviewed that does not exceed a certain cost while maximizes the sum of bug-proneness in this set. Fan et al. (2018), instead, suggested the estimation of the *chances for a change to be accepted or rejected* in the code review process. First, a model building phase takes as input a set of labeled changes, extracts 34 features, and train a prediction model. In a prediction phase, this model can be applied to estimate if a change is going to be accepted. The idea is to give higher priority to high-quality changes. Lastly, Wen et al. (2018) suggested directing the review effort considering the *impact on the project deliverables*. The proposed tool, BLIMP Tracer, first extracts the building dependency

graph of the system. Then, given the review request, the tool recursively traverses the graph to identify the impacted product deliverables. The output is an impact analysis report and the idea, in this case, is to give higher priority to changes impacting critical project deliverables or deliverables that cover a broad set of products.

Finding 16: Approaches to support authors of code changes focus mainly on the *Review Planning and Setup* phase of MCR (Fig. 1). Reviewer recommender is the most common type of support for the MCR process (14 of 53 PROPOSALS), to help the author select reviewers. Most of these techniques use a review (or development) history to build a recommendation. For the *Preparation* task, there is tool support that allows authors to include multimedia comments on the source code, which might help reviewer understanding.

5.2. 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 split into three categories—code checking, feedback provision and review decision—which are discussed next.

5.2.1. Code checking

Support for code checking has been focused on providing information to ease the manual work of reviewers, mainly the understanding of a change. To provide this support, the main strategies presented in the studies are (i) provide visualizations of code changes (Barnett et al., 2015; Luna Freire et al., 2018; Tao and Kim, 2015; Ge et al., 2017; Zhang et al., 2015; Duley et al., 2010; Huang et al., 2018b; Wang et al., 2019a; Guo et al., 2019; Huang et al., 2018a), (ii) present properties associated with them (Menarini et al., 2017; Tymchuk et al., 2015; Mishra and Sureka, 2014), and (iii) support the analysis of change impact (Wang et al., 2017; Hanam et al., 2019).

Some PROPOSALS assume that a change can be decomposed to ease understanding. ClusterChanges (Barnett et al., 2015), JClusterChanges (Luna Freire et al., 2018), CoRA (Wang et al., 2019a), and ChgCutter (Guo et al., 2019) are techniques for decomposing changes based on static analysis. ClusterChanges and JClusterChanges, instantiated for C# and Java language, respectively, analyze composite changes to uncover definitions and their uses, clustering diff-regions into trivial and non-trivial. Similarly, CoRA analyzes dependency relationships and similarity in tangled changes, further generating a description of each partition based on its importance, templates, and program analysis technique. ChgCutter, in turn, partitions a user-selected change subset based on dependencies and builds a compilable intermediate program version, in which a subset of regression tests can be run. In addition to this techniques, 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 type. 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. ISC (Huang et al., 2018b) focuses on identifying a salient class in a review request, which is a modified class that causes the modification of the remaining classes in the request. The problem is modeled as a binary classification using a large number of discriminative features. ClDiff (Huang et al., 2018a) then works to generate concise linked code differences exploring the abstract syntax tree (AST). As output, ClDiff provides a visualization of code differences grouped by the links found and also a description of each group. The other more specific approach consists of a differencing algorithm, named Vdiff (Duley et al., 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. Other approaches that aim to support code checking target on the software behavior, quality, change impact analysis, and effort. Getty (Menarini et al., 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 et al., 2015) gives a city-based code visualization to help reviewers 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. SemCIA (for JavaScript) and MultiViewer are assistance tools for change impact analysis. The former implements novel semantic relations and shows relationships between structural changes and changes to program behavior. MultiViewer (Wang et al., 2017), in turn, includes the formal definition of three metrics: effort, risk, and impact. It 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, Mishra and Sureka (2014) provide an estimation model for code review. Six variables to measure the size and complexity of the modified files are used to help reviewers predict the work needed to review a code change.

5.2.2. Feedback provision

In addition to analyze code changes, reviewers must be able to provide feedback to authors, which can be, e.g., request for changes, ask questions and clarification, or votes of acceptance or rejection. Two approaches have the goal of supporting this feedback provision. Rich Code Annotation (RCA) (Priest and Plimmer, 2006) is a digital ink tool integrated with the development environment to support annotations in reviews, so that reviewers can provide feedback using multimedia resources. The other approach consists of a prediction model—RevHelper (Rahman et al., 2017)—to indicate the usefulness of review comments in review submissions. This model was built on top of studies (Bosu et al., 2015; Rahman et al., 2017) (discussed in Section 4) on the usefulness of reviews comments (one of them published in the same paper in which RevHelper was proposed).

5.2.3. Review decision

After going through a round of review, a code change can be accepted, rejected or may need rework, possibly requiring further reviews. To help reviewers reach 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) 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 et al. (2016), in turn, proposed a prediction model similar to typical bug predictors, which builds 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.

> **Finding 17:** Most MCR approaches focus on supporting reviewers, being 22 of 53 PROPOSALS (41.5%) related to the reviewers' work, i.e., *review prioritization*, *code checking*, *feedback provision*, and *review decision*. From these 22 PROPOSALS, 15 target the *code checking* task, aiming to help understand code changes, which is a challenge for practitioners, as identified in FOUNDATIONAL STUDIES. In 66.7% of cases (10 of 15 studies), this checking support consists of providing alternative visualizations of code changes.

5.3. 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.

5.3.1. Methodology and guidelines

Existing works classified as methodology and guidelines 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. (2016b) and Li et al. (2017). The former presented a faceted classification scheme for industrial MCR processes, including variations on how the process is embedded, reviewers aspects, code checking, feedback, and overarching facets. 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).

Two works proposed guidelines. The first (Baum and Schneider, 2016) makes recommendations associated with tool support, suggesting improvements in code review tools to increase review efficiency and effectiveness. The second study (Baum et al., 2017b) 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.

5.3.2. Review retrospective

MCR repositories can be explored to improve its process in particular projects. Six works went to this direction. Uwano et al. (2006) presented an integrated environment to capture eye movements during the code review, the Crescent tool. The approach 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 approaches 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 et al. (2015) then proposed a classification based on eight comment attributes, such as the number of participants and comments in a thread, and the number of iterations in the

review. These attributes were defined based on the findings of a preliminary exploratory study, in which developers reported 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 are 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 hybrid classification of review topics. Grounded on its FOUNDATIONAL STUDY, the approach classifies the contents of review comments, allowing them to identify what reviewers are talking about. The intent is to organize the process and optimize the review tasks.

5.3.3. 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 et al., 2005; Lanubile and Mallardo, 2002; Zhang et al., 2011; Perry et al., 2002; Sripada et al., 2016). Table 9 summarizes the approaches identified in our SLR (ordered by their publication date), their ultimate goal, and strategies 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 (Sripada et al., 2016) 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 enable a more lightweight practice. This is the case of IBIS (Lanubile and Mallardo, 2002) and HyperCode (Perry et al., 2002), which can be used within a flexible and asynchronous review process. In contrast, recent tools have been focusing on more specific goals, aiming to address specific concerns of MCR, such as SmellTagger for tablets (Müller et al., 2012).

5.4. RQ2 - What approaches have been developed to support MCR?

Multiple approaches to support MCR have been proposed. Most of them consist of reviewer recommenders and techniques to provide visualizations of code changes. Reviewer recommenders focus on finding suitable candidates to review a code change, commonly to reduce the time taken for a review acceptance. These recommenders are usually based on historical data of code review and consider the file path similarity or common interests among reviewers in the past to rank and suggest the reviewers' candidates to a new review request. Approaches that provide visualizations of code changes focus on helping the reviewers during code checking, highlighting change aspects to ease the understanding and the finding defects. The main strategy to provide visualizations is to decompose a code change based on static analysis, presenting the differences between the old and new versions of the decomposed code.
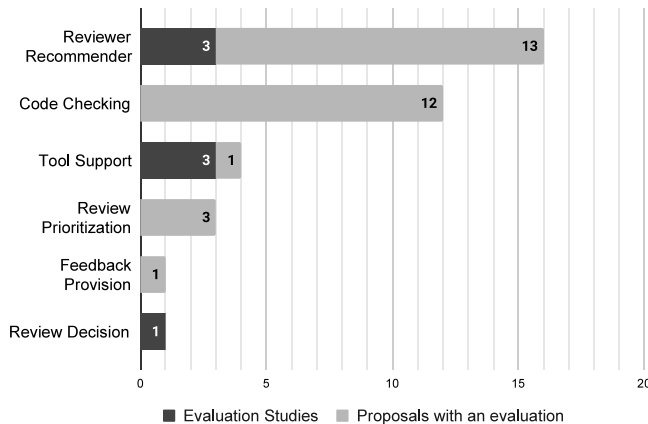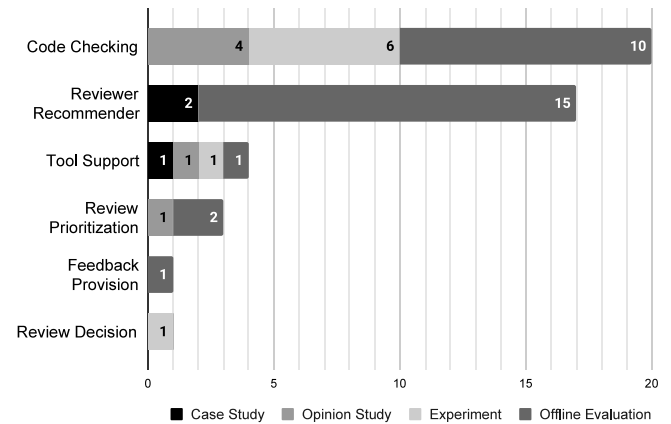
6. EVALUATIONS

We now focus on how the approaches to support MCR tasks have been evaluated either individually or compared to a baseline. We first introduce the types of evaluation that appeared either in papers that include a PROPOSAL (discussed in the previous section) or in those that has as main contribution an evaluation of existing approaches. Then we detail design aspects of performed evaluations, followed by a discussion of their key findings.

Table 9
Summary of the approaches that provide tool support for MCR.

Approach	Year	Main goal	Tool style
Support code review			
IBIS (Lanubile and Mallardo, 2002)	2002	Support to distributed code inspections	Web-based code inspection tool
HyperCode (Perry et al., 2002)	2002	Support to distributed code inspections	Web-based code inspection tool
Nagoya et al. (Nagoya et al., 2005)	2005	Support to the function-path review method	Desktop-based tool
Java Sniper (Zhang et al., 2011)	2011	Promotion of collaborative code review	Web-based code review tool
SmellTagger (Müller et al., 2012)	2012	Improvement in desirability and collaboration	Tool for tablet
Fistbump (Kalyan et al., 2016)	2016	Overcoming of limitations of existing tools	Web-based tool integrated with GitHub
Support the development of review tools			
Sripada et al. (Sripada et al., 2016)	2016	Increase in the developers' interest	Extensible framework for gamification

Fig. 6. Number of EVALUATION papers per type of MCR approach.

Fig. 7. Number of studies categorized by evaluation type and MCR approach type.

6.1. Evaluations types

Considering the identified PROPOSALS, we investigated whether they were evaluated in the paper that they were proposed. From the 53 PROPOSALS, 30 (56.6%) include an evaluation. We further identified seven papers whose main contribution is one or more evaluations. As result, there are 37 papers labeled as containing an EVALUATION. Fig. 6 shows the number of publications containing a PROPOSAL with an accompanying evaluation and evaluation studies. Most of these studies aim to assess a reviewer recommender technique, which is the most frequent type of approach. Typically, a newer approach is compared to the current state-of-the-art approach to demonstrate that it provides improvements at least in one aspect. Any form of evaluation of approach to support MCR was considered. We did not, however, consider as an evaluation a description of a scenario to illustrate the use or the benefits of a PROPOSAL, that is, when there is solely an example of its use made by its own authors; or the report of informally received feedback. This only provides anecdotal evidence of the effectiveness of the proposed approach.

Studies comparing MCR outcomes with other techniques, e.g. pair programming, are not included in this section because they are considered FOUNDATIONAL STUDIES (Section 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 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.

We classified the adopted research methods of EVALUATIONS into four main groups, as described in Table 10. The sum of the studies per type is higher than the number of papers because

there are papers that include more than one type of evaluation study. The most common is *offline evaluation*, being adopted in more than half (64.64%) 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 Fig. 7, we further detail the adopted evaluation type by the categories of MCR approaches. As can be seen, almost all reviewer 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, reported in seven papers (15.91%). These studies are characterized by a controlled environment, which involves participants and measured variables to analyze the effect of an intervention in the code review process. In contrast to experiments, *opinion studies* have none or limited control, being the participants invited to a hands-on trial to try the proposed tool or technique. From this interaction with the proposed approach, the researchers collect the user perception using interviews or questionnaires. Therefore, the evaluation is based on collected subjective data. Studies of this type are present in six (13.64%) papers. From these, four (Barnett et al., 2015; Zhang et al., 2015; Guo et al., 2019; Wang et al., 2019a) were used to complement the results of another study detailed in the same publication. One study (Müller et al., 2012) consists of a preliminary evaluation in which the subjects interact with a prototype of the proposed approach. Then, in another opinion study (Wen et al., 2018), the participants were invited to a semi-structured interview, in which they used the proposed approach and provided feedback. Lastly, *case studies* collect

Table 10
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.	29
Experiment	Empirical study in controlled settings to observe the effects of an MCR approach.	7
Opinion study	Subjective (qualitative or quantitative) evaluation of an MCR approach by subjects, after introducing the approach and allowing participants to experiment it.	6
Case study	Observation and collection of data from the instantiation of the approach in real settings, possibly using mixed research methods.	3

and analyze data from a particular non-controlled environment. Three studies fall into this category, two of them based on OSS projects, and one from industry. 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 and 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. Finally, one case study (Asthana et al., 2019) involved the deployment of the proposed approach in five repositories, following a collection of metrics to evaluate the usage and also a user study to analyze improvements. Later, this last case study was also complemented with an offline evaluation.

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

6.2. Offline evaluations

Focusing on 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.

6.2.1. Study design

Study object. From the 29 offline evaluations, most (51.72%) focuses on reviewer recommenders. Approaches to support code checking are the second most common (34.48%) (Barnett et al., 2015; Tao and Kim, 2015; Duley et al., 2010; Ge et al., 2017; Luna Freire et al., 2018; Huang et al., 2018b; Wang et al., 2019a; Hanam et al., 2019; Guo et al., 2019; Huang et al., 2018a). 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 et al., 2017) and a variation in the MCR process (Baum et al., 2016a).

Reviewer recommenders are usually evaluated using historical datasets, which are used to train and test generated models. The 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. The approaches are applied to existing changesets, and through a manual inspection the correctness of the output is verified, e.g., whether generated change clusters are acceptable. The manual inspection of the output followed two strategies, namely with and without a ground truth. Five approaches (Tao and Kim, 2015; Duley et al., 2010; Huang et al., 2018b; Wang et al., 2019a; Guo et al., 2019) to manipulate changeset visualizations 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 et al. (2010) and Guo et al. (2019) did not mention an external member in this step. Huang et al. (2018b) and Wang et al. (2019a), in turn, involved only external students in analyzing the existing data to manually establish the ground truth. In the three other studies, the output visualization was manually scrutiny by the 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. Lastly, in two evaluations (Hanam et al., 2019; Huang et al., 2018a), the output of the proposed approach is compared to another tool. Four of these papers related to support code checking (Barnett et al., 2015; Tao and Kim, 2015; Guo et al., 2019; Huang et al., 2018a) include studies with human subjects to complement the offline evaluation.

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 et al. (2017) proposed prediction models. Consequently, their evaluations consist of comparisons with other baselines. Moreover, Rahman et al. (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. (2016a) 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 11 descriptive statistics of the number of the projects analyzed in each study and whether these projects are open source or proprietary. Two studies (Rahman et al., 2016; Zanjani et al., 2016) are taken into account in both table rows as they used data from both types of projects. One study Baum et al. (2016a) 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.

The majority of the studies collected data only from OSS 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. Three offline evaluations (Yu et al., 2016; Peng et al., 2018; Huang et al., 2018b) using open-source data did not inform which repositories they mined. Only four studies (Balachandran, 2013; Barnett et al., 2015; Rahman et al., 2017; Asthana et al., 2019) used data solely from industry.

With respect to the number of projects, most of the studies had as target only a few projects—the median is 4 projects that are open source and 3 projects that are proprietary. As an outliers, Yu et al. (2016) and Huang et al. (2018b) used data from 84 and 120 projects, respectively. 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.

Table 11
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	24	13.29	28.22	4	1	120
	Proprietary Projects	6	3.83	3.43	3	1	10
Experiment	Subjects	8	34.25	60.21	13	8	183

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 et al., 2016; Fejzer et al., 2018; Asthana et al., 2019; Liao et al., 2019) 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 et al., 2016; Thongtanunam et al., 2015b; Ouni et al., 2016; Xia et al., 2015; Fejzer et al., 2018; Wang et al., 2019a; Jiang et al., 2019a). In particular, Jiang et al. (2019a) evaluated their proposed recommender considering the model construction time and prediction time.

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 assess its impact; (iii) Fejzer et al. (2018) examined the memory footprint; (iv) Baum et al. (2016a) used specific heuristics for quality, efficiency and cycle time; and (v) Hanam et al. (2019) analyzed the number of correct dependencies created by the proposed approach in comparison to others. Lastly, Wang et al. (2019a) evaluated the clustering result using Rand Index and Huang et al. (2018a) examined the generation of the proposed visualization using accuracy, conciseness, and time performance.

6.2.2. Findings

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

Reviewer recommenders. Most of the papers on reviewer recommenders (15 out of 16) report results of an offline comparison between a proposed approach and selected baselines. Consequently, the main result is an evidence that indicates that the proposed approach is better than an existing one (selected baseline) according to a selected metric. Typically, the studies consider the top-k recommendations (where k is the number of recommended reviewers). We detail in Table 12 the reported results when a new approach is compared to another algorithm of reviewer recommendation, showing which approach outperformed which baseline according to which metric.

Other studies—presented in Table 13—compared a reviewer recommender with alternative baselines, e.g. a simple heuristic or a standard learning technique. For instance, Balachandran (2013) developed RevHistRECO, which is a heuristic inspired by the observed manual process of the reviewer assignment, adopted as a baseline for his proposed Review Bot. Yu et al. (2016), in turn, extended and implemented as baseline recommenders based on existing techniques, such as information retrieval (IR). Some of the baseline approaches in this table, e.g. xFinder (Kagdi et al., 2008) and CoreDevRec (Jiang et al., 2015), are not in 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 to similar approaches, eight out of ten offline studies (Barnett et al., 2015; Tao and Kim, 2015; Duley et al., 2010; Ge et al., 2017; Luna Freire et al., 2018; Huang et al., 2018b; Wang et al., 2019a; Guo et al., 2019) that evaluate work to support code checking use a manually built ground truth. Hanam et al. (2019), however, used additional tools to build a ground truth. In both cases, 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 Luna Freire et al. (2018) highlighted which type of code change was not considered by their approach to distinguish trivial and non-trivial changes. The other offline evaluation (Huang et al., 2018a) of a code checking approach then mixed the manual analysis of output and the comparison with an existing tool. As key findings, Huang et al. (2018a) found that the proposed approach has higher accuracy and better conciseness as well as required less time in the study.

> **Finding 18:** Evaluations of approaches to support MCR mainly focus on the validation of the output using data analysis. This strategy—named *offline evaluation*—is usually used when assessing reviewer recommenders and code checking support. Most offline evaluations use data from OSS projects. Precision and recall are the most frequent metrics in this type of evaluation. Considering reviewer recommenders, RevFinder has been the most used as a baseline.

> **Finding 19:** Most conclusions of the EVALUATION studies are related to the feasibility and effectiveness of an approach to support MCR based on data analysis. The results of the evaluation of a newly-developed proposal outperforms baselines, achieving better accuracy, precision, and/or recall.

6.3. Experiments

A smaller amount of MCR approaches (in comparison to 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 also summarize their findings.

6.3.1. Study design
Variables of the experiments. The experimental studies to evaluate an MCR approach rely on various variables. Five 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. Similarly, Hanam et al. (2019) asked the study participants to perform a review task supported

Table 12
Results of comparisons of code reviewer recommenders. 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).

Approach	Outperformed reviewer recommenders							
	Review Bot	FPS	RevFinder	TIE	cHRev	IR+CN	Activeness	WRC
FPS (Thongtanunam et al., 2014)	ACC							
RevFinder (Thongtanunam et al., 2015b)	ACC							
TIE (Xia et al., 2015)			ACC					
Correct (Rahman et al., 2016)			ACC, P&R, MRR					
cHRev (Zanjani et al., 2016)			P&R					
RevRec (Ouni et al., 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 et al. (2018)	P&R		P&R					
TRFPre (Jiang et al., 2019a)				ACC	ACC			ACC
WhoDo (Asthana et al., 2019)					P&R			

Table 13
Results of comparisons between a reviewer recommender and a baseline technique. Each row indicates that an approach in the first column outperformed baselines in the third column, with respect to the metrics listed in the second column. The metrics are accuracy (ACC), F-measure (F1), mean reciprocal rank (MRR), and precision and recall (P&R).

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 et al., 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
TRFPre (Jiang et al., 2019a)	ACC	CoreDevRec and ACRec

by the proposed change impact analysis tool SemCIA and also by other tools, SynCIA and UnixDiff. In two studies, the participants reviewed a code change with and without the support of the proposed approach output, i.e. with or without partitions (Tao and Kim, 2015) and with or without a hint of salient class (Huang et al., 2018b). Lastly, participants of the study of Huang et al. (2018a) conducted the review with the proposed tool ClDiff and with GumTree, the state-of-the-art tool to generate fine-grained code differences. In all these cases, the researchers measured the correctness of the output of the review task and the time spent. More specifically, in two of these studies (Huang et al., 2018b,a) the experiment analyzed the degree of understanding the changes.

In contrast with these studies, two evaluations followed a between-subjects design. Khandelwal et al. (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 et al. (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 11, 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) was conducted by Khandelwal et al. (2017), while Runeson and Wohlin (1998) experiment involved the smallest sample, with 8 subjects. Moreover, considering the background of subjects, three of the experiments (Khandelwal et al., 2017; Zhang et al., 2015; Tao and Kim, 2015) were conducted solely with students; the other three experiments (Menarini et al., 2017; Runeson and Wohlin, 1998; Hanam et al., 2019) involved professionals in addition to students. In the experimented conducted by Huang et al. (2018b), the participants are characterized as people engaged in computer-related work with programming experience, but it is not clear whether they are professionals, students, or both.

6.3.2. Findings

Experiments performed in the context of MCR focus on evaluating particular aspects of the proposed approaches. Thus, the key findings target the value promoted by each approach. Two experiments (Zhang et al., 2015; Tao and Kim, 2015) provide 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 review activity. Similarly, three studies (Huang et al., 2018b; Hanam et al., 2019; Huang et al., 2018a) give evidence of a specific positive impact of the proposed approach on the understanding of code changes. In contrast, Khandelwal et al. (2017) found no evidence that gamified tools promote a positive impact on the subjects' interest. Additionally, grounded on observations of the experiment execution, Menarini et al. (2017) indicated that semantically-assisted code review is feasible and effective.

In addition to the main collected data, three experiments also include a follow-up study with the participants to collect their perceptions about the proposed approach. By means of a survey (Khandelwal et al., 2017; Zhang et al., 2015) or an interview (Menarini et al., 2017), the subjective opinion of the participants suggests that the approaches might indeed help the code review process.

6.4. RQ3 - How have MCR approaches been evaluated and what were the reached conclusions?

Most of the approaches proposed to support MCR have been evaluated using offline studies, with the goal of validating the output of the technique or tool, commonly using historical data from code review as input. From 37 EVALUATION papers, 29 (78.38%) present an offline study. In 12 of 29 cases, these validations consider the effectiveness by measuring precision and recall of the output compared to the stored information. In 10 of 29 offline studies, the validations consider the accuracy. The reached conclusions are usually related to a newly-developed approach that outperforms baselines by achieving better effectiveness in an offline evaluation. There are also few reports of EVALUATIONS with user studies (13 of 37 EVALUATIONS), and they are mainly focused on observing the effects and the opinion of the proposed approach.

7. Discussion

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

7.1. Historical developments

Given the primary studies analyzed in this systematic review, we provide a historical perspective of the developments in the field of MCR in Fig. 8. We summarize the total number of papers published per year, as well as papers of each type per year. We can observe an increase in research work on MCR in recent years. This trend indicates the importance and timeliness of the topic. Nevertheless, in the last two years (i.e., 2018 and 2019) we can observe a decrease in the total number of papers. This variation is mainly due to a decreased number of PROPOSALS. Nevertheless, this decline in the number of papers in the last couple of years does not provide enough evidence to claim any medium- and long-term trend.

7.2. Taxonomy of research work on MCR

Our analysis of the primary studies of MCR selected for being investigated in this SLR leads us to identify key topics in this research area. We categorized the topics investigated in FOUNDATIONAL STUDIES, PROPOSALS of approaches to support the practice, and types of EVALUATION studies in the previous sections. We now summarize and propose a taxonomy of aspects of the research on MCR in Fig. 9, which complements the mentioned categories.

In our taxonomy, we first highlight the *goals* of the research work on MCR, which can be related to: (i) a better understanding of one or more code review aspects, e.g. challenges faced by practitioners and factors influencing an outcome; (ii) novel approaches that provide support to practitioners; and (iii) evaluation techniques or tools, which aim to assess the effectiveness of MCR-supporting approaches. Works on MCR target a particular *scope*, which can be the MCR process as a whole or a particular task. Studies or proposals generally rely on collected or existing data, which can be mined from repositories or obtained from human subjects. These data can be from a combination of locations, e.g. OSS or academia. This is captured by the *source of data* facet. Finally, the last facet refers to the *output* associated with the contribution of the work (which can be assessed metrics and/or a particular type of contribution).
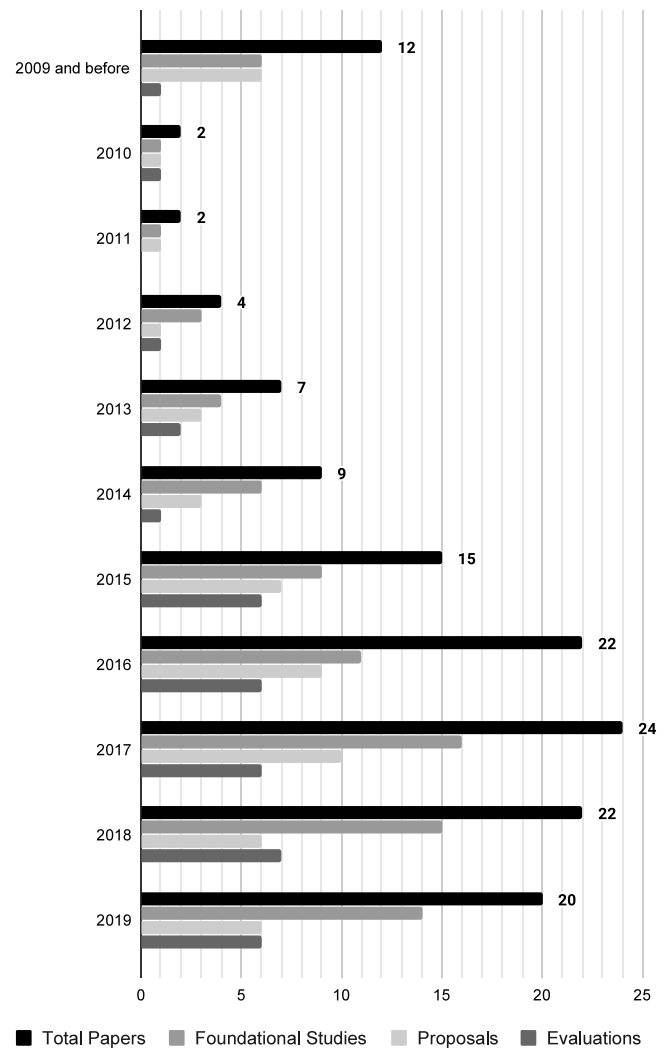

Fig. 8. Distribution of primary studies by type per year of publication.

7.3. Actionable implications and future directions

The findings of our systematic literature review revealed that multiple aspects of MCR have been addressed in the last years. This section discusses the implications of our findings and issues that remain unaddressed.

MCR process improvement. Empirical studies of MCR demonstrate the feasibility of extracting information from the history of review activity stored in tools, such as Gerrit. For practitioners, these findings suggest how to improve internal outcomes, such as that the submission of small code changes increases reviewers' participation and reduces the review duration. Moreover, the findings related to the CodeFlow Analytics from Microsoft (Bird et al., 2015) demonstrate that developers can use review data to improve MCR. Although CodeFlow Analytics is an internal platform of Microsoft, there is an opportunity for researchers to study how developers can use this data to improve the MCR process in their own projects and what outcomes can 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. Similar code improvements in reviewed requests
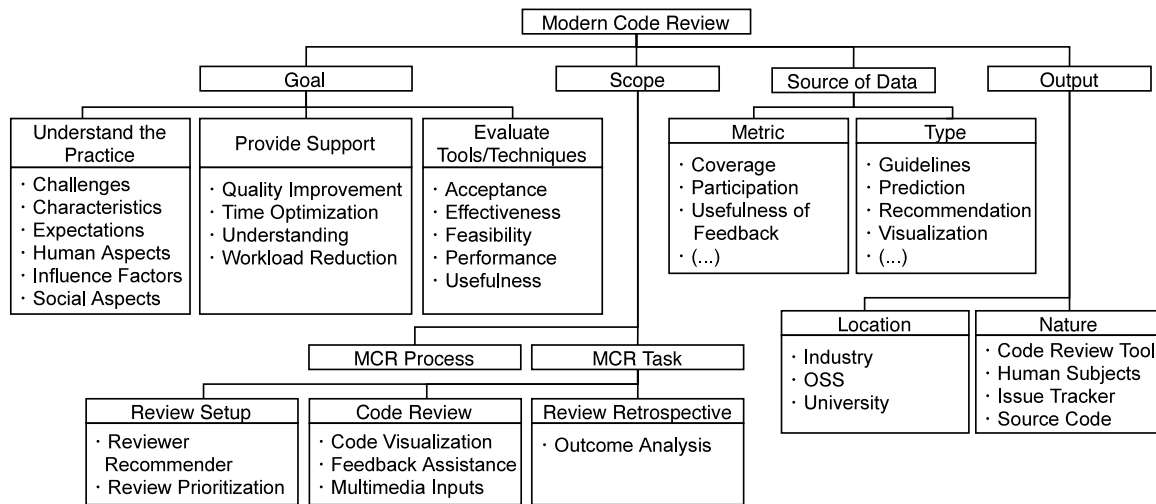
Fig. 9. A taxonomy of MCR research based on primary studies of our SLR.

have been identified (Ueda et al., 2019) and common themes emerged from the manual analyses of review feedback (Bacchelli and Bird, 2013; Spadini et al., 2018). Moreover, there is evidence that authors repeatedly introduce the same types of problems despite the reviewer feedback (Ueda et al., 2018). For practitioners, this implies more attention before submitting a request, disseminating and checking for issues that were earlier addressed, and reducing the review cost. For researchers, these findings suggest that natural language processing (NLP) techniques can be used to extract recurrent bad practices in 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. In our set of 139 primary studies, 19 reported an experiment (13.67%). Including other methodological approaches involving participants, i.e., experiments, interviews, and surveys, 47 of 139 (33.81%) papers present a study with practitioners or students. 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. Recently, Kovalenko et al. (2020) observed this issue and conducted an *in vivo* performance evaluation of a reviewer recommender. As a key finding, the researchers indicate the need for more user-centric approaches to designing and evaluating the recommenders. Therefore, user studies can help researchers to understand how approaches to support MCR are used and perceived and what are the barriers faced by practitioners.

Studies in small and medium-sized companies. From the 86 FOUNDATIONAL STUDIES, 55 studies selected data from 70 different open-source projects. The most frequent projects have been Qt, OpenStack, and Android with 23, 19, and 13 occurrences, respectively. In 17 of 55 studies, there is an overlap in which data from two or three of these projects were used in the same study, e.g., Qt and OpenStack are both data source projects in 13 studies. Considering research in the industry, our systematic review identified 17 papers involving data and participants from companies, a small number compared to studies involving open-source projects. From these studies in the industry, 6 are projects conducted at Microsoft. While the variety of open-source projects might suggest that the existing evidence of MCR spans multiple contexts, we observe as an implication the opportunity of future research to explore code review in other industrial contexts, such as small and medium-sized companies, to understand whether the existing knowledge might be generalized to other scenarios.

7.4. Research limitations

The goal of this study is to identify the state of the art on MCR, providing a structured overview of the research done in this field. We thus performed a *systematic* review in order to minimize the research bias, mitigating the influence of the researchers' expectations during the selection and analysis of a large number of primary studies.

To mitigate the limitations of our SLR in the review planning phase, we selected widely used digital libraries as sources and specified keywords as search string that cover the studied theme, assuming that the selected strategy would retrieve the largest number of relevant studies. Three of our selected databases—ACM Digital Library, IEEE Xplore, SpringerLink—were also used in other secondary studies on MCR (Badampudi et al., 2019; Fronza et al., 2020; Nazir et al., 2020). While these other studies also searched the Scopus database, we selected ScienceDirect as a source. However, both Scopus and ScienceDirect are provided by Elsevier.

Although this search is large, it is not complete and, therefore, our review does not cover all existing work on MCR. An example of a study that was not retrieved in our search is that of Rigby et al. (2008) because it uses the term peer review, which is not covered by our search string. In this case, however, the findings of this study were used for comparison purposes by Rigby and Bird (2013), which is a primary study in our review. As discussed when we introduced our search string, some authors used *peer review* to refer to the MCR practice, but this term is also used in other contexts, such as peer review of scientific papers. Our search identified papers that used the term *peer code review*, but not solely *peer review*. The advantage of systematic reviews is that they can be further extended in future reviews.

Moreover, to identify further studies, a snowballing approach could have been used. This approach has not been followed in the

present work because a preliminary analysis of the obtained results indicated that the most relevant studies had been retrieved by searching our selected digital libraries. Nevertheless, we estimated whether and how the lack of snowballing could have influenced our review. We randomly selected 20 papers from the set of primary studies, creating our referencing list. We then conducted a backward snowballing (Wohlin, 2014) using this reference list to identify additional primary studies. We followed the same selection process of our SLR in this backward snowballing, using the same selection criteria. We checked 619 new results from our reference list and identified 8 papers that should be included in our review but were not identified by our procedure. From these eight papers, we checked 240 other results, including 2 papers. Then, we checked 26 results from those 2 papers, but we did not find more entries. Our procedure to estimate whether and how the lack of snowballing could have influenced our review identified 10 papers that were not identified by our search. The papers identified in this procedure used specific terms to refer to MCR practice, such as "PR review" (Ying et al., 2016) and "patch review" (Nurolahzade et al., 2009). Our estimation illustrates the amount of work required to perform backward snowballing. As our focus was conducting a systematic literature review, not a systematic mapping study, we suggest snowballing as future work, in which both backward and forward snowballing can be conducted as a complement to our review. In addition, our SLR in the present form already includes several papers. Including the snowballing approach could lead to a lengthy paper, which might be difficult to read.

The selection and classification of primary studies might also be considered a threat to validity. To avoid bias, we followed systematic procedures for the search, selection, data extraction, information labeling, and data analysis. Moreover, these tasks were conducted by the first author of this paper, and then the outcomes were reviewed and discussed with the second author. A single researcher analyzing the primary studies in an SLR is a practice observed in other studies, e.g. in the work of Paiva et al. (2021). However, these are subjective tasks, and other studies can result in different categorizations and summarizations of the research on MCR.

8. Conclusion

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 paper the results of a systematic literature review on MCR, which includes 139 primary studies.

We identified three main categories of studies, namely FOUNDATIONAL STUDIES, PROPOSALS of novel approaches to support the practice, and EVALUATIONS of proposed approaches. Each paper category was systematically analyzed observing aspects relevant for each type of study. 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 PROPOSALS of novel approaches to support MCR, the most common are those to help code checking and to recommend reviewers. EVALUATIONS of MCR-supporting approaches have been done mostly offline and few studies involving human subjects have been conducted.

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. 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. 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.

CRediT authorship contribution statement

Nicole Davila: Conceptualization, Methodology, Data curation, Investigation, Validation, Writing - review & editing. **Ingrid Nunes:** Conceptualization, Methodology, Supervision, Validation, Writing - review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

References

Ahmed, T., Bosu, A., Iqbal, A., Rahimi, S., 2017. Senticr: A customized sentiment analysis tool for code review interactions. In: Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering. In: ASE 2017, IEEE Press, pp. 106–111.

Alami, A., Cohn, M.L., Wasowski, A., 2019. Why does code review work for open source software communities? In: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). pp. 1073–1083.

Albayrak, O., Davenport, D., 2010. Impact of maintainability defects on code inspections. In: Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement. In: ESEM '10, ACM, New York, NY, USA, pp. 50:1–50:4. http://dx.doi.org/10.1145/1852786.1852850.

Aman, H., 2013. 0-1 programming model-based method for planning code review using bug fix history. In: 2013 20th Asia-Pacific Software Engineering Conference (APSEC). Vol. 2, pp. 37–42.

An, L., Khomh, F., Mcintosh, S., Castelluccio, M., 2018. Why did this reviewed code crash? An empirical study of mozilla firefox. In: 2018 25th Asia-Pacific Software Engineering Conference (APSEC). IEEE, pp. 396–405.

Armstrong, F., Khomh, F., Adams, B., 2017. Broadcast vs. Unicast review technology: Does it matter? In: 2017 IEEE International Conference on Software Testing, Verification and Validation (ICST). IEEE, pp. 219–229. http://dx.doi.org/10.1109/ICST.2017.27.

Asthana, S., Kumar, R., Bhagwan, R., Bird, C., Bansal, C., Maddila, C., Mehta, S., Ashok, B., 2019. Whodo: Automating reviewer suggestions at scale. In: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. In: ESEC/FSE 2019, Association for Computing Machinery, New York, NY, USA, pp. 937–945. http://dx.doi.org/10.1145/3338906.3340449.

Asundi, J., Jayant, R., 2007. Patch review processes in open source software development communities: A comparative case study. In: Proceedings of the 40th Annual Hawaii International Conference on System Sciences. In: HICSS'07, IEEE Computer Society, USA, p. 166c. http://dx.doi.org/10.1109/HICSS.2007.426.

Bacchelli, A., Bird, C., 2013. Expectations, outcomes, and challenges of modern code review. In: Proceedings of the 2013 International Conference on Software Engineering. In: ICSE'13, IEEE Press, pp. 712–721.

Badampudi, D., Britto, R., Unterkalmsteiner, M., 2019. Modern code reviews - preliminary results of a systematic mapping study. In: Proceedings of the Evaluation and Assessment on Software Engineering. In: EASE'19, ACM, New York, NY, USA, pp. 340–345. http://dx.doi.org/10.1145/3319008.3319354.

Balachandran, V., 2013. 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. In: ICSE '13, IEEE Press, pp. 931–940.

Barnett, M., Bird, C., Brunet, J.a., Lahiri, S.K., 2015. Helping developers help themselves: Automatic decomposition of code review changesets. In: Proceedings of the 37th International Conference on Software Engineering. Vol. 1. In: ICSE '15, IEEE Press, pp. 134–144.

Baum, T., Kortum, F., Schneider, K., Brack, A., Schauder, J., 2016a. Comparing pre commit reviews and post commit reviews using process simulation. In: Proceedings of the International Conference on Software and Systems Process. In: ICSSP '16, ACM, New York, NY, USA, pp. 26–35. http://dx.doi.org/10.1145/2904354.2904362.

Baum, T., Leßmann, H., Schneider, K., 2017a. The choice of code review process: A survey on the state of the practice. In: Felderer, M., Méndez Fernández, D., Turhan, B., Kalinowski, M., Sarro, F., Winkler, D. (Eds.), Product-Focused Software Process Improvement. Springer International Publishing, Cham, pp. 111–127.

Baum, T., Liskin, O., Niklas, K., Schneider, K., 2016b. A faceted classification scheme for change-based industrial code review processes. In: 2016 IEEE International Conference on Software Quality, Reliability and Security (QRS). pp. 74–85. http://dx.doi.org/10.1109/QRS.2016.19.

Baum, T., Liskin, O., Niklas, K., Schneider, K., 2016c. Factors influencing code review processes in industry. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. In: FSE 2016, ACM, New York, NY, USA, pp. 85–96. http://dx.doi.org/10.1145/2950290.2950323.

Baum, T., Schneider, K., 2016. On the need for a new generation of code review tools. In: Abrahamsson, P., Jedlitschka, A., Nguyen Duc, A., Felderer, M., Amasaki, S., Mikkonen, T. (Eds.), Product-Focused Software Process Improvement. Springer International Publishing, Cham, pp. 301–308.

Baum, T., Schneider, K., Bacchelli, A., 2017b. On the optimal order of reading source code changes for review. In: 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE, Shanghai, China, pp. 329–340. http://dx.doi.org/10.1109/ICSME.2017.28.

Baum, T., Schneider, K., Bacchelli, A., 2019. Associating working memory capacity and code change ordering with code review performance. Empir. Softw. Eng. 24 (4), 1762–1798.

Bavota, G., Russo, B., 2015. 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). IEEE, pp. 81–90. http://dx.doi.org/10.1109/ICSM.2015.7332454.

Baysal, O., Kononenko, O., Holmes, R., Godfrey, M.W., 2012. The secret life of patches: A firefox case study. In: 2012 19th Working Conference on Reverse Engineering. IEEE Computer Society, pp. 447–455. http://dx.doi.org/10.1109/WCRE.2012.54.

Baysal, O., Kononenko, O., Holmes, R., Godfrey, M.W., 2016. Investigating technical and non-technical factors influencing modern code review. Empir. Softw. Eng. 21 (3), 932–959.

Begel, A., Vrzakova, H., 2018. Eye movements in code review. In: Proceedings of the Workshop on Eye Movements in Programming. In: EMIP '18, ACM, New York, NY, USA, pp. 5:1–5:5. http://dx.doi.org/10.1145/3216723.3216727.

Beller, M., Bacchelli, A., Zaidman, A., Juergens, E., 2014. Modern code reviews in open-source projects: Which problems do they fix? In: Proceedings of the 11th Working Conference on Mining Software Repositories. In: MSR 2014, ACM, New York, NY, USA, pp. 202–211. http://dx.doi.org/10.1145/2597073.2597082.

Bernhart, M., Grechenig, T., 2013. On the understanding of programs with continuous code reviews. In: 2013 21st International Conference on Program Comprehension (ICPC). pp. 192–198. http://dx.doi.org/10.1109/ICPC.2013.6613847.

Bird, C., Carnahan, T., Greiler, M., 2015. Lessons learned from building and deploying a code review analytics platform. In: 2015 IEEE/ACM 12th Working Conference on Mining Software Repositories. In: MSR '15, IEEE Press, pp. 191–201. http://dx.doi.org/10.1109/MSR.2015.25.

Bosu, A., Carver, J.C., 2012. Peer code review in open source communitiesusing reviewboard. In: Proceedings of the ACM 4th Annual Workshop on Evaluation and Usability of Programming Languages and Tools. In: PLATEAU '12, ACM, New York, NY, USA, pp. 17–24. http://dx.doi.org/10.1145/2414721.2414726.

Bosu, A., Carver, J.C., 2013. Impact of peer code review on peer impression formation: A survey. In: 2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement. IEEE, pp. 133–142. http://dx.doi.org/10.1109/ESEM.2013.23.

Bosu, A., Carver, J.C., 2014. 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. In: ESEM '14, ACM, New York, NY, USA, pp. 33:1–33:10. http://dx.doi.org/10.1145/2652524.2652544.

Bosu, A., Carver, J.C., Bird, C., Orbeck, J., Chockley, C., 2017. Process aspects and social dynamics of contemporary code review: Insights from open source development and industrial practice at microsoft. IEEE Trans. Softw. Eng. 43 (1), 56–75. http://dx.doi.org/10.1109/TSE.2016.2576451.

Bosu, A., Carver, J.C., Hafiz, M., Hilley, P., Janni, D., 2014. Identifying the characteristics of vulnerable code changes: An empirical study. In: Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. In: FSE 2014, ACM, New York, NY, USA, pp. 257–268. http://dx.doi.org/10.1145/2635868.2635880.

Bosu, A., Greiler, M., Bird, C., 2015. Characteristics of useful code reviews: An empirical study at microsoft. In: Proceedings of the 12th Working Conference on Mining Software Repositories. In: MSR '15, IEEE Press, pp. 146–156, http://dl.acm.org/citation.cfm?id=2820518.2820538.

Burhandenny, A.E., Aman, H., Kawahara, M., 2016. Examination of coding violations focusing on their change patterns over releases. In: 2016 23rd Asia-Pacific Software Engineering Conference (APSEC). IEEE, Hamilton, New Zealand, pp. 121–128. http://dx.doi.org/10.1109/APSEC.2016.027.

Chandrika, K.R., Amudha, J., Sudarsan, S.D., 2017. Recognizing eye tracking traits for source code review. In: 2017 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA). IEEE, pp. 1–8. http://dx.doi.org/10.1109/ETFA.2017.8247637.

Coelho, F., Massoni, T., Alves, E.L., 2019. Refactoring-aware code review: a systematic mapping study. In: 2019 IEEE/ACM 3rd International Workshop on Refactoring (IWoR). IEEE, pp. 63–66.

Cohen, J., 2010. Modern code review. In: Making Software: What Really Works, and Why We Believe It. O'Reilly Media, Inc., pp. 329–336.

di Biase, M., Bruntink, M., Bacchelli, A., 2016. A security perspective on code review: The case of chromium. In: 2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM). IEEE, pp. 21–30. http://dx.doi.org/10.1109/SCAM.2016.30.

Doğan, E., Tüzün, E., Tecimer, K.A., Güvenir, H.A., 2019. Investigating the validity of ground truth in code reviewer recommendation studies. In: 2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM). IEEE, pp. 1–6.

Duley, A., Spandikow, C., Kim, M., 2010. A program differencing algorithm for verilog HDL. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering. In: ASE '10, ACM, New York, NY, USA, pp. 477–486. http://dx.doi.org/10.1145/1858996.1859093.

Dunsmore, A., Roper, M., Wood, M., 2000. Object-oriented inspection in the face of delocalisation. In: Proceedings of the 22Nd International Conference on Software Engineering. In: ICSE '00, ACM, pp. 467–476. http://dx.doi.org/10.1145/337180.337343.

Duraes, J., Madeira, H., Castelhano, J., Duarte, C., Branco, M.C., 2016. WAP: Understanding the brain at software debugging. In: 2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE). IEEE, pp. 87–92. http://dx.doi.org/10.1109/ISSRE.2016.53.

Ebert, F., Castor, F., Novielli, N., Serebrenik, A., 2017. Confusion detection in code reviews. In: 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE, pp. 549–553. http://dx.doi.org/10.1109/ICSME.2017.40.

Ebert, F., Castor, F., Novielli, N., Serebrenik, A., 2018. Communicative intention in code review questions. In: 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE, pp. 519–523.

Ebert, F., Castor, F., Novielli, N., Serebrenik, A., Confusion in code reviews: reasons, impacts, and coping strategies. In: 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER). 2019, pp. 49–60.

Efstathiou, V., Spinellis, D., 2018. Code review comments: Language matters. In: Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results. In: ICSE-NIER '18, ACM, New York, NY, USA, pp. 69–72. http://dx.doi.org/10.1145/3183399.3183411.

El Asri, I., Kerzazi, N., Uddin, G., Khomh, F., Idrissi, M.J., 2019. An empirical study of sentiments in code reviews. Inf. Softw. Technol. 114, 37–54.

Fagan, M., 1976. Design and code inspections to reduce errors in program development. IBM Syst. J. 15 (3), 182–211.

Fan, Y., Xia, X., Lo, D., Li, S., 2018. Early prediction of merged code changes to prioritize reviewing tasks. Empir. Softw. Eng. 23 (6), 3346–3393.

Fatima, N., Nazir, S., Chuprat, S., 2019a. Knowledge sharing, a key sustainable practice is on risk: An insight from modern code review. In: 2019 IEEE 6th International Conference on Engineering Technologies and Applied Sciences (ICETAS). IEEE, pp. 1–6.

Fatima, N., Nazir, S., Chuprat, S., 2019b. Understanding the impact of feedback on knowledge sharing in modern code review. In: 2019 IEEE 6th International Conference on Engineering Technologies and Applied Sciences (ICETAS). IEEE, pp. 1–5.

Fejzer, M., Przymus, P., Stencel, K., 2018. Profile based recommendation of code reviewers. J. Intell. Inf. Syst. 50 (3), 597–619.

Floyd, B., Santander, T., Weimer, W., 2017. 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. In: ICSE '17, IEEE Press, pp. 175–186. http://dx.doi.org/10.1109/ICSE.2017.24.

Fronza, I., Hellas, A., Ihantola, P., Mikkonen, T., 2020. Code reviews, software inspections, and code walkthroughs: Systematic mapping study of research topics. In: Winkler, D., Biffl, S., Mendez, D., Bergsmann, J. (Eds.), Software Quality: Quality Intelligence in Software and Systems Engineering. Springer International Publishing, Cham, pp. 121–133.

Ge, X., Sarkar, S., Witschey, J., Murphy-Hill, E., 2017. Refactoring-aware code review. In: 2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC). IEEE, pp. 71–79. http://dx.doi.org/10.1109/VLHCC.2017.8103453.

German, D.M., Robles, G., Poo-Caamaño, G., Yang, X., Iida, H., Inoue, K., 2018. "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. In: ICSE '18, ACM, New York, NY, USA, pp. 523–534. http://dx.doi.org/10.1145/3180155.3180217.

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

Guo, B., Kwon, Y.-W., Song, M., 2019. Decomposing composite changes for code review and regression test selection in evolving software. J. Comput. Sci. Tech. 34 (2), 416–436.

Hanam, Q., Mesbah, A., Holmes, R., Aiding code change understanding with semantic change impact analysis. In: 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME). 2019, pp. 202–212.

Hannebauer, C., Patalas, M., Stünkel, S., Gruhn, V., 2016. Automatically recommending code reviewers based on their expertise: An empirical comparison. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering. In: ASE 2016, ACM, New York, NY, USA, pp. 99–110. http://dx.doi.org/10.1145/2970276.2970306.

Hao, Y., Li, G., Mou, L., Zhang, L., Jin, Z., 2013. MCT: A tool for commenting programs by multimedia comments. In: Proceedings of the 2013 International Conference on Software Engineering. In: ICSE '13, IEEE Press, pp. 1339–1342, http://dl.acm.org/citation.cfm?id=2486788.2487000.

Harel, A., Kantorowitz, E., 2005. 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). IEEE Computer Society, pp. 151–160. http://dx.doi.org/10.1109/SWSTE.2005.1.

Hirao, T., Ihara, A., Matsumoto, K.-i., 2015. 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. In: CASCON '15, IBM Corp., Riverton, NJ, USA, pp. 248–251, http://dl.acm.org/citation.cfm?id=2886444.2886485.

Hirao, T., Ihara, A., Ueda, Y., Phannachitta, P., Matsumoto, K.-i., 2016. The impact of a low level of agreement among reviewers in a code review process. In: Crowston, K., Hammouda, I., Lundell, B., Robles, G., Gamalielsson, J., Lindman, J. (Eds.), Open Source Systems: Integrating Communities. Springer International Publishing, Cham, pp. 97–110.

Hirao, T., McIntosh, S., Ihara, A., Matsumoto, K., 2019. The review linkage graph for code review analytics: A recovery approach and empirical study. In: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. In: ESEC/FSE 2019, Association for Computing Machinery, New York, NY, USA, pp. 578–589. http://dx.doi.org/10.1145/3338906.3338949.

Huang, K., Chen, B., Peng, X., Zhou, D., Wang, Y., Liu, Y., Zhao, W., 2018a. Cldiff: Generating concise linked code differences. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. In: ASE 2018, Association for Computing Machinery, New York, NY, USA, pp. 679–690. http://dx.doi.org/10.1145/3238147.3238219.

Huang, Y., Jia, N., Chen, X., Hong, K., Zheng, Z., 2018b. Salient-class location: Help developers understand code change in code review. In: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. In: ESEC/FSE 2018, Association for Computing Machinery, New York, NY, USA, pp. 770–774. http://dx.doi.org/10.1145/3236024.3264841.

Hundhausen, C.D., Agarwal, P., Trevisan, M., 2011. Online vs. Face-to-face pedagogical code reviews: An empirical comparison. In: Proceedings of the 42Nd ACM Technical Symposium on Computer Science Education. In: SIGCSE '11, ACM, New York, NY, USA, pp. 117–122. http://dx.doi.org/10.1145/1953163.1953201.

Izquierdo-Cortazar, D., Sekitoleko, N., Gonzalez-Barahona, J.M., Kurth, L., 2017. Using metrics to track code review performance. In: Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering. In: EASE'17, ACM, New York, NY, USA, pp. 214–223. http://dx.doi.org/10.1145/3084226.3084247.

Jiang, J., He, J.-H., Chen, X.-Y., 2015. Coredevrec: Automatic core member recommendation for contribution evaluation. J. Comput. Sci. Tech. 30 (5), 998–1016.

Jiang, J., Lo, D., Zheng, J., Xia, X., Yang, Y., Zhang, L., 2019a. Who should make decision on this pull request? Analyzing time-decaying relationships and file similarities for integrator prediction. J. Syst. Softw. 154, 196–210.

Jiang, J., Mohamed, A., Zhang, L., 2019b. What are the characteristics of reopened pull requests? a case study on open source projects in github. IEEE Access 7, 102751–102761.

Jiang, J., Yang, Y., He, J., Blanc, X., Zhang, L., 2017. Who should comment on this pull request? Analyzing attributes for more accurate commenter recommendation in pull-based development. Inf. Softw. Technol. 84, 48–62.

Kagdi, H., Hammad, M., Maletic, J.I., 2008. Who can help me with this source code change? In: 2008 IEEE International Conference on Software Maintenance. pp. 157–166. http://dx.doi.org/10.1109/ICSM.2008.4658064.

Kalyan, A., Chiam, M., Sun, J., Manoharan, S., 2016. A collaborative code review platform for github. In: 2016 21st International Conference on Engineering of Complex Computer Systems (ICECCS). IEEE, pp. 191–196. http://dx.doi.org/10.1109/ICECCS.2016.032.

Kerzazi, N., El Asri, I., 2016. Who can help to review this piece of code? In: Afsarmanesh, H., Camarinha-Matos, L.M., Lucas Soares, A. (Eds.), Collaboration in a Hyperconnected World. Springer International Publishing, Cham, pp. 289–301.

Khandelwal, S., Sripada, S.K., Reddy, Y.R., 2017. Impact of gamification on code review process: An experimental study. In: Proceedings of the 10th Innovations in Software Engineering Conference. In: ISEC '17, ACM, New York, NY, USA, pp. 122–126. http://dx.doi.org/10.1145/3021460.3021474.

Kitagawa, N., Hata, H., Ihara, A., Kogiso, K., Matsumoto, K., 2016. 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. In: CHASE '16, ACM, New York, NY, USA, pp. 64–67. http://dx.doi.org/10.1145/2897586.2897605.

Kitchenham, B., Charters, S., 2007. Guidelines for performing systematic literature reviews in software engineering. Technical Report EBSE-2007-01, MKeele University.

Kononenko, O., Baysal, O., Godfrey, M.W., 2016. Code review quality: How developers see it. In: Proceedings of the 38th International Conference on Software Engineering. In: ICSE '16, ACM, New York, NY, USA, pp. 1028–1038. http://dx.doi.org/10.1145/2884781.2884840.

Kononenko, O., Baysal, O., Guerrouj, L., Cao, Y., Godfrey, M.W., 2015. Investigating code review quality: Do people and participation matter? In: 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE Computer Society, pp. 111–120. http://dx.doi.org/10.1109/ICSM.2015.7332457.

Kovalenko, V., Bacchelli, A., 2018. Code review for newcomers: Is it different? In: Proceedings of the 11th International Workshop on Cooperative and Human Aspects of Software Engineering. In: CHASE '18, ACM, New York, NY, USA, pp. 29–32. http://dx.doi.org/10.1145/3195836.3195842.

Kovalenko, V., Tintarev, N., Pasynkov, E., Bird, C., Bacchelli, A., 2020. Does reviewer recommendation help developers? IEEE Trans. Softw. Eng. 46 (7), 710–731.

Lanubile, F., Mallardo, T., 2002. Tool support for distributed inspection. In: Proceedings of the 26th International Computer Software and Applications Conference on Prolonging Software Life: Development and Redevelopment. In: COMPSAC '02, IEEE Computer Society, pp. 1071–1076, http://dl.acm.org/citation.cfm?id=645984.675578.

Lee, A., Carver, J.C., 2017. 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). IEEE Press, pp. 1–10. http://dx.doi.org/10.1109/ESEM.2017.7.

Li, Z.-X., Yu, Y., Yin, G., Wang, T., Wang, H.-M., 2017. What are they talking about? Analyzing code reviews in pull-based development model. J. Comput. Sci. Tech. 32 (6), 1060–1075.

Liang, J., Mizuno, O., 2011. 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. IEEE, pp. 126–132. http://dx.doi.org/10.1109/IWSM-MENSURA.2011.33.

Liao, Z., Wu, Z., Wu, J., Zhang, Y., Liu, J., Long, J., 2019. TIRR: A code reviewer recommendation algorithm with topic model and reviewer influence. In: 2019 IEEE Global Communications Conference (GLOBECOM). pp. 1–6. http://dx.doi.org/10.1109/GLOBECOM38437.2019.9014249.

Luna Freire, V.d.C., Brunet, J., de Figueiredo, J.C.A., 2018. Automatic decomposition of java open source pull requests: A replication study. In: Tjoa, A.M., Bellatreche, L., Biffl, S., van Leeuwen, J., Wiedermann, J. (Eds.), SOFSEM 2018: Theory and Practice of Computer Science. Springer International Publishing, Cham, pp. 255–268.

MacLeod, L., Greiler, M., Storey, M.-A., Bird, C., Czerwonka, J., 2018. Code reviewing in the trenches: Challenges and best practices. IEEE Softw. 35 (4), 34–42. http://dx.doi.org/10.1109/MS.2017.265100500.

McIntosh, S., Kamei, Y., Adams, B., Hassan, A.E., 2014. 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. In: MSR 2014, ACM, New York, NY, USA, pp. 192–201. http://dx.doi.org/10.1145/2597073.2597076.

McIntosh, S., Kamei, Y., Adams, B., Hassan, A.E., 2016. An empirical study of the impact of modern code review practices on software quality. Empir. Softw. Eng. 21 (5), 2146–2189.

Menarini, M., Yan, Y., Griswold, W.G., 2017. Semantics-assisted code review: An efficient tool chain and a user study. In: 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE Press, pp. 554–565. http://dx.doi.org/10.1109/ASE.2017.8115666.

Meneely, A., Tejeda, A.C.R., Spates, B., Trudeau, S., Neuberger, D., Whitlock, K., Ketant, C., Davis, K., 2014. An empirical investigation of socio-technical code review metrics and security vulnerabilities. In: Proceedings of the 6th International Workshop on Social Software Engineering. In: SSE 2014, ACM, New York, NY, USA, pp. 37–44. http://dx.doi.org/10.1145/2661685.2661687, http://doi.acm.org/10.1145/2661685.2661687.

Mishra, R., Sureka, A., 2014. Mining peer code review system for computing effort and contribution metrics for patch reviewers. In: 2014 IEEE 4th Workshop on Mining Unstructured Data. IEEE Computer Society, pp. 11–15. http://dx.doi.org/10.1109/MUD.2014.11.

Mizuno, O., Liang, J., 2015. Does a code review tool evolve as the developer intended? In: Lee, R. (Ed.), Software Engineering Research, Management and Applications. Springer International Publishing, Cham, pp. 59–74. http://dx.doi.org/10.1007/978-3-319-11265-7_5.

Morales, R., McIntosh, S., Khomh, F., 2015. 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). IEEE Press, pp. 171–180. http://dx.doi.org/10.1109/SANER.2015.7081827.

Müller, M.M., 2005. Two controlled experiments concerning the comparison of pair programming to peer review. J. Syst. Softw. 78 (2), 166–179.

Müller, S., Würsch, M., Fritz, T., Gall, H.C., 2012. 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). IEEE Press, pp. 93–99. http://dx.doi.org/10.1109/CHASE.2012.6223031.

Murakami, Y., Tsunoda, M., Uwano, H., 2017. WAP: Does reviewer age affect code review performance? In: 2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE). IEEE, pp. 164–169. http://dx.doi.org/10.1109/ISSRE.2017.37.

Nagoya, F., Liu, S., Chen, Y., 2005. A tool and case study for specification-based program review. In: 29th Annual International Computer Software and Applications Conference (COMPSAC'05). Vol. 1. IEEE Press, pp. 375–380. http://dx.doi.org/10.1109/COMPSAC.2005.36.

Nazir, S., Fatima, N., Chuprat, S., 2019a. Does project associated situational factors have impact on sustainability of modern code review workforce? In: 2019 IEEE 6th International Conference on Engineering Technologies and Applied Sciences (ICETAS). IEEE, pp. 1–5.

Nazir, S., Fatima, N., Chuprat, S., 2019b. Situational factors affecting software engineers sustainability: A vision of modern code review. In: 2019 IEEE 6th International Conference on Engineering Technologies and Applied Sciences (ICETAS). IEEE, pp. 1–6.

Nazir, S., Fatima, N., Chuprat, S., 2020. Modern code review benefits-primary findings of a systematic literature review. In: Proceedings of the 3rd International Conference on Software Engineering and Information Management. In: ICSIM '20, Association for Computing Machinery, New York, NY, USA, pp. 210–215. http://dx.doi.org/10.1145/3378936.3378954.

Norikane, T., Ihara, A., Matsumoto, K., 2018. Do review feedbacks influence to a contributor's time spent on oss projects? In: 2018 IEEE International Conference on Big Data, Cloud Computing, Data Science & Engineering (BCD). IEEE, pp. 109–113.

Nurolahzade, M., Nasehi, S.M., Khandkar, S.H., Rawal, S., 2009. The role of patch review in software evolution: An analysis of the mozilla firefox. In: Proceedings of the Joint International and Annual ERCIM Workshops on Principles of Software Evolution (IWPSE) and Software Evolution (Evol) Workshops. In: IWPSE-Evol '09, Association for Computing Machinery, New York, NY, USA, pp. 9–18. http://dx.doi.org/10.1145/1595808.1595813.

Ouni, A., Kula, R.G., Inoue, K., 2016. Search-based peer reviewers recommendation in modern code review. In: 2016 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE, pp. 367–377. http://dx.doi.org/10.1109/ICSME.2016.65.

Paiva, D.M.B., Freire, A.P., de Mattos Fortes, R.P., 2021. Accessibility and software engineering processes: A systematic literature review. J. Syst. Softw. 171, 110819.

Paixao, M., Krinke, J., Han, D., Ragkhitwetsagul, C., Harman, M., 2019. The impact of code review on architectural changes. IEEE Trans. Softw. Eng..

Paixao, M., Maia, P.H., 2019. Rebasing in code review considered harmful: A large-scale empirical investigation. In: 2019 19th International Working Conference on Source Code Analysis and Manipulation (SCAM). IEEE, pp. 45–55.

Pangsakulyanont, T., Thongtanunam, P., Port, D., Iida, H., 2014. Assessing MCR discussion usefulness using semantic similarity. In: 2014 6th International Workshop on Empirical Software Engineering in Practice. pp. 49–54. http://dx.doi.org/10.1109/IWESEP.2014.11.

Panichella, S., Arnaoudova, V., Penta, M.D., Antoniol, G., 2015. Would static analysis tools help developers with code reviews? In: 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER). IEEE, pp. 161–170. http://dx.doi.org/10.1109/SANER.2015.7081826.

Pascarella, L., Spadini, D., Palomba, F., Bruntink, M., Bacchelli, A., 2018. Information needs in contemporary code review. Proc. ACM Hum.-Comput. Interact. 2 (CSCW).

Paul, R., Bosu, A., Sultana, K.Z., 2019. Expressions of sentiments during code reviews: Male vs. female. In: 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE, pp. 26–37.

Peng, Z., Yoo, J., Xia, M., Kim, S., Ma, X., 2018. Exploring how software developers work with mention bot in github. In: Proceedings of the Sixth International Symposium of Chinese CHI. In: ChineseCHI '18, ACM, New York, NY, USA, pp. 152–155. http://dx.doi.org/10.1145/3202667.3202694.

Perry, D.E., Porter, A., Wade, M.W., Votta, L.G., Perpich, J., 2002. Reducing inspection interval in large-scale software development. IEEE Trans. Softw. Eng. 28 (7), 695–705.

Petersen, A., Zingaro, D., 2018. Code reviews in large, first-year courses. In: Proceedings of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education. In: ITiCSE 2018, ACM, New York, NY, USA, pp. 354–355. http://dx.doi.org/10.1145/3197091.3205832.

Priest, R., Plimmer, B., 2006. 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. In: CHINZ '06, ACM, New York, NY, USA, pp. 53–60. http://dx.doi.org/10.1145/1152760.1152767.

Rahman, M.M., Roy, C.K., 2017. Impact of continuous integration on code reviews. In: Proceedings of the 14th International Conference on Mining Software Repositories. In: MSR '17, IEEE Press, pp. 499–502. http://dx.doi.org/10.1109/MSR.2017.39.

Rahman, M.M., Roy, C.K., Collins, J.A., 2016. 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). IEEE Press, pp. 222–231.

Rahman, M.M., Roy, C.K., Kula, R.G., 2017. Predicting usefulness of code review comments using textual features and developer experience. In: 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR). IEEE Press, pp. 215–226. http://dx.doi.org/10.1109/MSR.2017.17.

Ram, A., Sawant, A.A., Castelluccio, M., Bacchelli, A., 2018. What makes a code change easier to review: An empirical investigation on code change reviewability. In: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. In: ESEC/FSE 2018, Association for Computing Machinery, New York, NY, USA, pp. 201–212. http://dx.doi.org/10.1145/3236024.3236080.

Rigby, P., Bird, C., 2013. Convergent contemporary software peer review practices. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering. In: ESEC/FSE 2013, ACM, New York, NY, USA, pp. 202–212. http://dx.doi.org/10.1145/2491411.2491444.

Rigby, P., Cleary, B., Painchaud, F., Storey, M.-A., German, D., 2012. Contemporary peer review in action: Lessons from open source development. IEEE Softw. 29 (6), 56–61. http://dx.doi.org/10.1109/MS.2012.24.

Rigby, P.C., German, D.M., Storey, M.-A., 2008. Open source software peer review practices: A case study of the apache server. In: Proceedings of the 30th International Conference on Software Engineering. In: ICSE '08, Association for Computing Machinery, New York, NY, USA, pp. 541–550. http://dx.doi.org/10.1145/1368088.1368162.

Ruangwan, S., Thongtanunam, P., Ihara, A., Matsumoto, K., 2019. The impact of human factors on the participation decision of reviewers in modern code review. Empir. Softw. Eng. 24 (2), 973–1016.

Runeson, P., Andrews, A., 2003. 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. IEEE Computer Society, pp. 3–13. http://dx.doi.org/10.1109/ISSRE.2003.1251026.

Runeson, P., Wohlin, C., 1998. An experimental evaluation of an experience-based capture-recapture method in software code inspections. Empir. Softw. Eng. 3 (4), 381–406.

Sadowski, C., Söderberg, E., Church, L., Sipko, M., Bacchelli, A., 2018. Modern code review: A case study at google. In: Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice. In: ICSE-SEIP '18, ACM, New York, NY, USA, pp. 181–190. http://dx.doi.org/10.1145/3183519.3183525.

Santos, E.W.d., Nunes, I., 2018. Investigating the effectiveness of peer code review in distributed software development based on objective and subjective data. J. Softw. Eng. Res. Dev. 6 (1), 14.

Shimagaki, J., Kamei, Y., McIntosh, S., Hassan, A.E., Ubayashi, N., 2016. 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. In: ICSE '16, ACM, New York, NY, USA, pp. 212–221. http://dx.doi.org/10.1145/2889160.2889243.

Singh, D., Sekar, V.R., Stolee, K.T., Johnson, B., 2017. Evaluating how static analysis tools can reduce code review effort. In: 2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC). IEEE, pp. 101–105. http://dx.doi.org/10.1109/VLHCC.2017.8103456.

Soltanifar, B., Erdem, A., Bener, A., 2016. Predicting defectiveness of software patches. In: Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement. In: ESEM '16, ACM, New York, NY, USA, pp. 22:1–22:10. http://dx.doi.org/10.1145/2961111.2962601.

Spadini, D., Aniche, M., Storey, M.-A., Bruntink, M., Bacchelli, A., 2018. When testing meets code review: Why and how developers review tests. In: Proceedings of the 40th International Conference on Software Engineering. In: ICSE '18, ACM, New York, NY, USA, pp. 677–687. http://dx.doi.org/10.1145/3180155.3180192.

Spadini, D., Palomba, F., Baum, T., Hanenberg, S., Bruntink, M., Bacchelli, A., 2019. Test-driven code review: an empirical study. In: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). pp. 1061–1072.

Sripada, S.K., Reddy, Y.R., Khandelwal, S., 2016. Architecting an extensible framework for gamifying software engineering concepts. In: Proceedings of the 9th India Software Engineering Conference. In: ISEC '16, ACM, New York, NY, USA, pp. 119–130. http://dx.doi.org/10.1145/2856636.2856649.

Sutherland, A., Venolia, G., 2009. Can peer code reviews be exploited for later information needs? In: 2009 31st International Conference on Software Engineering - Companion Volume. IEEE, pp. 259–262. http://dx.doi.org/10.1109/ICSE-COMPANION.2009.5070996.

Swamidurai, R., Dennis, B., Kannan, U., 2014. Investigating the impact of peer code review and pair programming on test-driven development. In: IEEE SOUTHEASTCON 2014. IEEE, pp. 1–5. http://dx.doi.org/10.1109/SECON.2014.6950664.

Tao, Y., Kim, S., 2015. Partitioning composite code changes to facilitate code review. In: 2015 IEEE/ACM 12th Working Conference on Mining Software Repositories. In: MSR '15, IEEE Press, pp. 180–190. http://dx.doi.org/10.1109/MSR.2015.24.

Thompson, C., Wagner, D., 2017. 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. In: PROMISE, ACM, New York, NY, USA, pp. 83–92. http://dx.doi.org/10.1145/3127005.3127014.

Thongtanunam, P., Kula, R.G., Cruz, A.E.C., Yoshida, N., Iida, H., 2014. Improving code review effectiveness through reviewer recommendations. In: Proceedings of the 7th International Workshop on Cooperative and Human Aspects of Software Engineering. In: CHASE 2014, ACM, New York, NY, USA, pp. 119–122. http://dx.doi.org/10.1145/2593702.2593705.

Thongtanunam, P., McIntosh, S., Hassan, A.E., Iida, H., 2015a. Investigating code review practices in defective files: An empirical study of the qt project. In: 2015 IEEE/ACM 12th Working Conference on Mining Software Repositories. In: MSR '15, IEEE Press, pp. 168–179. http://dx.doi.org/10.1109/MSR.2015.23.

Thongtanunam, P., McIntosh, S., Hassan, A.E., Iida, H., 2016. 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. In: ICSE '16, ACM, New York, NY, USA, pp. 1039–1050. http://dx.doi.org/10.1145/2884781.2884852.

Thongtanunam, P., McIntosh, S., Hassan, A.E., Iida, H., 2017. Review participation in modern code review. Empir. Softw. Eng. 22 (2), 768–817.

Thongtanunam, P., Tantithamthavorn, C., Kula, R.G., Yoshida, N., Iida, H., ichi Matsumoto, K., 2015b. 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). IEEE, pp. 141–150. http://dx.doi.org/10.1109/SANER.2015.7081824.

Tymchuk, Y., Mocci, A., Lanza, M., 2015. Code review: Veni, ViDI, vici. In: 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER). IEEE, pp. 151–160. http://dx.doi.org/10.1109/SANER.2015.7081825.

Ueda, Y., Ihara, A., Hirao, T., Ishio, T., Matsumoto, K., 2017. 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). IEEE Press, pp. 207–213. http://dx.doi.org/10.1109/ISSREW.2017.32.

Ueda, Y., Ihara, A., Ishio, T., Matsumoto, K., Impact of coding style checker on code review - A case study on the openstack projects. In: 2018 9th International Workshop on Empirical Software Engineering in Practice (IWESEP). 2018, pp. 31–36.

Ueda, Y., Ishio, T., Ihara, A., Matsumoto, K., 2019. Mining source code improvement patterns from similar code review works. In: 2019 IEEE 13th International Workshop on Software Clones (IWSC). IEEE, pp. 13–19.

Uwano, H., Nakamura, M., Monden, A., Matsumoto, K.-i., 2006. Analyzing individual performance of source code review using reviewers' eye movement. In: Proceedings of the 2006 Symposium on Eye Tracking Research &Amp; Applications. In: ETRA '06, ACM, New York, NY, USA, pp. 133–140. http://dx.doi.org/10.1145/1117309.1117357.

Wang, M., Lin, Z., Zou, Y., Xie, B., 2019a. CoRA: Decomposing and describing tangled code changes for reviewer. In: Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering. In: ASE '19, IEEE Press, pp. 1050–1061. http://dx.doi.org/10.1109/ASE.2019.00101.

Wang, Q., Xia, X., Lo, D., Li, S., 2019b. Why is my code change abandoned? Inf. Softw. Technol. 110, 108–120.

Wang, C., Xie, X., Liang, P., Xuan, J., 2017. Multi-perspective visualization to assist code change review. In: 2017 24th Asia-Pacific Software Engineering Conference (APSEC). IEEE Press, pp. 564–569. http://dx.doi.org/10.1109/APSEC.2017.66.

Wen, R., Gilbert, D., Roche, M.G., McIntosh, S., BLIMP tracer: Integrating build impact analysis with code review. In: 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME). 2018, pp. 685–694.

Wohlin, C., 2014. Guidelines for snowballing in systematic literature studies and a replication in software engineering. In: Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering. In: EASE '14, Association for Computing Machinery, New York, NY, USA, http://dx.doi.org/10.1145/2601248.2601268.

Xia, X., Lo, D., Wang, X., Yang, X., 2015. 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). IEEE Press, pp. 261–270. http://dx.doi.org/10.1109/ICSM.2015.7332472.

Xia, Z., Sun, H., Jiang, J., Wang, X., Liu, X., 2017. A hybrid approach to code reviewer recommendation with collaborative filtering. In: 2017 6th International Workshop on Software Mining. IEEE, pp. 24–31. http://dx.doi.org/10.1109/SOFTWAREMINING.2017.8100850.

Yang, C., Zhang, X., Zeng, L., Fan, Q., Yin, G., Wang, H., 2017. An empirical study of reviewer recommendation in pull-based development model. In: Proceedings of the 9th Asia-Pacific Symposium on Internetware. In: Internetware'17, ACM, New York, NY, USA, pp. 14:1–14:6. http://dx.doi.org/10.1145/3131704.3131718.

Ying, H., Chen, L., Liang, T., Wu, J., 2016. Earec: Leveraging expertise and authority for pull-request reviewer recommendation in GitHub. In: 2016 IEEE/ACM 3rd International Workshop on CrowdSourcing in Software Engineering (CSI-SE). pp. 29–35. http://dx.doi.org/10.1109/CSI-SE.2016.013.

Yu, Y., Wang, H., Yin, G., Wang, T., 2016. Reviewer recommendation for pull-requests in github: What can we learn from code review and bug assignment? Inf. Softw. Technol. 74, 204–218.

Zampetti, F., Bavota, G., Canfora, G., Di Penta, M., 2019. A study on the interplay between pull request review and continuous integration builds. In: 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE, pp. 38–48.

Zanaty, F.E., Hirao, T., McIntosh, S., Ihara, A., Matsumoto, K., 2018. An empirical study of design discussions in code review. In: Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement. In: ESEM '18, Association for Computing Machinery, New York, NY, USA, http://dx.doi.org/10.1145/3239235.3239525.

Zanjani, M.B., Kagdi, H., Bird, C., 2016. Automatically recommending peer reviewers in modern code review. IEEE Trans. Softw. Eng. 42 (6), 530–543.

Zhang, X., Dorn, B., Jester, W., Pelt, J.V., Gaeta, G., Firpo, D., 2011. Design and implementation of java sniper: A community-based software code review web solution. In: 2011 44th Hawaii International Conference on System Sciences. IEEE Computer Society, pp. 1–10. http://dx.doi.org/10.1109/HICSS.2011.145.

Zhang, T., Song, M., Pinedo, J., Kim, M., 2015. Interactive code review for systematic changes. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering. In: ICSE '15, Vol. 1, IEEE Press, pp. 111–122. http://dx.doi.org/10.1109/ICSE.2015.33.

Nicole Davila received the M.Sc. degree in computer science from the Federal University of Rio Grande do Sul (UFRGS), Brazil. She is working toward a Ph.D. degree in computer science at UFRGS in Brazil. Her research interests include software maintenance and evolution, code review, and human factors in software engineering.

Ingrid Nunes is an Associate Professor at the Institute of Informatics, Universidade Federal do Rio Grande do Sul (UFRGS), Brazil. She obtained her Ph.D. in Informatics at the Pontifical Catholic University of Rio de Janeiro (PUC-Rio), Brazil. Her Ph.D. was in cooperation with King's College London (UK) and University of Waterloo (Canada). She is the head of the Prosoft research group, and her main research areas are software maintenance and evolution and self-adaptive systems.