Modern Code Reviews—Survey of Literature and Practice

DEEPIKA BADAMPUDI, MICHAEL UNTERKALMSTEINER, and RICARDO BRITTO,
Blekinge Institute of Technology, Sweden

Background: Modern Code Review (MCR) is a lightweight alternative to traditional code inspections. While secondary studies on MCR exist, it is u*a*nknown whether the research community has targeted themes that practitioners consider important.
Objectives: The objectives are to provide an overview of MCR research, analyze the practitioners' opinions on the importance of MCR research, investigate the alignment between research and practice, and propose future MCR research avenues.
Method: We conducted a systematic mapping study to survey state of the art until and including 2021, employed the Q-Methodology to analyze the practitioners' perception of the relevance of MCR research, and analyzed the primary studies' research impact.
Results: We analyzed 244 primary studies, resulting in five themes. As a result of the 1,300 survey data points, we found that the respondents are positive about research investigating the impact of MCR on product quality and MCR process properties. In contrast, they are negative about human factor– and support systems–related research.
Conclusion: These results indicate a misalignment between the state of the art and the themes deemed important by most survey respondents. Researchers should focus on solutions that can improve the state of MCR practice. We provide an MCR research agenda that can potentially increase the impact of MCR research.

CCS Concepts: • **General and reference** → **Surveys and overviews**; • **Software and its engineering** → **Software creation and management;**

Additional Key Words and Phrases: Modern code review, literature survey, practitioner survey

ACM Reference format:
Deepika Badampudi, Michael Unterkalmsteiner, and Ricardo Britto. 2023. Modern Code Reviews—Survey of Literature and Practice. *ACM Trans. Softw. Eng. Methodol.* 32, 4, Article 107 (May 2023), 61 pages.
https://doi.org/10.1145/3585004

1 INTRODUCTION

Software code review is the practice that involves the inspection of code before its integration into the code base and deployment. Software code reviews have evolved from being rigorous, co-located, and synchronous to lightweight, distributed, tool based, and asynchronous [34]. **Modern**

Ricardo Britto also with Ericsson AB.

We acknowledge that this work was supported by the Knowledge Foundation through the projects SERT – Software Engineering ReThought and OSIR Open-source inspired reuse (reference number 20190081) at Blekinge Institute of Technology, Sweden.

Authors' address: D. Badampudi, M. Unterkalmsteiner, and R. Britto, Blekinge Institute of Technology, Software Engineering Research and Education Lab Sweden, Valahallavägen 1, Karlskrona, 37141, Sweden; emails: {deepika.badampudi, michael.unterkalmsteiner, ricardo.britto}@bth.se.

1049-331X/2023/05-ART107 $15.00
https://doi.org/10.1145/3585004

Code Review (MCR) is a lightweight alternative to traditional code inspections [20] that focuses on code changes and allows software developers to improve code quality and reduce post-delivery defects [3, 7]. MCR is an essential practice in modern software development not only due to its contribution to quality assurance but also because it helps with design improvement, knowledge sharing, and code ownership [30].

The research interest on code inspections declined in the middle of the 2000s [25]. Due to the value of code reviews in general, it is reasonable to assume that the research focus has shifted to MCR. After over a decade of research on MCR, several initiatives were born to aggregate a body of knowledge on the increasing research of this essential quality assurance practice. To the best of our knowledge, we presented in our previous work [4] the first overview on the state of the art of MCR research. In our previous mapping study, we reported the preliminary results of systematically searching and analyzing the existing literature (based on titles and abstracts) and identified major research themes. Likely in parallel, other studies have also explored and made an attempt to aggregate the existing literature on MCR, either on particular aspects of the practice (refactoring-aware code reviews [16], benefits of MCR [30], MCR in education [22], reviewer recommendations [14]), or in general [18, 37].

Since there exists a considerable and diverse amount of research on the MCR practice, we were curious whether the research community has targeted themes that are also perceived as important by MCR practitioners. Similar investigations have been conducted in the past on software engineering research in general [13, 27] and requirements engineering research in particular [21].

The main goal of this study is therefore *to provide an overview of the different research themes on MCR, analyze practitioners' opinions on the importance of the research themes, and outline a roadmap for future research on MCR.* To achieve this goal, we extended our earlier work [4] by including publications up until the year 2021 and synthesizing the contributions of the 244 identified primary studies in MCR research. Then we constructed 47 statements that describe the research covered in the primary studies and surveyed 28 practitioners using the Q-Methodology [41] to gauge their perception on the statements representing the research conducted in this field. Finally, we compare the practitioners perception on the investigated themes in MCR research with the amount of publications and research impact of those themes. The main contributions of this article are as follows:

- **A comprehensive aggregation of research conducted on MCR research themes until and including 2021:** We identify potential gaps that researchers could address in the future and provide a summary on the state of the art in MCR research that can be useful for practitioners (e.g., to benefit from existing findings and solutions).
- **Level of alignment between MCR state of the art and practitioners' perception on the relevance of the MCR state of the art:** We assess the practitioners' perception on the relevance of the MCR state of the art represented by statements that summarize each topic in the MCR state of the art. We assess the alignment between what the research community has focused on the most and how MCR practitioners perceive its relevance. This analysis can help researchers to focus on themes that are deemed relevant by practitioners but do not have enough research coverage. We propose a research roadmap based mainly on the analysis of the reviewed primary studies and qualified by the responses from the survey.

The remainder of this article is structured as follows: Section 2 presents background on the MCR practice and relevant related work to this study. Section 3 describes the design of our research, which is followed by Sections 4 and 5, where we describe the mapping study and survey results, respectively. In Section 6, we compare the state of the art and practitioners' perspectives. Section 7 discusses our results and illustrates our MCR research roadmap. Finally, Section 8 presents our conclusions and view on future work.

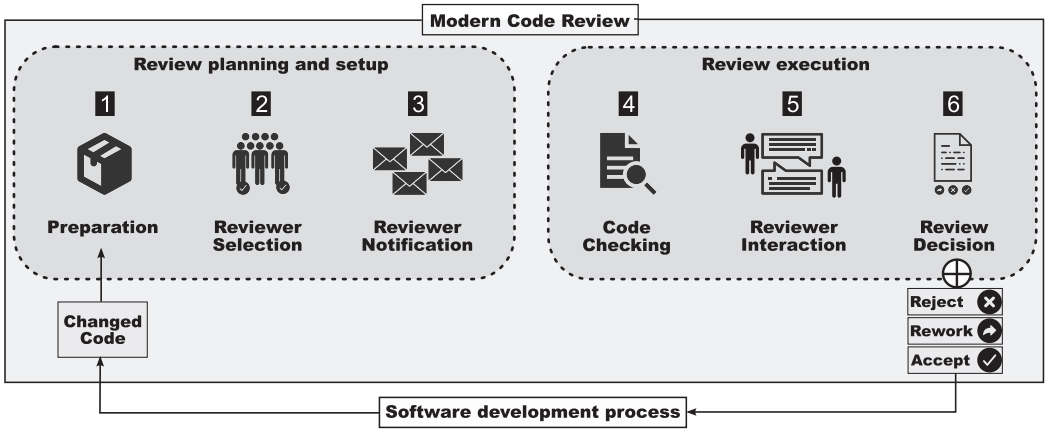


Fig. 1. Overview of steps in modern code reviews (adapted from Davila and Nunes [18]).

2 BACKGROUND AND RELATED WORK

In this section, we briefly revise the history of peer code reviews (2.1), illustrate the MCR process (2.2), and discuss related work surveying MCR literature (2.3) and practitioner surveys in SE in general (2.4), illustrating the research gap that this study aims to fill (summarized in 2.5).

2.1 Peer Code Review

It is widely recognized that the peer review of code is an effective quality assurance strategy [2, 25]. Formal code inspections were introduced in the mid-1970s by Fagan [20]. The formal code inspection process requires well-defined inputs (a software product that is ready for review), planning and organizing of review resources (time, location, expertise), execution of the review following guidelines that facilitate the detection of defects, and a synthesis of the findings, which are used for improvement [2]. Kollanus and Koskinen [25] have reviewed the research on code inspections between 1980 and 2008 and found a peak of publications between the late 1990s and 2004 (averaging 14 papers per year) with a strong decline between 2005 and 2008 (4 papers per year). This change in research interest coincides with the rise of MCR research, starting around 2007, which had a steady upward trend since 2011 [4]. Research on code inspections focused on reading techniques, effectiveness factors, processes, the impact of inspections, defect estimation, and inspection tools [25]. Interestingly, the tool aspect was the least researched one, with 16 of 153 studies (10%). MCR were borne of the need to perform lightweight yet efficient and effective quality assurance [3]. It is a technology-driven practice that complements **continuous integration and deployment (CI/CD)**, a method to frequently and reliably release new features. CI/CD also saw a rise in practical adoption and research interest around 2010 [35].

2.2 Modern Code Review

Figure 1 illustrates the two phases and main six steps in MCR, which are typically supported by tools that integrate with version control systems (e.g., Gerrit, GitHub, and GitLab). The main actors involved in MCR are the code author(s) and the reviewer(s). While there may be organizational, technical, and tooling differences between open source and commercial software development implementing MCR, the general steps are valid for both contexts. A significant difference of MCR in open source and commercial development is its perceived purpose. In open source development, reviewers focus on building relationships with core developers, while in commercial development, knowledge dissemination through MCR is more important [76].

In *Step 1*, the code author(s) prepare the code change for review, which usually includes a description of the intended modification and/or a reference to the corresponding issue recorded in a bug tracking system. When tools like Gerrit and GitHub are used, the change author creates a pull request. Questions that arise in this step are as follows: What is the optimal size of a pull request?, How can large changes be broken down into manageable pull requests?, and How should changes be documented?

In *Step 2*, the project/code owner selects one or more reviewers, typically using heuristics such as expertise in the affected code or time availability. Questions that arise in this step are as follows: What is the optimal number of reviewers?, Who is the best reviewer for a particular code change?, What is the optimal workload for a reviewer?, and How should code changes be prioritized for review?

In *Step 3*, the reviewer(s) are notified of their assignment, concluding the planning phase of MCR. Questions that arise in this step are as follows: How much time should be allocated to a review? and How should reviews be scheduled (batch, scattered throughout the work day/week)?

In *Step 4*, the reviewer(s) check the code changes for defects or suggest improvements. The procedure for this step is highly individualized and depends on tool support, company culture, and personal preference and experience. Typically, the reviewer inspects the code changes that are surrounded by unchanged (context) code and provide feedback to particular lines of code, as well as to the overall change. Questions that arise in this step are as follows: What information is needed and what are the best practices for an effective review?, What is the most effective way to describe findings and comments to code changes?, and Can the identification of certain defects or improvements be automated?

In *Step 5*, the reviewer(s) and author(s) discuss the code changes and feedback, often facilitated by tools that enable asynchronous communication and allow referencing code and actors. This interaction creates a permanent record of the technical considerations regarding the change that emerged during the review. Questions that arise in this step are as follows: What are the key considerations for effective communication between reviewer(a) and author(s)?, How can endless (unprofessional) discussions be avoided?, and How can consensus be facilitated?

In *Step 6*, the change is rejected, accepted, or sent back to the author(s) for refinement. The decision process can be implemented with majority voting or rests upon the judgement of the project/code owner. Questions that arise in this step are as follows: To what extent can the decision process be automated? and What can we learn from accepted/rejected changes that can be used to accept/filter high/low quality patches earlier?

The questions above, together with other questions, are investigated in the primary studies identified in our study but also in the literature surveys presented in the related work, which is discussed next.

2.3 Literature Surveys

While surveys on software inspections in general [2, 25, 26], checklists [10], and tool support [28] have been conducted in the past, surveys on MCR have only recently received an increased interest from the research community (since 2019). We identified six studies, besides our own, that mentioned MCR in their review aim within a very short time frame (2019–2021). Table 1 summarizes key data of these reviews.

To the best of our knowledge, our systematic mapping study [4] presented the first results on the state of the art in MCR research (April 2019). We identified and classified 177 research papers covering the time frame between 2007 and 2018. The goal of this mapping study was to identify the main themes of MCR research by analyzing the papers' abstract. We observed an increasing trend of publications from 2011, with the major themes related to MCR processes, reviewer

Table 1. Comparison of Review Studies on MCR

Review by	Time-frame	Studies	Focus	Research questions
Badampudi et al. [4]	2007–2018	177	MCR in general	RQ1. What topics of modern code reviews are investigated? RQ2. How were the aspects in R1 investigated?
Coelho et al. [16]	2007–2018	13	Refactoring-aware code review	RQ1. What are the most common research topics? RQ2. What are the methods/techniques/tools proposed? RQ3. What are the validation methods applied?
Nazir et al. [30]	2013–2019	51	Benefits of MCR	RQ1. What are the real benefits of performing MCR? RQ2. How identified benefits can be grouped into relevant themes?
Indriasari et al. [22]	2013–2019	51	MCR as a teaching vehicle	RQ1. What are the reported motivations for conducting peer code review activities in tertiary-level programming courses? RQ2. How has peer code review been practiced in tertiary-level programming courses? RQ3. What are the main benefits and barriers to the implementation of peer code review in tertiary-level programming courses?
Çetin et al. [14]	2009–2020	29	Code reviewer recommendations (CCR)	RQ1. What kind of methods/algorithms are used in CRR studies? RQ2. What are the characteristics of the datasets in CRR studies? RQ3. What are the characteristics of the evaluation setups used in CRR studies? RQ4. Are the models proposed in CRR studies reproducible? RQ5. What kind of validity threats are discussed in CRR studies? RQ6. What kind of future works are discussed in CRR studies?
Wang et al. [37]	2011–2019	112	Benchmarking MCR	RQ1. What contributions and methodologies does code review (CR) research target? RQ2. How much CR research has the potential for replicability? RQ3. What metric and topics are used with CR studies?
Davila and Nunes [18]	1998–2019	138	MCR in general	RQ1. What foundational body of knowledge has been built based on studies of MCR? RQ2. What approaches have been developed to support MCR? RQ3. How have MCR approaches been evaluated and what were the reached conclusions?
This study	2007–2021	244	MCR in general	RQ1. Which MCR themes have been investigated by the research community? RQ2. How do practitioners perceive the importance of the identified MCR research themes? RQ3. To what degree are researchers and practitioners aligned on the goals of MCR research?

characteristics and selection, tool support, source code characteristics and review comments. In this article, we update the search to include studies published including 2021, and we considerably deepen the classification and analysis of the themes covered in MCR research, reporting on the major contributions, key takeaways, and research gaps. Furthermore, we survey practitioners opinions on MCR research to juxtapose research trends with the perspective from the state of practice.

Briefly after our mapping study, Coelho et al. [16] published their mapping study on refactoring-aware code reviews (May 2019). They argue that MCR can be conducted more efficiently if reviewers are aware of the type of changes and focus therefore their search on methods/techniques/tools that support the classification of code changes. They identified 13 primary studies (2007–2018), of which 9 are unique to their review. This could be due to the inclusion of "code inspection" in their search string, resulting in papers that are not related to MCR (e.g., Reference [1, 15]), even though Coelho et al. mentioned MCR explicitly in their mapping aim.

Nazir et al. [30] published preliminary results of a systematic literature review on the benefits of MCR in January 2020. They identified 51 primary studies, published between 2013 and 2019, and synthesized nine clusters of studies that describe benefits of MCR: software quality improvement, knowledge exchange, code improvement, team benefits, individual benefits, ensuring documentation, risk minimization, distributed work benefits, and artifact knowledge dissemination.

Indriasari et al. [22] reviewed the literature on the benefits and barriers of MCR as a teaching and learning vehicle in higher education (September 2020). They identified 51 primary studies, published between 2013 and 2019, and found that skill development, learning support, product

Table 2. Comparison of Studies Analyzing Practitioners' Perception of Research

Survey by	Source of research	Practitioner perception
Lo et al. [27]	General Software Engineering (ICSE and ESEC/FSE)	71% positive
Carver et al. [13]	Empirical Software Engineering	67% positive
Franch et al. [21]	Requirements Engineering	70% positive

quality improvement, administrative process effectiveness, and the social implications are the main drivers for introducing peer code reviews in education. Analyzing the set of primary studies they included, we observe that this review has the least overlap of all with the other reviews. This is likely due to the particular focus on peer code reviews in education, which was explicitly excluded, for example, in our study.

Çetin et al. [14] focused in their systematic literature review on the aspect of reviewer recommendations in MCR (April 2021). They identified 29 primary studies, published between 2009 and 2020, and report that the most common approaches are based on heuristics and machine learning, are evaluated on open source projects but still suffer from reproducibility problems, and are threatened by model generalizatibility and data integrity.

We discuss now the two reviews of MCR that are closest in scope and aim to our work and illustrate similarities in observations and the main differences in contributions between the reviews. Wang et al. published a pre-print [36] on the evolution of code review research (November 2019), which has been extended, peer reviewed, and published in 2021 [37]. They identified 112 primary studies, published between 2011 and 2019. Similarly to our results (see Figure 5(b)), they observe a predominant focus on evaluation and validation research, with fewer studies reporting experiences and solution proposals for MCR. The unique contributions of their review are the assessment of the studies' replicability (judged by availability of public datasets) and the identification and classification of metrics used in MCR research. The former is important, as it allows other researchers to conduct replication studies and the latter helps researchers to design studies whose results can be benchmarked. Compared to Wang et al. our review of the themes studied in MCR research is more granular (9 vs. 47), and we provide a narrative summary of the papers' contributions.

Finally, Davila and Nunes [18] performed a systematic literature review with the aim to provide a structured overview and analysis of the research done on MCR (2021). They identified 138[1] primary studies published between 1998 and 2019 and provide an in-depth analysis of the literature, classifying the field into foundational studies (which try to understand the practice), proposal studies (improve the practice), and evaluation studies (measure and compare practices). Their synthesis provides excellent insights in the MCR state of the art with findings that are interesting for researchers as well as practitioners.

2.4 Practitioner Surveys

Several studies have investigated the practical relevance of software engineering by surveying practitioners (see Table 2). Lo et al. [27] were interested to gauge the relevance of research ideas presented at the International Conference on Software Engineering (ICSE) and Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), two premier conferences in software engineering. They summarized the key contributions of 571 papers and let practitioners (employees at Microsoft) rate the ideas on a scale from essential to worthwhile to unimportant to unwise. Overall, 71% of the ratings were positive. However, they found no correlation between academic impact (citation count) and relevance score. Carver et al. [13] replicated Lo et al.'s study with research contributions from a different

[1]When we analyzed their primary studies, we found they had one duplicated paper in their set, which included 139 studies.

conference (ACM/IEEE International Symposium on Empirical Software Engineering and Measurement), targeting a wider and international audience of practitioners. They also investigated what practitioners think SE research should focus on. Their conclusions are similar to the ICSE and ESEC/FSE study, with 67% overall positive ratings for the research, and no correlation between academic impact and relevance score. Furthermore, they found that the research at the conference addresses the needs expressed by the practitioners quite well. However, they highlight the need for improving the discoverability of the research to enable knowledge transfer between research and practice.

Finally, Franch et al. [21] surveyed practitioners in the field of requirements engineering and found mostly positive ratings (70%) for the research in this area. The practitioners justifications for positive ratings are related to the perceived problem relevance and solution utility described in research. The requirements engineering activities that should receive the most attention from research, according to the practitioners needs, are traceability, evaluation, and automation.

2.5 Research Gap

While recent years have seen several literature surveys on MCR, we know very little about how this research is perceived by practitioners. Looking at the research questions shown in Table 1, one can observe that a few studies sought to investigate how MCR techniques [14] and approaches [18, 37] have been evaluated. None has yet studied practitioner perception of MCR research, even though general software and requirements engineering research has been the target of such surveys (see Table 2). In this study, we focus the literature review on identifying the main themes and contributions of MCR research, summarizing that material in an accessible way in the form of evidence briefings, and gauging practitioners perceptions of this research using a survey. Based on the results of this two data collection strategies, we outline the most promising research avenues, from the perspective of their potential impact on practice.

3 RESEARCH DESIGN

Based on our main goal introduced in Section 1, we formulated the following research questions.

- **RQ1** Which MCR themes have been investigated by the research community?
  - **RQ1.1** How was the research on MCR conducted and in which context?
  - **RQ1.2** What is the quality of the conducted research?
  - **RQ1.3** Which were the most investigated MCR themes and what were the major findings of the MCR research?
- **RQ2** How do practitioners perceive the importance of the identified MCR research themes?
- **RQ3** To what degree are researchers and practitioners aligned on the goals of MCR research?

To answer these questions, we followed a mixed-methods approach. We conducted a systematic mapping study to answer RQ1 and its sub-questions. In our previous study [4], we presented preliminary results of the mapping study with the review period until 2018. To answer RQ2, we created statements representing the primary studies' research objectives. We then created the survey questionnaire using the statements representing the primary studies until 2018. We conducted the survey using the Q-Methodology, collecting practitioners' opinions on the importance of the statements representing the MCR research topics. We extended the mapping study period to 2021, analyzed the new primary studies (2018 onwards), and mapped to the statements representing them. Finally, we answer RQ3 by comparing both the frequency and research impact of MCR research with its perceived importance by practitioners. The research design is depicted in Figure 2. All research material we produced in this study (search results, selected primary studies, data extraction, evidence briefings, survey material, and citation analysis) is available online [5, 6]. In the

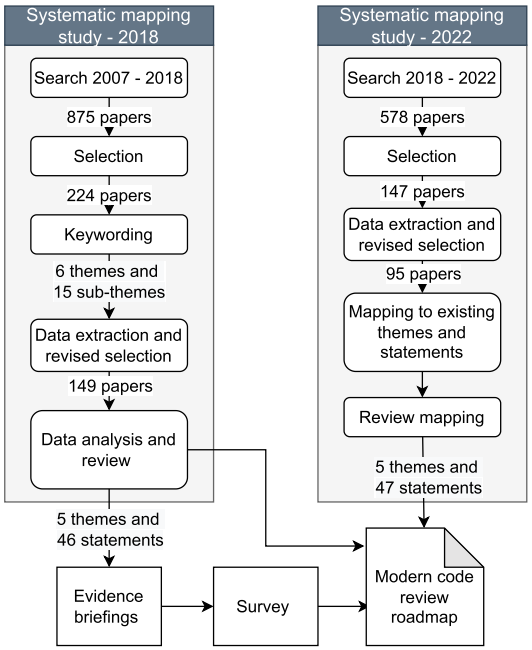


Fig. 2. Research methodology followed in this study.

remainder of this section, we illustrate the research methodologies we employed to answer our research questions.

3.1 Systematic Mapping Study

We followed the guidelines for conducting systematic mapping studies by Petersen et al. [32], which include the following steps: (1) definition of review questions, (2) conduct search for primary papers, (3) screening relevant papers, (4) keywording of abstracts, and (5) data extraction and mapping of studies. As seen in Figure 2, the **systematic mapping study (SMS)** was conducted in two phases: one study until the review period 2018 (SMS2018) and another from review period until 2021 (SMS2022). The review questions are represented by the sub-questions RQ1.1, RQ1.2, and RQ1.3. Below we provide details on the execution of the consolidated mapping study (including the initial (SMS2018) and extend mapping study (SMS2022)).

Search. Databases: The following databases were selected in SMS2018 and SMS2022 studies based on their coverage of papers: Scopus, IEEE Explore, and ACM Digital Library. Scopus indexes a wide range of publishers such as ScienceDirect and Springer.
Search Strings: We used the keywords listed in Table 3(a) to search in the three databases, using the search strings shown in Table 4, combining each keyword with a logical "OR" operation and adding a wildcard (*) operator. We intentionally did not include the term "inspection" due to its association with traditional code inspections, which researchers have reviewed in the past (see discussion at the beginning of Section 2.3).
Search Scope: The search results, including SMS2018 and SMS2022 studies, are presented in Table 3(b). To consider full years results we included papers until the year 2021 in the SMS2022 study. All the search results were exported into a csv file. We identified duplicates using Microsoft Excel's conditional formatting and applying the duplicate values rule to all the titles. We manually checked the formatted duplicate entries before removing them from the list.

Table 3. Search Keywords and Results

ID	Keyword
K1	code review
K2	patch accept
K3	commit review
K4	pull request
K5	modern code inspect

(a) Keywords used in search strings

Database	Papers
Scopus	1412
IEEE Explore	497
ACM Digital Library - Title	104
ACM Digital Library - Abstract	291
Total	2392
Total after removing duplicates and noise	1453

(b) Search results from each database

Table 4. Search Strings Used in Different Databases

Database	Search string
Scopus	TITLE-ABS-KEY (“K1*” OR “K2*” OR “K3*” OR “K4*” OR “K5*”)
IEEE Explore	(“All Metadata”:“K1*” OR “K2*” OR “K3*” OR “K4*” OR “K5*”)
ACM Digital Library - Title and abstract	acmdlTitle:(“K1*” “K2*” “K3*” “K4*” “K5*”) recordAbstract:(“K1*” “K2*” “K3*” “K4*” “K5*”)

Table 5. Inclusion and Exclusion Criteria

	ID	Criterion
Inclusion	I1	Papers related to MCR in general.
	I2	Papers discussing source code (including test code) review done on a regular basis.
	I3	Papers discussing MCR-related aspects (reviewer selection, benefits, outcomes, challenges, motivations, etc.)
	I4	Papers proposing solutions for particular MCR activities.
Exclusion	E1	Papers not discussing MCR or the subject of investigation is not MCR process.
	E2	Papers that do not discuss the implications of a solution on the MCR process.
	E3	Papers that discuss MCR in education.
	E4	Papers not in English and without accessible full text.
	E5	Panel abstracts and proceedings summaries.
	E6	Short papers without results and symposium summarising results of other published papers.

Selection. The search results were reviewed based on a defined set of inclusion and exclusion criteria (Table 5).

SMS2018 Selection: Before we started the selection process, we conducted a pilot selection on randomly selected papers from the result set. All three authors performed an independent decision on whether the paper should be included, excluded, or tentatively included (we decided to be rather inclusive and exclude a paper later based on reading the full text). During the first pilot on 20 papers, we noticed a paper on test case review. We refined the inclusion criterion I2 (see Table 5) to add test code review as well. Some papers discussed approaches to support the MCR process to make it more efficient, for example, by selecting a relevant reviewer. Therefore, we added a specific inclusion criterion related to the MCR process (I4). As one of the goals of our study is to understand practitioners’ perception on MCR research, we decided to only include studies focusing on MCR practice (including open source). Therefore, we excluded papers that discuss MCR in education (E3). We modified exclusion criterion E1 to emphasize the subject of the investigation, i.e., we only include papers where the process of MCR is under investigation. We also came across papers that discuss solutions that might benefit, among other things, the MCR process, without discussing the implications of the approach on the code review process itself (e.g., defect prediction). As a result, we excluded such papers and added exclusion criterion E2. We conducted a second pilot study on 20 additional papers using the revised criteria. As a result, we achieved better understanding of the selection criteria. We decided therefore to distribute the selection of the remaining papers among all three authors equally. In cases where more that one version of a paper was available (e.g., a conference paper and a journal extension), we selected the most recent version.

SMS2022 Selection: After conducting the survey, we extended our mapping study. We conducted another pilot study on 30 random studies to evaluate if our selection (inclusion/exclusion) criteria

Table 6. Data Extraction Form

Type	Name	Description
Meta-data	Authors	The authors of the paper.
	Publication type	Conference or journal.
	Year	Publication year of the paper.
	Citations	To evaluate the research impact we count the paper's Google Scholar citations (as of June 2021).
Content	Research facet	Based on Wieringa's [38] classification: evaluation research, solution proposals, validation research, philosophical papers and experience papers.
	Rigor and relevance	To evaluate the rigor and relevance [23] of the paper we extract the following aspects. (a) Rigor: context, study design and validity threats. (b) Relevance: subjects involved in the study, context, scale and research method.
	Main contribution	A summary (either verbatim from the paper or formulated by the extractor) of the paper's main contribution and results.

need updating. We were consistent to a large degree in the pilot study and found we were able to make the selection based on our initial selection criteria. Of 30 papers, we disagreed on 6 papers. By disagreement, we mean that one of the authors decided to exclude the paper while the other author included the paper. We agreed to revisit the papers before making the final decision. After revisiting the papers, we eventually decided to exclude them. We divided the remaining papers and continued the selection process independently.

Keywording. **SMS2018 Keywording:** The goal of keywording the abstracts is to extract and classify the main contribution of the selected primary studies. We performed the keywording during the selection process. The keywording process resulted in a preliminary grouping into six main, and 15 studied sub-aspects in MCR research, published in our previous work [4]. The result of the keywording process was used to create initial themes and to identify extraction items.
SMS2022 Keywording: We used the identified themes in the SMS2018 study to classify the primary studies in SMS2022; therefore, we did not need any keywording process.

Data extraction. We extracted the data items shown in Table 6 in both mapping studies.
SMS2018 Data extraction: We used the preliminary results (studied aspects in MCR research) as an input to extract additional items related to the main contribution of the primary studies. For example, we extracted the "techniques used" and "purpose" in papers providing tool support and the "techniques used" and "selection criteria" as additional extraction items for papers providing reviewer recommendation support. Similarly to the paper selection, we planned to distribute the data extraction work among the three authors of this paper. Hence, to align our common understanding of the data extraction form, we conducted two pilots.

Data extraction pilot 1: Before starting the pilot process, we reviewed the rigor and relevance criteria provided by Ivarsson and Gorschek [23]. Assessing the research method in the relevance dimension of primary studies that analyze repositories was not straightforward. For evaluating the relevance of the research methods, we agreed that the tools/solutions or findings from primary studies should be validated or evaluated by the users in the field to get a high score. Results based on solely analyzing repositories are not enough; other sources such as interviews/surveys should be conducted. Once we had an understanding of the criteria we piloted the extraction process using six primary studies where each author independently extracted the data. All the extracted items were consistent among the authors, except for the type of the subjects in the relevance, and study design in the rigor dimension. For study design, the extracted values were different for one paper and the difference was resolved in a meeting. For the type of subjects, we decided to give 0 to subjects if no subjects are involved or if the main findings of the paper are not discussed with any subjects. It is important to evaluate the relevance of the findings with users involved in code review; therefore, we give a high score when subjects are involved to corroborate the findings. We

calculated the **inter rater agreement (IRA)** for each of the rigor and relevance aspects. For pilot 1 the average IRA for all aspects was 82%.

Data extraction pilot 2: Based on the updated description of the rigor and relevance criteria, we conducted another pilot study on 11 papers. In the second pilot study, there was higher agreement in the extracted items. In particular, the inter rater agreement for relevance of subjects was increased considerably. For pilot 2 the average IRA for all aspects increased to 88%.

After piloting the data extraction, we divided the data extraction of included studies until year 2018 with 20% overlap among the authors. The average IRA between the first and second authors was 95% and 86% between first and third authors. For the second and third authors, the IRA was 75%; however, this low percentage is mainly due to a conflict in one paper.

SMS2022 Data extraction: We used the same extraction form as in SMS2018. We independently conducted the extraction of the studies identified in SMS2022 as we had a good understanding of the extraction items.

Data analysis. **SMS2018 Data analysis:** Using a deductive approach, we used thematic analysis [17] to categorize the primary studies into themes. The main contribution extracted from the primary studies was used to generate themes. For example, if the paper's main contribution is to provide solution support for reviewer selection, then we assign the paper to the "solution support" theme. We divided the primary studies based on the studies aspects identified in the keywording process among all three authors to generate themes. For example, the first author identified themes for all the primary studies related to the MCR process and studies investigating source code and review comments. The second author identified themes within the primary studies providing tool support, and finally, the third author analyzed the primary studies providing reviewer recommendations. We then followed a review process where two other authors reviewed each paper classified by one author. Based on the discussions in the review process, we moved the primary studies into different themes when needed. This process continued until all three authors reached a consensus.

SMS2022 Data analysis: We mapped the contributions of the primary studies identified in SMS2022 study to existing themes. All authors were involved in the mapping process. We reviewed the mapping process to ensure that the mapping was done fairly and not forced into existing themes.

Evidence briefings. Evidence briefings are a technology transfer medium that summarizes the research results in a practitioner-friendly format [12]. We created evidence briefings based on the main findings of the primary studies identified in SMS2018. We provided the link to the evidence briefings at the end of the survey allowing the practitioners to read more on the themes that they find most interesting. The evidence briefings are available online [5].[2]

3.2 Q-Methodology Survey

We chose Q-Methodology as a data collection and analysis approach, since we were interested in understanding the viewpoint of practitioners on the numerous modern code review research topics we have identified in our mapping study. Q-Methodology allows us to collect viewpoints on concepts that might share underlying common factors and that are brought to the surface by revealing relations between concepts instead of rating these concepts in isolation [8]. The factors are identified by analyzing the subjective opinions of individuals, not facts, revealing common viewpoints within the surveyed community [9]. A strength of the Q-Methodology is that it can

[2] https://rethought.se/research/modern-code-reviews/ is the original link we shared with the participants of the survey. We created the Zenodo record for archival purposes.

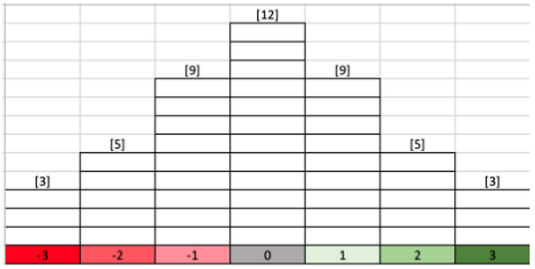


Fig. 3. Q-Sort structure: The number in the square brackets represents the number of statements per rating.

provide insights on the matter of study with a relatively low number of respondents, compared to conventional surveys [19].

Q-Methodology consists of the following seven steps [41]:

(1) Defining the concourse: The concourse consists of all the statements on a particular topic, which is in our case, modern code reviews. In our systematic mapping study, we have extracted the primary study's main contributions and research objective to represent the concourse.

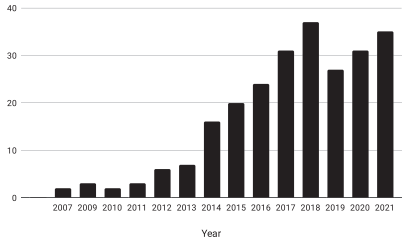
(2) Defining the Q-Set: The Q-Set is a sample of statements drawn from the concourse. These statements are formulated in a way such that it is possible to form an opinion about the content of the statements. The statements should not be factual, i.e., should not present a truth such as "the earth revolves around the sun" but rather evoke agreement or disagreement. Using our mapping study (SMS2018), we formulated statements from the papers' research objective in the form of "It is important to investigate ... [research statement]." We consolidated overlapping research objectives into single statements, i.e., the number of statements (46) < number of primary studies (149). All authors were responsible for creating the statements from the primary studies in SMS2018. In a review process, we revised the mapping whenever necessary. We followed the same mapping process for the primary studies identified in SMS2022. We list the statements and the frequency we encountered the corresponding research objective in our mapping study in Table 7.

(3) Defining the P-Set: The P-Set represents the survey participants. We sent the survey to practitioners in our professional network who are involved in software development and the code review process. We sent the survey link to our contact persons in 17 partner companies, who distributed the survey within the company. In addition, we published the survey link on our LinkedIn profile. The participants (P-Set) rank the statements (Q-Set) on a scale of +3 to −3 representing their agreement level. The details on the P-Set, i.e., the demographic information of participants is provided in Table 8.

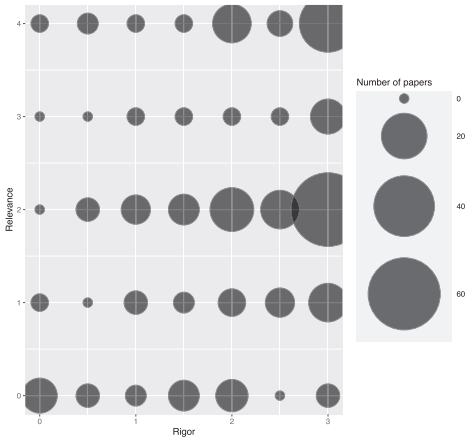
(4) Defining the Q-Sort structure: The participants were asked to place a certain amount of statements in each rating. The number of statements that can be placed in each rating is shown in Figure 3. For example, the participants can file nine statements into the rating +1 and −1 rating respectively.

(5) Conducting the Q-Sorting process: In this step, the participants in the P-Set rate the statements in Q-Set. We adapted an existing tool [11] that implements the Q-Sort in a web application. After we piloted the tool with three industry practitioners, we improved the tools user interface and description of steps. The updated tool[3] and the survey,[4] consisting of four steps, can be accessed online. In step 1, we provide an introduction to the study purpose so that the participants are aware of the importance of their input. In step 2, we ask the participants to place the statements

[3]https://github.com/DeepikaBadampudi/qmethod.
[4]https://mcrq.rethought.se/#/step1.



(a) Year-wise count of publication types



(b) Total scores of rigor and relevance

Fig. 4. Frequencies of project types and research facets in primary studies.

in agree, disagree, and neutral piles. In step 3, the participants place the statements in the Q-Sort structure depicted in Figure 3. In this way, the participants read the statements two times; once when they put them in the agree/disagree/neutral piles and then when they file them into the Q-Sort structure. In step 4, we ask the participants to provide explanations for the placement of the six statements that they filed in the extreme agreement (+3) and disagreement (−3) piles.

(6) Conducting Factor Analysis: In the analysis, all similar viewpoints are grouped into a factor. Q-Methodology automatically classifies the viewpoints and provides a list of statements along with z-scores and factors scores for each factor. In other words, a list of statements that differentiate a viewpoint is generated along with the scores that indicate how the viewpoints differ. We elaborate on how the factors are generated in Section 5.3.

(7) Conducting Factor Interpretation: This last step refers to the interpretation of the factors by considering the statements and their scores in that factor and the participants' demographic information. All the statements in each factor along with the ratings were reviewed to understand the nuances of each viewpoint (factor). The first author formulated interpretations of each factor, considering the participants explanations for the statements rated with high agreement and disagreement. The interpretation was then reviewed by the second and third author. The review process resulted in minor reformulations. The factor interpretation is provided in Section 5.3.

4 MAPPING STUDY RESULTS

In this section, we report on the results that answer RQ1–*Which MCR themes have been investigated by the research community?* and its sub questions, based on 244 identified primary studies.

4.1 Research Venue, Type, Quality, and Context

We see a steady growth in the number of publications on modern code review as seen in Figure 4(a). In this section, we provide details on the quality and the context of the primary studies.

Quality: Rigor and relevance - The description of the context, study design, and validity threats are used to determine the rigor. We assign a score of 1 for each rigor aspect when all details are reported, 0.5 when partial details are reported and 0 when no details are reported. The relevance is scored between 0 and 1 for the relevance of subjects involved, context, research method, and scale. Once we scored each quality aspect, we calculated the total of rigor and relevance for each primary study. The maximum rigor score is 3 and relevance score 4. The total scores of rigor and

(a) Projects types

(b) Research facet and project types

Fig. 5. Frequencies of project types and research facets in primary studies.

relevance shown in Figure 4(b). The scores of rigor and relevance are shown on the x- and y-axes, respectively. The bubble represents the number of papers for a given score.

As seen in Figure 4(b), the biggest bubble representing (26%) of the primary studies has high rigor total (score = 3) and low (score = 2) relevance. Only 14% of the primary studies have both high rigor (3) and relevance (4). Overall relevance is low, because most of the primary studies did not include any human subjects in the research design. Only 32% of the primary studies involved human subjects in their study and evaluated their findings in a realistic environment. Most primary studies analyzed repositories without triangulating the findings from other sources such as interviews or surveys.

Context - The type of projects investigated in the primary studies is shown in Figure 5(a). Most of the primary studies (59%) investigated **open source (OSS)** projects. Less than 20% of the primary studies investigated proprietary projects. About 10% of the papers did not base their findings on any projects; these papers are mainly solutions papers.

Evaluation research investigates a problem or implements a solution in practice. As seen in Figure 5(b), 67% of evaluation is done in open source projects and 26% in proprietary projects. A small percentage (4%) of evaluation is done in academic projects. Four papers did not describe the context of the evaluation. The same trend can be seen for validation research coverage. "Validation research investigates the properties of a solution proposal that has not been implemented in practice" [38]. Most of the validation is done in open source projects, and only four studies are validated in a proprietary project. Solution proposals are primary studies that propose a solution and argue for its relevance. Unlike validation research, there is no thorough evaluation or validation of the solution in solution papers. One of the experience papers is based on a proprietary project; however, the project type is not discussed in the other two philosophical and experience papers.

4.2 MCR Themes and Contributions

We grouped the 244 primary studies in five themes. In the remainder of this section, we summarize their main contributions.

4.2.1 Support Systems for Code Reviews. This theme includes primary studies that contribute to solutions to support the MCR process, such as review prioritization, review automation, and reviewer recommendation.

Reviewer recommendations. A majority of papers on this theme focus on proposing tools to recommend reviewers and validate their approaches using historical data extracted from open source projects.

Most approaches recommend code reviewers based on the similarity between files modified or reviewed by each developer and the files of a new pull request (path similarity) [87, 106, 122, 144, 145, 158, 187, 198, 204, 222, 230, 238, 242, 268, 271, 273, 275, 279]. Some studies include other

predictors such as previous interactions between submitter and potential reviewers [144, 145, 187, 273, 275, 285], pull request content similarity [145, 211, 268, 275], contribution to similar files [122, 158, 231, 274], review linkage graphs [132], and developer activeness in a project [144, 145, 158, 211, 271]. Another popular predictor to recommend code reviewers is the similarity between the content of previous and new pull requests [145, 152, 166, 268–270, 275, 276].

In one study, the authors used participants' preferences in review assignment [263], while in the other study, the authors combined the metadata of pull requests with the metadata associated with potential reviewers [92]. Another study focuses on detecting and removing systematic labeling bias to improve prediction [235]. Another interesting direction is to focus recommend reviewers that will ensure code base knowledge distribution [86, 176, 207]. Finally, some studies have included balancing review workload as an objective [43, 49, 86, 230].

In relation to how the predictors are used to recommend code reviewers, many employ traditional approaches (e.g., cosine similarity), while some use machine learning techniques, such as Random Forest [92], Naive Bayes [92, 235], Support Vector Machines [144, 276], Collaborative Filtering [87, 230], and Deep Neural Networks [222, 274], or model reviewer recommendation as an optimization problem [43, 86, 187, 207, 211].

The performance of the identified approaches varies a lot and is often measured using Accuracy [92, 122, 144, 204, 230, 238, 270], Precision and Recall [106, 145, 166, 187, 204, 222, 275, 276, 279, 285], or Mean Reciprocal Rank [106, 144, 230, 230, 231, 235, 268, 279]. Of the identified studies, only a few [49, 158, 198, 230] have evaluated code reviewer recommendation tools in live environments. Instead, the majority of the studies measures performance (accuracy, precision, recall, and mean reciprocal rank) by comparing the actual list of reviewers present in historical data with the list of developers recommended by their respective approaches.

One study focuses on identifying factors that should be accounted for when recommending reviewers [223], such as the number of files and commits in a pull request, pull requester profile, previous interactions between contributors, previous experience with related code, and ownership of modified code are factors related to how code reviewers are selected.

Finally, only two studies evaluate whether reviewer recommendation really adds any value [158, 230], with mixed results.

Understanding the code changes that need to be reviewed. Refactoring changes the code structure to improve testability, maintainability, and quality without changing its behavior. Supporting the review of such changes has been the focus of refactoring-aware tools. Refdistiller aims at detecting behavior-changing edits in refactorings [46]. The tool uses two techniques: (a) a template-based checker that finds missing edits and (b) a refactoring separator that finds extra edits that may change a program's behavior. In a survey of 35 developers, they found that it would be useful to differentiate between refactored and behavior-changing code, making reviews more efficient and correct. ReviewFactor is a tool able to detect both manual and automated refactorings [111]. The evaluation of the tool showed that it can detect behavior-changing refactorings with high precision (92%) and recall (94%). RAID [79] aims at reducing the reviewers' cognitive effort by automatically detecting refactorings and visualizing information relevant for the refactoring to the reviewer. In a field experiment, professional developers reduced the number of lines they had to review for move and extraction refactorings. CRITICS is an interactive approach to review systematic code changes [280]. It allows developers to find changes similar to a specified template, detecting potential mistakes. The evaluation indicates that (a) six engineers who used the tool would like to have it integrated in their review environment, and (b) the tool can improve reviewer productivity, compared to a regular diffing tool. A study at Microsoft proposes a solution to automatically cluster similar code changes [58]. The clusters are then submitted for review. A preliminary user study suggests that the understanding of code did indeed improve when

the changes were clustered. Similarly, SIL identifies similar code changes in a pull request and flags potential inconsistent or overlooked changes [51]. In an inspection of 453 code changes in open source projects, it was found that up to 29% of the changes are composite, i.e., address different concerns [234]. Decomposing large pull requests into cohesive change sets could therefore be valuable. A controlled experiment study found that change decomposition leads to fewer wrongly reported issues [94]. ChgCutter [118] provides an interactive environment that allows reviewers to decompose code changes into atomic modifications that can be reviewed and tested individually. Professional developers suggested in a interview study that the approach helps to understand complex changes. CoRA automatically decomposes commits in a pull request into clusters of related changes (e.g., bug fixes, refactorings) and generates concise descriptions that allows users to better understand the changes [260]. Another idea to reduce review effort is to prioritize code that is likely to exhibit issues. One approach is to train a Convolutional Neural Network with old review comments and source code features to identify code fragments that require review [229]. Similarly, CRUSO classifies to be reviewed code by identifying similar code snippets on StackOverflow and analyzing the corresponding comments and metadata, leveraging crowd-knowledge [148, 217].

Other research looked into the order in which changed files should be presented to the reviewer to achieve an effective review process [63]. The study proposes the following main principle for the ordering: related change parts should be grouped as closely as possible. Another contribution to improve the understanding of changed code suggests identifying the "salient" class, i.e., the class in which the main change was made and that affects changes in other dependent classes [136]. The authors hypothesize that reviews could be more efficient if the salient class would be known, making the logic of the commit easier to understand. A preliminary evaluation (questionnaire-based) with 14 participants showed that the knowledge about the salient class improves the understanding of a commit. A follow-up study with a broader evaluation confirms these results [137]. A similar idea is implemented in BLIMP tracer, which inspects the impact of changes on a file level rather than on a class level [264]. The tool was evaluated with 45 developers, and it improved speed and accuracy of identifying the artifacts that are impacted by a code change. SEMCIA was developed to reduce noise in change impact analysis and uses semantic rather than syntactic relationships. This approach reduces false positives by up to 53% and reduces the change impact sets considerably [121]. MultiViewer is a code change review assistance tool that calculates metrics to better understand the change effort, risk, and impact of a change request [257]. A step further goes the approach implemented in the tool GETTY, which aims at providing meaningful change summaries by identifying change invariants through analyzing code differences and test run results [173]. With GETTY, reviewers can determine if a set of code changes has produced the desired effect. The approach was evaluated with the participation of 18 practitioners. The main finding was that GETTY substantially modified the review process to a hypothesis-driven process that led to better review comments.

Another direction of research for improving code understanding for reviews uses visualization of information. For example, ViDI supports visual design inspection and code quality assessment [245]. The tool uses static code analysis reports to identify critical areas in code, displays the evolution of the amount of issues found in a review session, and allows the reviewer to inspect the impact of code changes. Git Thermite focuses on structural changes made to source code [214]. The tool analyzes and visualizes metadata gathered from GitHub, code metrics of the modified files, and static source code analysis of the changes in pull requests. DERT aims at complementing line-based code diff tools with a visual representation of structural changes, similarly to UML but in a dynamic manner, allowing the reviewer to see an overview as well as details of the code change [57]. Similarly, STRIFFS visualizes code changes in an UML class diagram, providing the reviewer an overview [103]. CHANGEVIZ allows developers to inspect method calls/declarations

related to the reviewed code without switching context, helping to understand a change and its impact [110]. OPERIAS focuses on the problem of understanding how particular changes in code relate to changes to test cases [186]. The tool visualizes source code differences and a change's coverage impact. Finally, a tool was developed to improve the review process of visual programming languages (such as Petri nets) [202]. It supports the code review of visual programming languages, similarly to what is already possible with textual programming languages.

Meyers et al. [175] developed a dataset and proposed a Natural Language–based approach to identify potential problems and elicit an active response from the colleague responsible for modifying the target code. They trained a classifier to identify acted-upon comments with good prediction performance (AUC = 0.85).

Monitoring review performance and quality. González-Barahona et al. [116] have proposed to quantitatively study the MCR process, based on traces left in software repositories. Without having access to any code review tool, they analyzed changelog files, commit records, and attachments and flags in Bugzilla records to monitor the size of the review process, the involved effort, and process delay. A similar study focused on review messages wherein changes are first reviewed through communication in a mailing list [141]. They developed a series of metrics to characterize code review activity, effort, and delays [142], which are also provided through a dashboard that shows the evolution of the review process over time [140]. Another study looked at code reviews managed with Gerrit and proposed metrics to measure velocity and quality of reviews [161]. Similar metrics, such as code churn, modified source code files, and program complexity, were used to analyze reviewer effort and contribution in the Android open source project [177]. Other tools to analyze Gerrit review data are ReDa, which provides reviewer, activity, and contributor statistics [243], and Bicho, which models code reviews as information from an issue tracking system, allowing us to query review statistics with standard SQL queries [115]. Finally, Codeflow Analytics aggregates and synthesizes code review metrics (over 200) [72].

Determining the usefulness of code reviews. A study of three projects developed a taxonomy of review comments [163, 164]. After training a classifier and categorizing 147K comments, they found that inexperienced contributors tend to produce code that passes tests while still containing issues, and external contributors break project conventions in their early contributions. In another study, Rahman et al. [205] analyzed the usefulness of 1,116 review comments (a manual process that has also been attempted to be automatized [192]) in a commercial system. They marked a comment as useful if it triggered a code change within its vicinity (up to 10 lines) and analyzed features of the review comment pertaining to its content and author. The results indicate that useful comments share more vocabulary with the changed code, contain relevant code elements, and are written by more experienced reviewers. Similarly, another study found that experienced reviewers are capable of pointing out broader issues than inexperienced ones [131]. The study concluded that reviewer experience and patch characteristics such as commits with large and widespread modifications drive the number of comments and words in a comment [131]. A study investigated the use of existing comments in code reviews [225]. The study concluded that when the existing code review comment is about a type of bug, participants are more likely to find another occurrence of this type of bug. However, existing comments can also lead to availability bias [225].

A study of 2,817 review comments found that only about 14% of comments are related to software design, of which 73% provided suggestions to address the concerns, indicating that they were useful [278]. Another study investigated the characteristics of useful code reviews by interviewing seven developers [78]. The study found that the most useful comments identify functional issues, scenarios where the reviewed code fails, and suggest API usage, design patterns, or coding conventions. "Useless" comments ask how an implementation works, praise code, or point to work needed

in the future. Armed with this knowledge, the researchers trained a classifier that achieved 86% precision and 93% recall in identifying useful comments. Applying the classifier on 1.5M review comments, they found that (a) reviewer experience with the code base correlates with usefulness of comments, suggesting that reviewer selection is crucial, (b) the smaller the changeset, the more useful the comments, and (c) a comment usefulness density metric can be used to pinpoint areas where code reviews are ineffective (e.g., configuration and build files). Criticism of the above pure statistical, syntactic approaches arose as the actual meaning of comments is not analyzed [100].

Managing code reviews. Before code hosting platforms, such as Github, became popular, researchers investigated how to provide support for reviews in IDEs. SeeCode integrates with Eclipse and provides a distributed review environment with review meetings and comments [220]. Similarly, ReviewClipse supports a continuous post-commit review process [70]. Scrub combines regular peer reviews with reports from static source code analyzers in a stand-alone application [133]. Java Sniper is a web-based, collaborative code reviewing tool [282]. All these early tools have been outlived by modern code hosting and reviewing infrastructure services such as GitHub, GitLab, BitBucket, Review Board, and Gerrit. However, while these platforms provide basic code reviewing functionalities, research has also looked at improving the reviewing process in different ways [62]. For example, Dürschmid [96] suggested continuous code reviews that allow anyone to comment code they are reading or reusing, e.g., from libraries. Developers can then push questions and comments to upstream authors from within their IDE without context switching. Fistbump is a collaborative review platform built on top of GitHub, providing an iteration-oriented review process that makes it easier to follow rationale and code changes during the review [147].

Fairbanks [104] has proposed the use of **Design by Contract (DBC)** and highlighted how it can improve how software development teams do code reviews. When both the code author and code reviewer agree to a goal of writing code with clear contracts, they can look out for the DBC practices being followed (or not) in the code being reviewed. The author lists a few DBC examples that can be used by software development teams.

Balachandran and Vipin [56] proposed changes in the developer code review workflow to leverage online clone detection to identify duplicate code during code review. They evaluated their approach through a developer survey and learned that the proposed workflow change will increase the usage of clone detection tools and can reduce accidental clones.

Hasan et al. [123] developed and evaluated an approach to measure the effectiveness of developers when doing code reviews. They defined a set of metrics and developed a model to measure code review usefulness. Their approach improved the state of the art by ~25%. They conducted a survey with participants from Samsung and learned that the respondents found their approach useful.

Optimizing the order of reviews. Code reviewers often need to prioritize which changes they should focus on reviewing first. Many studies propose to base the review decision on the likelihood that a particular change will eventually be accepted/merged [53, 105, 213]. Fan et al. [105] proposed an approach based on Random Forest. They evaluated their approach using data from three open source projects and learned that their approach is better than a random guess. In addition to the acceptance probability, Azeem et al. [53] also considered the probability that a code integrator will review/respond to a code review request. They rank the code review requests based on both the acceptance and response probabilities, which are calculated using machine learning models. They evaluated their approach using data from open source projects and obtained solid results. Saini and Britto [213] developed a Bayesian Network to predict acceptance probability. The acceptance probability is combined with other aspects, such as task type and the presence of merge conflicts, to order the list of code review requests associated with a developer. They evaluated their approach both using historical data and user feedback (both from Ericsson). They learned that their approach has good prediction performance and was deemed as useful by the users.

PRioritizer is a pull request prioritization approach that, similarly to a priority inbox, sorts requests that require immediate attention to the top [253]. Users of the system reported that they liked the prioritization but miss insights on the rationale for the particular pull request ordering.

Other studies looked especially at the historic proneness of files to defects to direct review efforts. A study suggests combining bug-proneness, estimated review cost, and the state of a file (newly developed or changed) to prioritize files to review [47]. The evaluation, performed on two open source projects, indicates that the approach leads to more effective reviews. A similar approach attempts to classify files as potentially defective, based on historic instances of detected faults and features on code ownership, change request, and source code metrics [170].

Some studies focus on predicting the time to review [281, 284]. Zhao et al. [284] focused on developing an approach that focus on time to review to prioritize review requests. In their approach, they employed a learning-to-rank approach to recommend review requests that can be reviewed quickly. They evaluated their approach through a survey with GitHub code reviewers. The survey participants acknowledge the usefulness of the approach. Zhang et al. [281] employed Hidden Markov Chains to predict the review time of code changes. They evaluated their approach using data from open source projects, with promising results.

Wang et al. [261] aimed at supporting review request prioritization by identifying duplicated requests. To do so, they consider a set of features, including the time when review requests are created. They developed a machine learning model to classify whether a code review request is duplicated. They validated their approach using data from open source, obtaining mixed results.

Automating code reviews. Gerede and Mazan [112] have proposed to train a classifier on whether a change request is likely to be accepted or not, knowing in advance the likelihood of a rejected change request would reduce the review effort, as those changes would not even reach the reviewing stage. They found that the change requests by inexperienced developers that involve many reviewers are the most likely to be rejected. In the same line of research, Li et al. [162] used Deep Learning to predict a change's acceptance probability. Their approach, called DeepReview, outperformed traditional single-instance approaches.

Review Bot uses multiple static code analysis tools to check for common defect patterns and coding standard violations to create automated code reviews [55]. An evaluation with seven developers found that they agreed to 93% of the automatically generated comments, likely due to the lack of consistent adoption of coding standards, which were the majority of the identified defects. Similarly, Singh et al. [221] studied the overlap of static analyzer findings with reviewer comments in 92 pull requests from GitHub. Of 274 comments, 43 overlapped with static analyzer warnings, indicating that 16% of the review workload could have been reduced with automated review feedback.

A series of studies investigated the effect of bots on code reviewing practice. Wessel et al. [266] conducted a survey to investigate how software maintainers see code review bots. They identified that the survey participants would like enhancements in the feedback bots provide to developers, along with more help from bots to reduce the maintenance burden for developers and enforce code coverage. A follow-up study [267], in which 21 practitioners were interviewed, identified distracting and overwhelming noise caused by review bots as a recurrent problem that affects human communication and workflow. However, a quantitative analysis [265] of 1,194 software projects from GitHub showed that review bots increase the number of monthly merged pull requests. It showed also that after the adoption of review bots, the time to review and reject pull requests decreased, while the time to accept pull requests was unaffected. Overall, bots seem to have a positive effect on code reviews, and countermeasures to reduce noise, as discussed by Wessel et al. [267], can even improve that effect.

CFar has been used in a production environment resulting in (a) enhanced team collaboration as analysis comments were discussed; (b) improved productivity as the tool freed developers from providing feedback about shallow bugs; and (c) improved code quality, since the flagged issues were acted upon, and (d) the automatic review comments were found useful by the 98 participating developers [127].

Recently, researchers have invested in using Deep Learning to aiming at code review automation [52, 162, 218]. Some studies have focused on identifying the difference between different code revisions [52, 218], while Tufano et al. [244] focused on providing an end-to-end solution, from identifying code changes to providing review comments. Finally, Hellendoorn et al. [126] evaluated if it is feasible at all to automate code reviews by developing a Deep Learning–based approach to identify the location of comments. They concluded that just this simple task is very challenging, indicating that a lot of research is still required before fully automated code review becomes a reality.

Analyzing sentiments, attitudes, and intentions in code reviews. Understanding review comments in greater detail could lead to systems that support reviewers in both formulating and interpreting the intentions of code reviews. A study on Android code reviews investigated the communicative goals of questions stated in reviews [98], identifying five different intentions: suggestions, requests for information, attitudes and emotion, description of a hypothetical scenario, and rhetorical questions. A study at Microsoft showed that the type of a change intent can be used to predict the effort for a code review [262]. A study on the Chromium project found that code reviews with lower inquisitiveness, higher sentiment (positive or negative), and lower syntactic complexity were more likely to miss vulnerabilities in the code [180].

Several studies investigated how sentiments are expressed in code reviews [42, 101, 134]. SentiCR flags comments as positive, neutral, or negative with 83% accuracy [42] and was later compared to classifiers developed for the software engineering context. Surprisingly, it was outperformed by Senti4SD [80]. The same investigation found that contributors often express their sentiment in code reviews and that negative and controversial reviews lead to a longer review completion time [102]. A study at Google investigated interpersonal conflict and found in a survey that 26% have at least once a month negative experiences with code reviews [101]. Furthermore, they found that rounds of a review, reviewing and shepherding time, have a high recall but low precision in predicting negative experiences. Other research has focused on nonverbal physiological signals, such as electrodermal activity, stress levels, and eye movement, to measure affect during code reviews. These signals were associated with increased typing duration and could be used in the future to convey emotional state to improve the communication in code reviews that are typically conducted without direct interaction [256]. A study categorized incivility in open source code review discussions [107]. The results indicate that more than half (66.66%) of the nontechnical emails included uncivil features. Frustration, name calling, and impatience are the most frequent features in uncivil emails. The study also concluded that sentiment analysis tools cannot reliably capture incivility. In a study of six open source projects, men expressed more sentiments (positive/negative) than females [196].

Code reviews on touch enabled devices. Müller et al. [179] have proposed to use multi-touch devices for collaborative code reviews in an attempt to make the review process more desirable. The approach provides visualizations, for example, to illustrate code smells and metrics. Other researchers have compared reviews performed on the desktop and on mobile devices [108]. In an experiment, they analyzed 2,500 comments, produced by computer science students, and found that (a) the reviewers on the mobile device found as many defects as the ones on the desktop and (b) seemed to pay more attention to details.

Other solutions. Some primary studies propose an initial proof of concept approaches for different purposes: to automatically classify commit messages as clean or buggy [160], to

eliminate stagnation of reviews and to optimize the average duration of open reviews [255], use of interactive surfaces for collaborative code reviews [200], and to link code reviews to code changes [188]. In addition, a study [167] compares two reviewer recommendations algorithms, concluding that recommendation models needs to be trained for a particular project to perform optimally.

Links to tools and databases available reported in the primary studies: We extracted the links to tools and databases reported in the primary studies providing solutions to support modern code reviews. Only a few primary studies provide links to the proposed tools or databases used in the studies. Most of the proposed solutions were for supporting reviewer recommendations. However, only 2 of 36 solutions provided links to the tools, and seven primary studies provided links to the database they used in their studies. We observed most reporting of links (17/28) to tools and databases for the primary studies providing support to understand changes that need to be reviewed. The complete list of the links to the tools and databases, along with the purpose of the links, is available in our online repository [6].

4.2.2 Human and Organizational Factors. This theme includes primary studies that investigate the reviewer and/or contributor as subject, for example, reviewer experience and social interactions. Studies that contribute to the human factors (e.g., experience) and organizational factors (e.g., social network) are categorized into this theme.

Review performance and reviewers' age and experience. The most investigated topic in this theme is the relation between the reviewers' age and experience on the review performance. Studies found that reviewer expertise is a good indicator of code quality [90, 154, 155, 174, 208, 240]. In addition, studies found that reviewers' experience [154] and developers' experience [66, 157] influence the code review outcome such as review time and patch acceptance or rejection. A study investigated human factors (review workload and social interactions) that relate to the participation decision of reviewers [210]. The results suggest that human factors play a relevant role in the reviewer participation decision. Another study investigated if age affects reviewing performance [181]. The study compared students in their 20s and 40s showed no difference based on age or development experience. Finally, there exists some early work on harvesting reviewer experience through crowdsourcing the creation of rules and suggestions [139].

Review performance and reviewers' reviewing patterns and focus. Eye tracking has been used in several studies to investigate how developers review code. Researchers found that a particular eye movement, the scan pattern, is correlated with defect detection speed [67, 216, 252]. The more time the developer spends on scanning, the more efficient the defect detection [216]. Based on these results, researchers have also stipulated that reviewing skill and defect detection capability can be deduced from eye movement [83]. Studies compared the review patterns of different types of programmers [124, 138]. A study compared novice and experienced programmers and based on the eye movements and reading strategies concluded that experienced programmers grasped and processed information faster and with less effort [124]. When comparing the eye-tracking results based on gender, a study found that men fixated more frequently, while women spent significantly more time analyzing pull request messages and author pictures [138].

Review performance and reviewers' workload. The impact of workload on code reviews has been investigated from two perspectives. First, a study found that workload (measured in pending review requests) negatively impacts review quality in terms of bug detection effectiveness [155]. Second, a study crossing several open source projects found that workload (measured in concurrent and remaining review tasks) negatively impacts the likelihood that the reviewers accepts a new review invitation [210].

Review performance and reviewers' social interactions. Code reviews have been studied with different theoretical lenses on social interactions. A study used social network analysis to

model reviewer relationships and found that the most active reviewers are at the center of peer review networks [272]. Another study used the snowdrift game to model the motivations of developers participating in code reviews [153]. They describe two motivations: (i) a reviewer has a motive of choosing a different action (review, not review) from the other reviewer, and (ii) a reviewer cooperates with other reviewers when the benefit of review is higher than the cost. A study found that past participation in reviews on a particular subsystem is a good predictor for accepting future review invitations [210]. Similarly, another study looking at review dynamics found the amount of feedback at patch has received is highly correlated with the likelihood that the patch is eventually voted to be accepted by the reviewer [237].

Review performance and reviewers' understanding of each other's comments. A study on code reviews investigated if reviewers' confusion can be detected by humans and if a classifier can be trained to detect reviewers' confusion in review comments [97]. The study concludes that while humans are quite capable of detecting confusion, automated detection is still challenging. Ebert et al. [99] identified causes of confusion in the code: the presence of long or complex code changes, poor organization of work, dependency between different code changes, lack of documentation, missing code change rationale, and lack of tests. The study also identified the impact of confusion and strategies to cope with confusion.

Review performance and reviewers' perception of code and review quality. A survey study conducted among reviewers identified factors that determine their perceived quality of code and code reviews [154]. High-quality reviews provide clear and thorough feedback, in a timely manner, by a peer with a supreme knowledge of the code base, strong personal and interpersonal qualities. Challenges to achieve high-quality reviews are of technical (e.g., familiarity with the code) and personal (e.g., time management) nature.

The difference between core and irregular contributors and reviewers. Studies investigated the difference between core and irregular contributors and reviewers in terms of review requests, frequency, and speed [65, 73, 75, 150, 199]. A study found that contributions from core developers were rejected faster (to speed-up development), while contributions from casual developers were accepted faster (to welcome external contributions) [65]. Similar observations were made in other studies [75, 150], while Bosu and Carver [73] found that top code contributors were also the top reviewers. A study explored different characteristics of the patches submitted to company-owned OSS project and found that volunteers face 26 times more rejections than employees [199]. In addition, the review of patches submitted by volunteers have to wait, on average, 11 days, whereas employees wait 2 days on average. Studies also investigated the acceptance likelihood of core and irregular contributors [75, 125]. Bosu and Carver [75] found that core contributors are more likely to have their changes accepted to the code base than irregular contributors. A potential explanation for this observation was found in another study [125], showing that rejected code is significantly different (due to different code styles) to the project code than accepted code. More experienced contributors submit code that is more compliant to the project's code style. A study investigated the consequences of disagreement between reviewers who review the same patch [130]. The study found that more experienced reviewers are more likely to have a higher level of agreement than less-experienced reviewers. A study investigating the career paths of contributors (from non-reviewer, i.e., developer, to reviewer, to core reviewer) found that (a) there is little movement between the population of developers and reviewers, (b) the turnover of core reviewers is high and occurs rapidly, (c) companies are interested in having core reviewers in their full-time staff, and (d) being a core reviewer seems to be helpful in achieving a full-time employment in a project [254].

The effect of the number of involved reviewers on code reviews. A study found that the more the developers are involved in the discussion of bugs and their resolution, the less likely the

reviewers are to miss potential problems in the code [155]. The same holds not true for reviewer comments: Surprisingly, the studied data indicate that the more reviewers participate with comments on reviews, the more likely they miss bugs in the code they review. Another study also made a counter-intuitive observation: Files vulnerable to security issues tended to be reviewed by more people [174]. One reported explanation is that reviewers get confused about what their role in the review is if there are many reviewers involved (diffusion of responsibility). Similar results were found in a study of a commercial application: The more reviewers are active, the less efficient the review and the lower the comment density [95]. In a study including both open source and commercial projects, it was observed that it is general practice to involve two reviewers in a review [209].

Information needs of reviewers in code reviews. A study identified the following information need categories: alternative solutions and improvements, correct understanding, rationale, code context, necessity, specialized expertise, and splitability of a change [195]. The authors of the study find that some of the information needs can be satisfied by current tools and research results, but some aspects seem not to be solved yet and need further investigation. Studies investigated the use of links in review comments [143, 259]. A case study of the OpenStack and Qt projects indicated that the links provided in code review discussion served as an important resource to fulfill various information needs such as providing context and elaborating patch information [259]. Jiang et al. [143] found that 5.25% of pull requests in 10 popular open source projects have links. The authors conclude that pull requests with links have more comments, more commenters and longer evaluation time. Similar results were found in a study of three open source projects [258] where patches with links took longer to review. The study also finds combining two features (i.e., textual content and file location) to be effective in detecting patch linkages. Similarly machine learning classifiers can be used to automate patch linkages [132].

4.2.3 Impact of Code Reviews on Product Quality and Human Aspects (IOF). This theme includes primary studies that investigate the impact of code reviewers on artifacts such as code, design, and human aspects, such as attitude and understanding.

The impact of code reviews on defect detection or repair. A study showed that unreviewed commits have over twice as many chances of introducing bugs than reviewed commits [7]. Similarly, observations from another study show that both defect-prone and defective files tend to be reviewed less rigorously in terms of review intensity, participation, and time than non-defective files [239].

Another study has investigated how code review coverage (the proportion of reviewed code of the total code), review participation (length and speed of discussions), and reviewer expertise affect post-release defects in large open source projects [172]. The findings suggest that reviewer participation is a strong indicator for defect detection ability. While high code review coverage is important, it is even more important to monitor the participation of reviewers when making release decisions and select reviewers with adequate expertise on the specific code. However, these findings could not be confirmed in a replication study [159]. They found that review measures are neither necessary nor sufficient to create a good defect prediction model. The same conclusions we confirmed in a project of proprietary software [219]. In their context, other metrics such as the proportion of in-house contributions, the measure of accumulated effort to improve code changes, and the rate of author self-verification contributed significantly to defect proneness [219].

Defective conditional statements are often the source of software errors. A study [248] found that negations in conditionals and implicit indexing in arrays are often replaced with function calls, suggesting that reviewers found that this change leads to more readable code.

The impact of code reviews on code quality. Studies were conducted to find the problems fixed by code reviews. A study concluded that 75% of the defects identified during code review

are evolvability type defects [201]. They also found that code review is useful in improving the internal software quality (through refactoring). Similarly, other studies [68, 171] found that 75% of changes are related to evolvability and only 25% of changes are related to functionality.

Studies investigated the impact of code reviews on refactoring. A study on 10 JAVA OSS projects found the most frequent changes in MCR commits are on code structure (e.g., re-factorings and re-organizations) and software documentation [194]. An investigation of 1,780 reviewed code changes from 6 systems pertaining to two large open source communities found that refactoring is often mixed with other changes such as adding a new feature [191]. In addition, developers had explicit intent of refactoring only in 31% of review that employed refactoring [191]. An empirical study on refactoring-inducing pull requests found that 58.3% presented at least one refactoring edit induced by code review [88]. In addition, Beller et al. [68] found that 78–90% of the triggers for code changes are review comments. The remaining 10–22% are "undocumented." Another study showed that reviewed commits have significantly higher readability and lower complexity. However, no conclusive evidence was reported on coupling [7].

A study of Openstack patches found higher code conformance of a patch after being reviewed than a patch that was first submitted [228]. An investigation on the impact of code review on coding convention violation found that convention violations disappear after code reviews. However, only a minority of the violations were removed, because they were flagged in a review comment [119]. The comparison of cost required to produce quality programs using code reviews and pair programming showed that code reviews costs 28% less compared to pair programming [233].

The impact of code reviews on detection or fixes of security issues. According to a study [77], code review leads to the identification and fixes of different vulnerability types. The experience of reviewers regarding vulnerability issues is an important factor in finding security-related problems, as a study indicates [174]. Another large study [236] also has similar findings. The results indicate that code review coverage reduces the number of security bugs in the investigated projects. A study looked into the language used in code reviews to find if the linguistics characters could explain developers missing a vulnerability [180]. The study found that code reviews with lower inquisitiveness (fewer questions per sentence), higher positive or negative sentiment, lower cognitive load, and higher assertions are more likely to miss a vulnerability. A study investigated the security issues identified through code reviews in an open source project [93]. They found that 1% of reviewers' comments are security issues. Language-specific issues (e.g., C++ issues and buffer overflows) and domain-specific ones (e.g., such as Cross-Site Scripting) are often missed security issues, and initial evidence indicates that reviews conducted by more than two reviewers are more successful at finding security issues. Another online study on freelance developers' code review process has similar findings indicating that developers did not focus on security in their code reviews [91]. However, the results showed that prompting for finding security issues in code reviews significantly affects developers' identification of security issues. A study found the relevant factors in successful identification of security issues in code reviews [197]. The results indicate that the probability of security issues identification decreases with the increase in review factors such as number of reviewer's prior reviews and number of review comments authored on a file during the current review cycle. In addition, the probability of security issues identification increases with review time, number of mutual reviews between the code author and a reviewer, and a reviewer's number of prior reviews of the file.

The impact of code reviews on software design. A study [178] found that high code review coverage can help to reduce the incidence of anti-patterns such as Blob, Data class, Data clumps, Feature envy and Code Duplication in software systems. In addition, the lack of participation (length and speed of discussions) during code reviews has a negative impact on the occurrence of certain code anti-patterns. Similarly, a study [184] specifically looked for the occurrences of review

comments related to five code smells (Data Clumps, Duplicate Code, Feature Envy, Large Class, and Long Parameter List) and found that the code review process did identify these code smells. An empirical study of code smells in code reviews in two most active OpenStack projects (Nova and Neutron) found that duplicated code, bad naming, and dead code are the most frequently identified smells in code reviews [120]. Another investigation of 18,400 reviews and 51,889 revisions found that 4,171 of the reviews led to architectural changes, 731 of which were significant changes [189].

The impact of code reviews on design degradation is investigated in two studies [246, 247]. A study on code reviews in OSS projects found that certain code review practices such as long discussions and reviewers' disagreements can lead to design degradation [247]. To prevent design degradation, it is important to detect design impactful changes in code reviews. A study found that technical features (code change, commit message, and file history dimensions) are more accurate than social ones in predicting (un)impactful changes [246].

The impact of code reviews on teams' understanding of the code under review. An interview study [227] found that code reviews help to improve the team's understanding of the code under review. In addition, code review may be a valuable addition to pair programming, particularly for newly established teams [227]. Similarly, a survey of developers and a study of email threads found that developers find code review dialogues useful for understanding design rationales [232]. Another survey of developers [76] found code reviews to help in knowledge dissemination. This was also found in a survey of reviewers that code review promotes collective code ownership [69]. However, Caulo et al. [82] were not able to capture the positive impact of code review in knowledge translation among developers. The authors contribute the negative results to fallacies in their experiment design and notable threats to validity.

The impact of code reviews on peer impression in terms of trust, reliability, perception of expertise, and friendship. A survey of open source contributors [74] found that there is a high level of trust, reliability, and friendship between open source software projects' peers who have participated in code review for some time. Peer code review helped most in building a perception of expertise between code review partners [74]. Similarly, another survey [76] found that the quality of the code submitted for review helps reviewers form impressions about their teammates, which can influence future collaborations.

The impact of code reviews on developers's attitude and motivation to contribute. An analysis of two years of code reviews showed that review feedback has an impact on contributors becoming long-term contributors [185]. Specific feedback such as "Incomplete fix" and "Suboptimal solution" might encourage contributors to continue to work in open source software projects [185]. Similarly, a very large study found that negative feedback has a significant impact on developers' attitude [215]. Developers might not contribute again after receiving negative feedback, and this impact increases with the size of the project [215].

4.2.4 Modern Code Review Process Properties (CRP). This theme includes primary studies investigating how and when reviews should be conducted and characteristics such as review benefits, motivations, challenges, and best practices.

When should code reviews be performed? Research shows that code reviews in large open source software projects are done in short intervals [208, 209]. In particular, large and formal organizations can benefit from creating overlap between developers' work, which produces invested reviewers, and from increasing review frequency [208].

What are the benefits of code reviews besides finding defects? A study on large open source software projects found that code reviews act as a group problem-solving activity. Code reviews support team discussions of defect solutions [208, 209]. The analysis of over 100,000 peer reviews found that reviews also enable developers and passive listeners to learn from the

discussion [208, 209]. A similar observation was made in a survey of 106 practitioners, where, besides knowledge sharing, the development of cognitive empathy was identified as a benefit of code reviews [89].

How are review requests distributed? Research found that reviews distributed via broadcast (e.g., mailing list) were twice as fast as unicast (e.g., Jira). However, reviews distributed via unicast were more effective in capturing defects [48]. In the same investigation, code reviewers reported that a unicast review allows them to comment on specific code, visualize changes, and have less traffic of patches circulating among reviewers. However, new developers learn the code structure faster with frequent circulation of patches among those who subscribe to broadcast reviews.

Efficiency and effectiveness of code reviews compared to team walkthroughs. Team walkthroughs are often used in safety-critical projects but come with additional overhead. In a study that developed an airport operational database, the MCR process was compared with a walk-through process [71]. The authors suggest to adopt MCR to ensure coverage while adapting the formality to the criticality of the item under review. **Over-the-shoulder (OTS)** reviews are synchronous code reviews where the author leads the analysis. A study compared an experiment OTS with **tool-assisted (TA)**, asynchronous, code reviews. It was found that OTS generates higher-quality comments about more important issues and better supports knowledge transfer, while TA generates more comments [146].

Mentioning peers in code review comments. A study explored the use of @-mentions, a social media tool, in pull requests [283]. The main findings were that @-mentions are used more frequently in complex pull requests and lead to shorter delays in handling pull requests. Another study investigated which socio-technical attributes of developers are able to predict @-mentions. It found that a developers visibility, expertise, and productivity are associated with @-mentions, while, contrary to intuition, responsiveness is not [149]. Generalizing the idea of @-mentions, other researchers investigated to what information objects to stakeholders refer to in pull request discussions. Building taxonomies of reference and expression types, they found that source code elements are most often referred to, even though the studied platform (GitHub) does not provide any support in creating such links (in contrast to references to people or issue reports) [84].

Test code reviews. Observations on code reviews found that the discussions on test code are related to testing practices, coverage, and assertions. However, test code is not discussed as much as production code [224]. When reviewing test code, developers face challenges such as lack of testing context, poor navigation support (between test and production code), unrealistic time constraints imposed by management, and poor knowledge of good reviewing and testing practices by novice developers [224]. Test-driven code review is the practice of reviewing test code before production code and studied in a controlled experiment and survey [226]. It was found that the practice does not change review comment quality nor the overall amount of identified issues. However, more test issues were identified on the expense of maintainability issues in production code. Furthermore, in a survey it was found that reviewing tests was perceived as having low importance and lacking tool support.

Decision-making in the code review process. The review process and the resulting artifacts are an important source of information for the integration decision of pull requests. In a qualitative study limited to two OSS projects, it was found that the common, most frequent reason for rejection is unnecessary functionality [117]. In a quantitative study of 4.8K GitHub repositories and 1M comments, it was found that there are proportionally more comments, participants and comment exchanges in rejected than in accepted pull requests [114]. Another aspect of decision-making in code reviews is multi-tasking. It was observed that reviewers participating simultaneously in several pull requests (which happens in 62% of the 1.8M studied pull requests) increase the resolution latency [135]. MCR processes often contain a voting mechanism that informs the

integrator about the community consensus about a patch. The analysis of a project showed that integrators use patch votes only as a reference and decide in 40% of the cases against the simple majority vote [129]. Still, patches that receive more negative than positive votes are likely to be rejected.

Comparison of pre-commit and post-commit reviews. In change-based code reviews, one has the choice to perform either pre-commit or post-commit reviews. Researchers have created and validated a simulation model, finding that there are no differences between the two approaches in most cases [59]. In some cases, post-commits were better regarding cycle time and quality. For pre-commit reviews, the review efficiency was better.

Strategies for merging pull requests. A survey of developers and analysis of data from a commercial project found that pull request size, the number of people involved in the discussion of a pull request, author experience, and their affiliation are significant predictors of review time and merge decisions [156]. It was found that developers determine the quality of a pull request by the quality of its description and complexity and the quality of the review process by the feedback quality, test quality, and the discussion among developers [156].

Motivations, challenges, and best practices of the code review process. Several studies have been conducted to investigate benefits and challenges of modern code reviews. An analysis found that improving code, finding defects, and sharing knowledge were the top three of nine identified benefits associated with code reviews [169]. Similar studies identified knowledge sharing [34, 89], history tracking, gatekeeping, and accident prevention as benefits of code reviews [34]. Challenges such as receiving timely feedback, review size, and managing time constraints were identified as the top 3 of 13 identified challenges [3, 169]. Challenges such as geographical and organizational distance, misuse of tone and power, unclear review objectives and context were also identified [34]. In the context of refactoring, a survey found that changes are often not well documented, making it difficult for reviewers to understand the intentions and implications of refactorings [45]. The best practices for code authors include writing small patches, describing and motivating changes, select appropriate reviewers, and being receptive toward reviewers' feedback [169]. The code reviewers should provide timely and constructive feedback through effective communication channels [169]. Code reviews are a well-established practice in open source development (FOSS). An interview study [44] set out to understand why code review works in FOSS communities and found that (1) negative feedback is embraced as a mean for a positive opportunity for improvement and should not be reduced nor eliminated, (2) the ethic of passion and care create motivation and resilience to rejection, and (3) both intrinsic (altruism and enjoyment) and extrinsic (reciprocity, reputation, employability, learning opportunities) motivation are important. Another study proposes a catalog of MCR anti-patterns that describe reviewing behaviour or process characteristics that are detrimental to the practice: confused reviewers, divergent reviewers, low review participation, shallow review, and toxic review [85]. Preliminary results from studying a small sample (100) of code reviews show that 67% contain at least one anti-pattern.

4.2.5 Impact of Software Development Processes, Patch Characteristics, and Tools on Modern Code Reviews (ION). This theme includes primary studies investigating the impact of processes (such as continuous integration), patch characteristics (such as change size, descriptions), and tools (e.g., statics analyzers) on modern code reviews.

The impact of static analyzers on the code review process. A study on six open source projects analyzed which defects are removed by code reviews and are also detected by static code analyzers [193]. In addition, a study [249] found that the issues raised by coding style checker can improve patch authors' coding style to avoid the same type of issues in subsequent patch submissions. However, the warnings from static analyzers could be irrelevant for a given project or development context. To address this issue, a study [250] proposed a coding convention checker

that detects project-specific patterns. While most of the produced warnings would not be flagged in a review, addressing defects regarding imports, regular expressions, and type resolutions before the patch submission would indeed reduce the reviewing effort. Through an experiment [128], it was found that the use of a symbolic execution debugger to identify defects during the code review process is effective and efficient compared to a plain code-based view. Another study [168] proposed a static analyzer for extracting first-order logic representations of API directives that reduces the code review time.

The impact of gamification elements on the code review process. Gamification mechanisms for peer code review are proposed in a study [251]. However, an experiment with gamification elements in the code review process found that there is no impact of gamification on the identification of defects [151].

The impact of continuous integration on the code review process. Experiments with 26,516 automated build entries reported that successfully passed builds are more likely to improve code review participation and frequent builds are likely to improve the overall quality of the code reviews [203]. Similar findings were confirmed in a study [277] that found that passed builds have a higher chance of being merged than failed ones. On the impact of CI on code reviews, a study [81] found that on average CI saves up to one review comment per pull request.

The impact of code change descriptions on the code review process. Interviews with industrial and OSS developers concluded that providing motivations for code changes along with a description of what is changed reduces the reviewer burden [206]. Similarly, an analysis of OSS projects found that a short patch description can lower the likelihood of attracting reviewers [241].

The impact of code size changes on the code review process. An investigation of a large commercial project with 269 repositories found that when patch size increases, the reviewers become less engaged and provide less feedback [172]. An interview study with industrial and OSS developers found that code changes that are properly sized are more reviewable [206]. The size of patches negatively affects the review response time, as observed in a study on code reviews [66], and reduces the number of review comments [165] and code review effectiveness, as shown in a study of an OSS project [64]. Similarly, an analysis of more than 100,000 peer reviews in open source projects recommends that changes to be reviewed should be small, independent, and complete [95].

The impact of commit history coherence on the code review process. An interview study on industrial and OSS project developers found that the commit messages that are self-explanatory and have meaningful messages are easier to review [206]. In addition, interviewees suggest that the ratio of commits in a change to the number of files changed should not be high [206].

The impact of review participation history on the code review process. An analysis of three OSS projects found that the likelihood of attracting reviewers is higher when past changes to the modified files are reviewed by at least two reviewers [241]. Prior patches that had few reviewers tend to be ignored [241]. Another study, looking at reviews from two OSS projects found that more active reviewers have faster response times [66].

The impact of fairness on the code review process. Fairness, in general, refers to the decision and allocation of resources in a way that is fair to the individuals and the group. A study [113] in an OSS project investigated different fairness aspects and recommends, besides the common aspects of fairness such as politeness and precise and constructive feedback, to (a) distribute reviews fairly and (b) establish a clear procedure for how reviews are performed. A study [109] investigated how contributions from different countries are treated. The study found that developers from countries with low human development face rejection the most. From the perspective of bias, a study [182] investigated the benefits of anonymous code reviews. The results indicate that while anonymity reduces bias, it is sometimes possible to identify the reviewer, and there are some

practical disadvantages such as not being able to discuss with the reviewer. The study recommends to have a possibility to reveal the reviewer when required. Another qualitative study [183] found that there may be perceptible race bias in the acceptance of pull requests. Similarly, a study investigated the impact of gender, human, and machine bias in response time and acceptance rate [138]. The results indicate that gender identity has significant effect on response time, and all participants spend less time evaluating the pull requests of women and are less likely to accept the pull requests of machines.

The impact of rebasing operations in the code review process. An in-depth large-scale empirical investigation of the code review data of 11 software systems, 28,808 code reviews, and 99,121 revisions found that rebasing operations are carried out in an average of 75.35% of code reviews, of which 34.21% operations tend to tamper with the reviewing process [190].

4.2.6 Other. In this theme, we have papers that investigate code review process on a generic level. Two papers classified the code review process in open source projects [50] and in proprietary projects [60]. A study identifies the factors influencing the code review process [61].

4.3 Mapping of Papers to Statements

As explained in Section 3.2, Q-Methodology relies on a set of statements that can be assessed by survey participants. We created these statements based on the primary studies' research objective. All primary studies with similar research objectives are mapped to one statement. Note that each primary study could be mapped to different statements when their research objectives are multifaceted. We did not want to make the survey too long. Thus, we merged the statements that are closely related. In total, we generated 47 statements representing the primary studies. We do not have statements for three papers in the "other" theme as they were not as well aligned to the main themes. Similarly, we do not have statements representing four short papers with very brief descriptions of solutions proposals; we aimed to include statements with at least two or more solutions per statement. We provide the statements derived from the primary studies in Table 7. Note that we extended our mapping study from the review period until 2018 to 2021. Since we created the statements on a high level, we could map most of the new studies to the existing statements representing primary studies until 2018. However, we identified one new statement that we could not map to any existing statement. The new statement is related to investigating the impact of concurrent code changes on modern code reviews (last statement in the ION theme in Table 7).

5 SURVEY RESULTS

In this section, we report on the results that answer RQ2–*How do practitioners perceive the importance of the identified MCR research themes?* based on 46[5] statements that we derived from the identified five main themes on MCR research.

5.1 Demographics

We received 28 responses in total. We excluded three respondents, as they did not have any code review experience or entered invalid responses. The remaining 25 respondents work in different roles in large multinational organizations; 56% of the participants are working in Swedish software organizations. The company name was not a mandatory field in the survey. However, approximately 70% of the respondents provided their company name. Most of the respondents are from the telecommunication domain. We also received responses from practitioners working in product-based companies and IT services and consulting companies. We received one response

[5]As the survey was conducted after our initial mapping study, we could not include the new statement representing one primary study from the updated mapping study.

Table 7. Statements Representing the Five Main Themes and Frequency of Primary Studies

Statement	Papers	Statement	Papers
Support Systems: *It is important to investigate support for...*		Human and Organizational Factors: *It is important to investigate...*	
selection of appropriate code reviewers.	36	reviewers' age and experience.	11
understanding changes that need review.	28	reviewers' reviewing patterns and focus.	6
automating code reviews.	11	reviewers' understanding of each other's comments.	2
analyzing the sentiment/attitude/intention.	11	reviewers information needs.	5
monitoring review performance and quality.	10	reviewers' social interactions.	4
determining the usefulness of code reviews.	9	effect of number of involved reviewers.	4
managing code reviews.	9	reviewers' workload.	2
optimizing code review order.	9	core vs. irregular - requests, frequency and speed.	5
code reviews on touch-enabled devices.	2	core vs. irregular - acceptance likelihood.	2
		reviewers' perception of code and review quality.	1
		core vs. irregular - agreement level.	1
		core vs. irregular - career paths.	1
TOTAL	125	TOTAL	45
Impact of code reviews on product quality and human aspects: *It is important to investigate the impact of code reviews on. . .*		Modern code review process properties: *It is important to investigate. . .*	
code quality.	10	motivations, challenges and best practices.	6
defect detection or repair.	6	decision making process	4
detection or fixes of security issues.	7	when to review	3
software design.	6	review benefits.	3
teams' understanding of the code under review.	5	mentioning peers in comments.	3
peer impression.	2	comparison to team walkthroughs.	2
developers' attitude and motivation to contribute.	2	it is important to investigate reviews of test code.	2
		how requests are distributed.	1
		pre and post-commit code reviews.	1
		strategies for merging pull requests.	1
TOTAL	39	TOTAL	27
Impact of software development processes, patch characteristics, and tools on modern code reviews: *It is important to investigate the impact of . . . on the code review process*			
static analyzers.	5		
code size changes.	6		
fairness on the code reviews process.	4		
continuous integration.	3		
gamification elements.	2		
code change descriptions.	2		
review participation history.	2		
commit history coherence.	1	GRAND TOTAL (some papers were classified in more than one statement, hence the grand total exceeds the number of reviewed primary studies)	263
concurrent code changes.	1		
TOTAL	27		

Note: All statements start with prefix text mentioned in *italics* under each theme.

from the insurance domain. The respondents in the *other* category are those who previously worked in software companies and worked, at the time of the survey, in academia. Each respondent provided a rating for each of the 46 statements (25×46) and six explanations for the three statements in most positive and most negative ratings, respectively (25×6), resulting in 1,300 data points.

The demographic information of the participants (their role, and experience in development, and code review) is provided in Table 8. The respondents have varying experiences from two to 30 years and work in 10 different roles such as developer, architect, and tester. Moreover, 60% of the participants have a Master's degree, 30% a Bachelor's degree, and 10% have a Ph.D. The respondents who provided their company names covered four different domains and seven different large companies. The details that could trace back to the respondents, such as their names and company names, are not provided to ensure the confidentiality of our respondents.

5.2 Agreement Level—Rating of Statements

The agreement levels of 25 participants is illustrated in Figure 6. The vertical axis shows the 46 statements, while the horizontal axis shows the percentage on the agreement levels. The statements are sorted from most to least agreement.

Table 8. Demographic Information of the Survey Participants

ID	Role	Experience (years)		ID	Role	Experience (years)	
		Development	Code review			Development	Code review
P1	Expert engineer	15	10	P2	Testing	6	5
P3	Software engineer	4	4	P4	Quality	3	3
P5	Developer	5	2	P6	Developer	8	6
P7	Developer	3	3	P8	Software engineer	12	12
P9	Developer	7	7	P10	Developer	6	3
P11	Manager	14	10	P12	Architect	6	4
P13	Quality	20	15	P14	Architect	25	10
P15	Testing	10	3	P16	Developer	14	14
P17	Developer	15	7	P18	Developer	5	3
P19	Developer	15	15	P20	Designer	16	12
P21	Software engineer	30	30	P22	Software engineer	5	5
P23	Automation Engineer	6	2	P24	Architect	8	8
P25	Developer	10	2				

Three of the top five statements are related to the impact *of* and *on* code reviews. In addition, the benefits of code/test reviews have high positive ratings; 92% of the respondents agreed on "It is important to investigate the impact of code reviews on code quality." Similarly, four of the last five statements are on investigating the difference between core and irregular reviewers. None of the participants agreed with the statement "It is important to investigate the difference between core vs. irregular reviewers in terms of the level of agreement between reviewers." The statement "It is important to investigate support for code reviews on touch-enabled devices" received the most negative ratings. However, we can see from Figure 6 that there is no consensus on most of the statements. In other words, most statements received both positive and negative as well as neutral responses. In some cases (35% of the statements), the difference in positive and negative ratings is not vast (less than 20%). For example, "It is important to investigate support for determining the usefulness of code reviews" has 40% negative and 36% positive responses, which is a difference of only 4%. A deeper look at the differences is needed.

We grouped the rating on the five themes to see how the ratings vary within each. As seen in Figure 7, the theme IOF (see Figure 7(a)) received the most positive response. However, within the theme, the impact of code reviews on product quality (i.e., code quality, security issues, software design, and defect detection or repair) received a more positive response compared to human aspects, particularly on developers' attitude and peer impression. This indicates that practitioners perceive that research on the impact of code reviews on the outcome (e.g.,quality) is more important than research on human aspects. This observation is further corroborated by the fact that the theme **human and organizational factors (HOF)** had the second least agreement, short of the theme **support systems for code reviews (SS)**.

Figure 7(b) depicts the ratings on the theme ION. More than 50% of the respondents perceive the investigation of code change description, continuous integration, code size changes, and static analyzers on code review process to be important. However, the impact of commit history coherence, fairness, review participation history, and gamification on code reviews are not considered as important. We observe as well in the ION theme that practitioners are more negative toward research on human aspects such as impact of fairness in code reviews, which received the highest negative rating in the ION theme.

The investigation in the theme CRP is considered important, especially research on the investigation of benefits, challenges, and best practices (see Figure 7(c)). However, some of the topics, such as the process for distributing review requests and merging pull requests, were not considered as important.

In the theme HOF, research on the effect of the number of involved reviewers, reviewers' information needs, reviewers perception of code and review quality, and reviewers understanding each

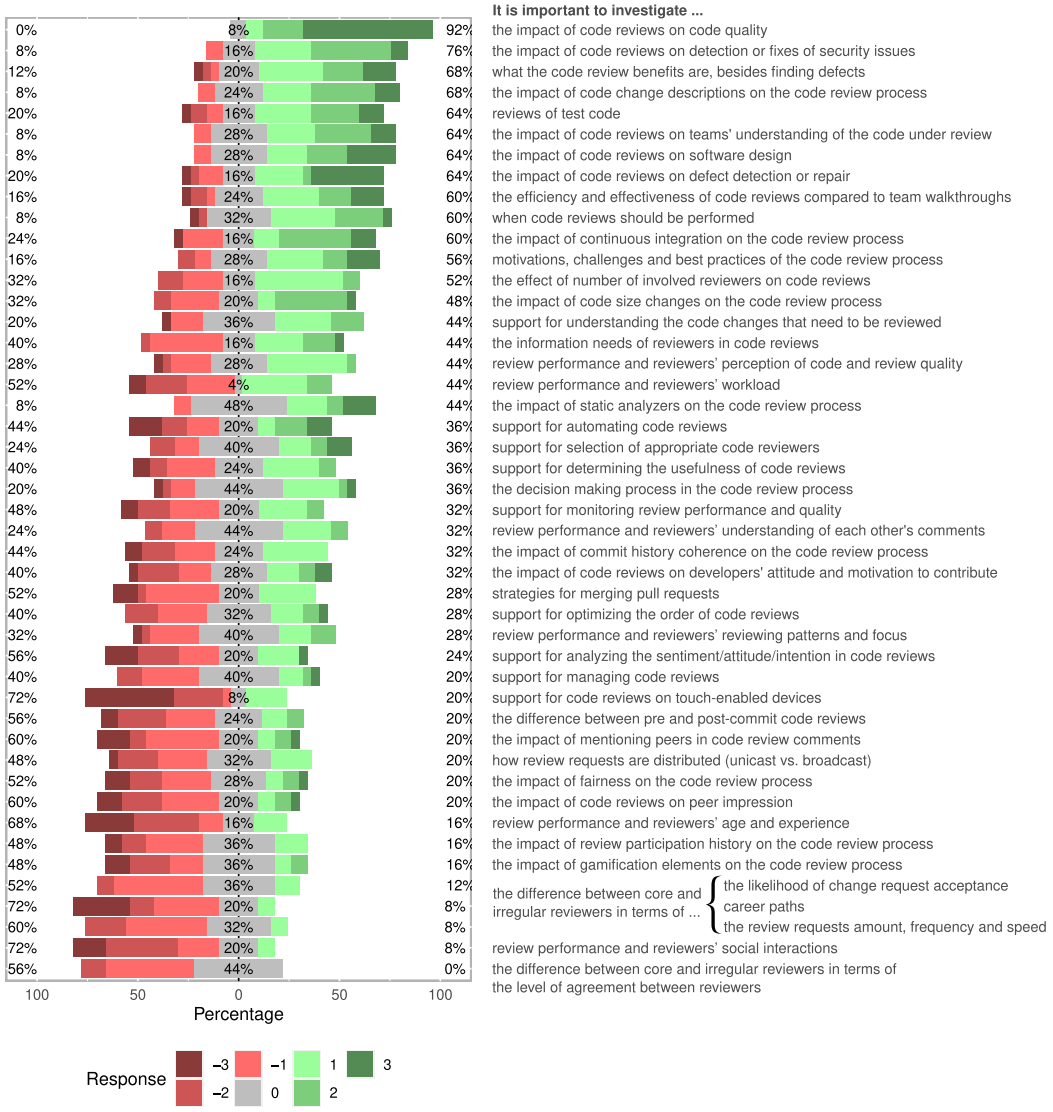
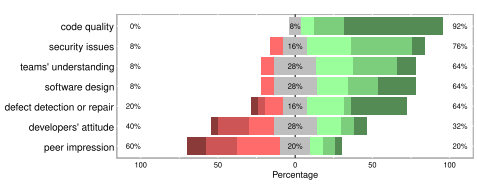


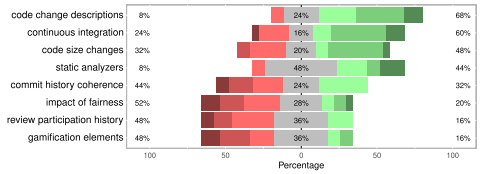
Fig. 6. Agreement level of the survey respondents. **Note**: All statements start with a prefix mentioned at the top of all statements.

others' comments was by the majority perceived as important. However, 72% of the respondents did not agree on the need to investigate reviewers' career paths and social interactions, as seen in Figure 7(d). In addition, 68% of the respondents did not perceive research on the reviewers' age and experience to be important.

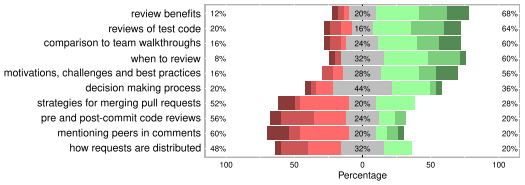
In the theme SS, only research on support for understanding what changes need review and selection of appropriate reviewers was perceived as important, as shown in Figure 7(e). Support for code reviews on touch-enabled device received most negative response where 72% of the respondents gave negative ratings. It is rather surprising that this theme received the least agreement overall, given that it is the theme with the majority of publications.



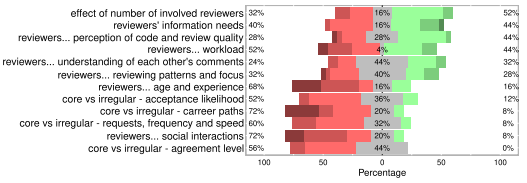
(a) Impact of code reviews on product quality and human aspects (IOF)



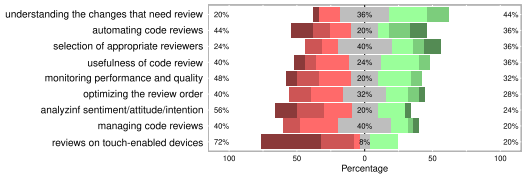
(b) Impact of software development processes, patch characteristics, and tools on modern code reviews (ION)



(c) Modern code review process properties (CRP)



(d) Human and organizational factors (HOF)



(e) Support systems for code reviews (SS)

Fig. 7. Ratings for the five themes.

When looking at the statements grouped in themes, there is a clear trend for the practitioners' preference on research that investigates causal relationships between code reviews and factors relevant for software engineering in general (themes ION and IOF). There is also a strong interest on modern code review process properties. Surprisingly, research on human and organizational factors as well as support systems for code reviews was not perceived as important by practitioners, which together represent nearly 70% (164 of 244) of the primary studies from our mapping study.

5.3 Factor Analysis

We further analyzed the survey data to identify patterns in the respondents' viewpoints using factor analysis, as suggested by the Q-Methodology. In the survey, we asked respondents to put only a fixed number of statements per rating, for example, only three statements in each of the −3 and 3 ratings. However, due to an error in the survey tool, four respondents could put more than the desired statements in some ratings. Therefore, we only included 21 of 25 valid participants responses in Q-method analysis. As mentioned in Section 3.2, the participants rate the statements, which is represented in a Q-Sort. For example, a Q-Sort of one participation for 46 statements is as follows (−3 3 2 3 2 −3 2 0 −3 2 0 0 −2 −2 0 2 −2 1 −1 3 0 1 −1 1 −1 1 0 0 −1 1 −1 −1 0 −1 1 −1 1 1 1 0 0 −2 0 −2 −1 0), where each value is the rating given by the participant for the statement. The Q-Sorts of all participants is used as input for factor analysis.

The steps followed in the Q-method analysis are (see the result of each intermediate step in the Q-method report available online [6]) as follows:

(1) Creating an initial matrix - An initial two-dimensional matrix is created (statements × participants), where the value of each cell is the rating given by the participants (between −3 to 3).

Table 9. Factor Characteristics from Q-methodology

Characteristics	F1	F2	F3
Average reliability coefficient	0.80	0.80	0.80
Number of loading Q-sorts	9.00	5.00	5.00
Eigenvalues	3.53	3.24	3.17
Percentage of explained variance	16.81	15.43	15.07
Composite reliability	0.97	0.95	0.95
Standard error of factor scores	0.16	0.22	0.22

(2) Creating correlation matrix - A correlation matrix between each Q-Sort (i.e., participant ratings) is generated using Pearson correlation coefficient test.
(3) Extracting factors and creating factor matrix - New Q-Sorts called factors are extracted that are the weighted average Q-Sorts of all participants with similar ratings. A factor represents Q-Sort of a hypothetical participant representing a similar viewpoint. We used **principal component analysis (PCA)** to extract the factors. The two-dimensional factor matrix is created (participants × factors). The value of each matrix cell is the correlation between the participants Q-Sort and the factors called factor loading. A higher loading value indicates more similarity between the participant and the factor.
(4) Calculating rotated factor loading - To clarify the relation among factors and increase explanatory capacity of the factors resulting from PCA, we conducted varimax factor rotation. Only a few factors are selected that represent the maximum variance. We used both a quantitative and qualitative approach to find select the number of factors. The quantitative criteria recommend in the literature are as follows [11]: A minimum of two loading Q-Sorts are highly correlated to the factor, (2) the composite reliability is greater or equal to 0.8 for each factor, (3) eigenvalues are above 1 for each factor, and (4) the sum of explained variance percentage of all the selected factors should be between 40% and 60%.
As shown in Table 9, when we select three factors all of the above criteria are satisfied. We also performed a qualitative analysis and excluded solutions with more than three factors, as there were few or no distinguishable statements. For example, a statement is distinguishable when its rank in one factor differs from all other factors.
(5) Finalising factor loading - The rotated factor loadings from the previous step are finalised by flagging the Q-Sorts that best represent the factors. We flagged the Q-Sorts based on the follow criteria: (1) Q-Sorts with factor loading higher than the threshold for p value < 0.05 and (2) Q-Sorts with square loading higher than the sum of square loadings of the same Q-Sort in all other factors. As seen in Table 9, the sum of number of loading Q-Sorts is 19, which means two respondents could not be significantly loaded into any of the factors.
(6) Calculating the z-scores and factor scores - The z-scores and factor scores indicate the statement's relative position within the factor. The z-score is a weighted average of the values of flagged participants' ratings given to a statement in the factor. Factor scores are based on ordering z-scores and mapping to the Q-Sort structure (−3 to 3); they are integer values instead of continuous. Factor scores are important for factor interpretation.
(7) Identifying distinguish statements - As mentioned in Step 4, a statement is distinguishable when its rank in one factor differs from all other factors. The factor scores from Step 5 are used to identify distinguished statements that represent the factor and used for factor interpretation. If there is a significant difference (more than 0.05) in factor score of a statement in one factor from all other factors, then that the statement is identified as distinguished statement. The distinguished statements in Factor 1 is provided in Table 10, Factor 2 in Table 11, and Factor 3 in Table 12.

Table 10. Distinguishing Statements in Factor 1

Statements (It is important to investigate …)	F1	F2	F3
IOF: …the impact of code reviews on teams' understanding of the code under review	3	1	1
IOF: …the impact of code reviews on developers' attitude and motivation to contribute	1	−1	−1
IOF: …the impact of code reviews on peer impression in terms of trust, reliability, perception of expertise, and friendship	0	−3	−2
HOF: …the difference between core and irregular reviewers in terms of career paths	−1	−3	−3
SS: …support for understanding the code changes that need to be reviewed	−1	1	0

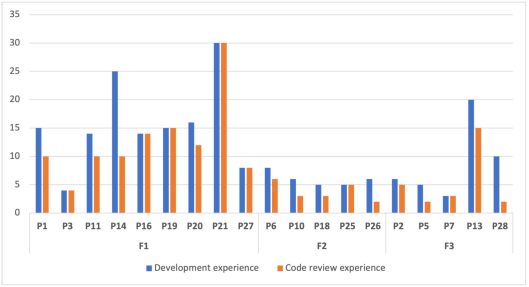


Fig. 8. Participants in each factor.

Figure 8 provides a summary of the respondents' experience in development and code reviewing in each factor. It is clear that respondents with more experience are grouped in Factor 1 compared to other factors. We provide an interpretation of the factors in the next subsections.

5.3.1 Factor 1 Interpretation—It is Important to Investigate the Code Reviewer as a Subject. Table 10 shows the distinguishing statements on Factor 1, which represents 43% of the respondents and explains 16.81% of the variance in responses. As seen in Figure 8, participants loaded in Factor 1 have more experience. They have expert/senior roles in architecture and design, an average of 16 years experience in software development, and 13 years of code review experience. Participants loaded in Factor 1 are more positive regarding the impact of code review on human factors than the ones loaded in Factors 2 and 3. For example, statements regarding the teams' understanding of the code under review, developers' attitude, and peer impression are perceived to be important in Factor 1. Regarding the impact of code reviews on teams' understanding, one of the respondents in Factor 1 wrote, P27: "Without understanding the requirement of the code, there is no point to review the code." Another respondent was interested in the investigation of knowledge sharing; he wrote, P3: "Code reviews enable knowledge sharing." Research on the impact of code review on developers' attitude is considered important, as a considerable amount of effort goes into reviewing code. A participant wrote, P14: "Everyone needs to see the importance of better quality." However, respondents in Factors 2 and 3 disagree. One of the respondents in Factor 2 wrote, P26: "this is more of an individual's approach towards any work. Once a reviewer is made to follow the correct set of principles, this [investigation on developers' attitude] can be eliminated."

All respondents display a neutral or even negative attitude toward the importance of investigating the impact of peer impression on code reviews. They feel that people should be objective and not be influenced by peer impressions. One of the respondents in Factor 1 wrote, "P1: It should not be necessary to do research on the obvious fact that people should be responsible." Similarly, respondents in Factor 1 are less negative compared to F2 and F3 about the importance to investigate the difference between core and irregular reviewers in terms of their career paths. However, respondents in Factor 1 are more negative compared to F2 and F3 about the importance to investigate support for understanding the code changes that need review. One of the respondents wrote, "P14:

Table 11. Distinguishing Statements in Factor 2

Statements (It is important to investigate …)	F1	F2	F3
IOF: …the impact of code reviews on defect detection or repair	1	3	0
ION: …the impact of continuous integration on the code review process	0	3	0
CRP: …the impact of mentioning peers in code review comments	−2	0	−1
HOF: …review performance and reviewers' age and experience	−2	−3	−1
HOF: …review performance and reviewers' understanding of each other's comments	0	−1	0
HOF: …review performance and reviewers' reviewing patterns and focus	0	−1	1
HOF: …review performance and reviewers' perception of code and review quality	1	−1	1
HOF: …the difference between core and irregular reviewers in terms of the likelihood of change request acceptance	0	−2	0
HOF: …the difference between core and irregular reviewers in terms of the level of agreement between reviewers	0	−2	0

Table 12. Distinguishing Statements in Factor 3

Statement (It is important to investigate …)	F1	F2	F3
IOF: …the impact of code reviews on software design	3	2	0
ION: …the impact of code change descriptions on the code review process	2	2	0
ION: …the impact of code size changes on the code review process	1	1	−1
CRP: …when code reviews should be performed	2	2	1
SS: …support for analyzing the sentiment/attitude/intention in code reviews	0	−1	−3
SS: …support for optimizing the order of code reviews	−1	−1	1

Everything should be reviewed, this is a non-question." However, the practitioner interpreted the question as "understanding what should be reviewed" rather than understanding the code under review. Despite the potential misinterpretation, this statement has been ranked as most important statement in the solutions theme (see Figure 7(e)).

5.3.2 Factor 2 Interpretation—It is not Important to Investigate Human Aspects Related to Code Review. Table 11 shows the distinguishing statements on Factor 2, which represents 24% of the participants and explains 15.43% of the variance in responses. The respondents grouped in this factor have less experience compared to the respondents in Factor 1 and 3 (see Figure 8). They have roles in development and testing with an average of 6 years experience in software development and 4 years of code review experience. Respondents in this factor are more positive about research on the impact of code review on defect detection or repair and impact of continuous integration than research on human factors. On the importance of defect detection or repair, one of the respondents wrote, P18: "This [defect detection] is generally why code reviews take place—it is interesting to perform a more formal causal analysis on this [the impact of code reviews on defect detection]."

Respondents do not see the importance of investigating human aspects unlike in Factor 1, where respondents with more experience are positive toward investigations on human factors. In this factor, more importance is given to having good code review guidelines as stated by one of the participants, P25: "Standard review procedure should be independent of individual/team members' age and experience."

5.3.3 Factor 3 Interpretation—It is More Important to Investigate the Support for Optimizing Code Reviews than Support for Analyzing Human Aspects. Table 12 shows the distinguishing statements on Factor 3, which represents 24% of the participants and explains 15.07% of the variance in responses. Respondents in this factor have mainly testing roles and an average of 9 years experience in software development and 5 years of code review experience.

Overall respondents in Factor 3 are less positive about research on the impact on and of code reviews and code review process. They are more interested in research on the support for optimizing code reviews than analyzing human aspects.

We did not get any explanations for the ratings as most of the ratings are between −2 to 2. For the −3 rating the respondent had no comments.

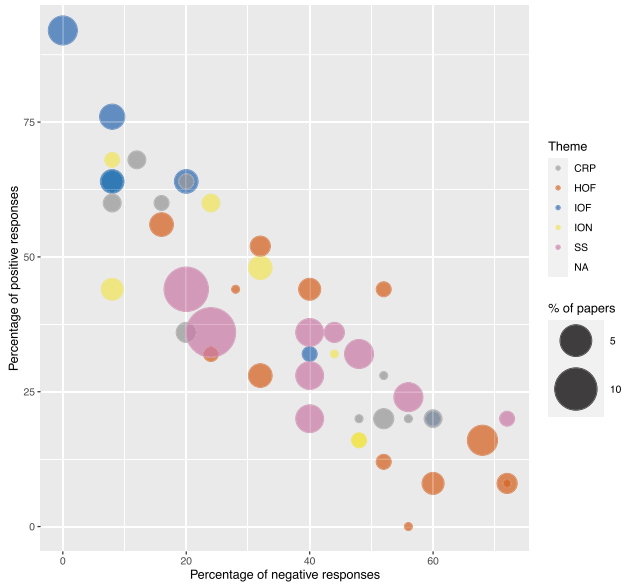


Fig. 9. Comparison of literature and practitioners' perceptions.

6 COMPARING THE STATE OF THE ART AND THE PRACTITIONERS' PERCEPTIONS

In this section, we answer RQ3–*To what degree are researchers and practitioners aligned on the goals of MCR research?* by juxtaposing the results from the mapping study (Section 4) and the responses from the survey (Section 5).

6.1 Comparing the Number of Research Articles and the Practitioners' Perceptions

In Figure 9, we map survey responses, the percentage of papers representing a survey statement, and modern code review themes. The percentage of negative and positive responses for each statement is shown on the x- and y-axes, respectively. Each bubble represents a statement from the survey and its size indicates the percentage of representing papers. The different colors represent the five themes we identified in the mapping study.

In addition, we evaluated if there is a statistical correlation between the number of papers and practitioners' perceptions. Using Shapiro–Wilk normality test we determined that our data are normally distributed. We then conducted a Pearson correlation test to evaluate if there is a significant relation between the ratings and the number of papers in different themes. The result of the correlation test is provided in Table 13, and the statistically significant results are in bold.

Figure 9 accentuates a result we reported on the agreement levels in Section 5.2: While there is considerable research on SS, and HOF, as indicated by the number and size of bubbles, practitioners seem to have a rather negative attitude toward the research done in this theme. None of the solution statements received more than 50% positive responses. Within this theme, research on support for understanding the code changes that need to be reviewed and support for the selection of appropriate reviewers received the most positive responses and was also associated with the most papers. This is a good example of alignment between research and practitioners' interest. The positive alignment is also confirmed in the correlation test as the solutions that have fewer publications received also more negative ratings (cf. Table 13). On the topic of reviewer selection, one of the respondents noted that "P9: The most effective review is the one done by developers who are the most familiar with a particular functionality or have worked on a similar functionality

Table 13. Correlation between the Number of Papers Published within a Theme and Practitioners' Perceptions

Pearson correlation	IOF		ION		CRP		HOF		SS	
	p value	*r*	*p* value	*r*	*p* value	*r*	*p* value	*r*	*p* value	*r*
Positive rating - Percentage of papers	**0.06555**	**0.9693013**	0.4908	0.2869428	0.2859	0.400119	0.3291	−0.2942445	0.09438	0.5901029
Negative rating - Percentage of papers	**0.01202**	**−0.8646518**	0.3612	−0.3741424	0.125	−0.55	0.2675	0.3321971	**0.009847**	**−0.7986266**

on a different project. I think there is no helping tool to tell who is the most appropriate reviewer." Several studies propose or evaluate tools that do just exactly that. While the respondents' answer is certainly not representative, more focus on knowledge translation and transfer to practitioners about existing solutions can be a beneficial target for researchers in this area. Furthermore, as seen in Section 4.2.1, only 2 of 36 solutions supporting the reviewer recommendation provide links to the tools, which could explain why practitioners are unaware of the existing solutions.

Looking at Figure 9, we see more negative than positive responses for statements related to HOF. However, we did not find any statistical significant relation between the number of papers in the HOF theme and the rating, as indicated in Table 13. The most positively received statement is related to investigating the effect of the number of involved reviewers in code reviews. The statement investigating review performance and reviewers' age and experience in this theme is associated with the most studies, but it is also perceived mostly negatively. For example, a respondent wrote, P2: "Age and experience is less important than code knowledge or ability to read code. An 18 year old with no experience writes the best comments, then that is the person I will invite to review." Another participant elaborated more on the age factor, P7: "I don't understand how the age of reviewer can help in performance, Experience to certain extent but that doesn't mean the experienced person knows new technologies that are emerging so this statement should be viewed as 2 separate things with respect to experience yes important to investigate to certain extent. But with respect to age some younger ones are actually doing more reviews now a days." Another respondent emphasised the importance on a standard review process being more important than reviewer age and experience, P25: "Standard review procedure is to be independent of individual/team members' age and experience."

Looking at the top-left corner of Figure 9, the area with high positive and low negative ratings is dominated by statements related to research on the IOF and CRP. Although, we can see that only the relation between the ratings and papers in the IOF theme is statistically significant (cf. Table 13). This result indicates that practitioners are interested in research that investigates causal relationships, as indicated by a respondent, P11: "Understanding how people approach and make decisions when performing a code review may open up some other interesting questions in how to structure and format code reviews to be more effective." However, there is only a relatively low number of studies in this area.

6.2 Comparing Research Impact and Practitioners' Perceptions

We retrieved citations of all primary studies as of August 2022. Peer citation is one way of assessing the research impact and the activity of a theme. We compared the research impact with the practitioners responses from the survey. As we have the practitioners responses on each statement, we calculated the research impact for each statement by considering the sum of citations of all primary studies representing a statement (see Table 7).

We grouped the analysis by creating bins for the publication year, since more recent publications have likely less citations than older publications, which may have had simply more time for being cited. The primary studies are published between 2007 and 2021 (Figure 10). The percentage of

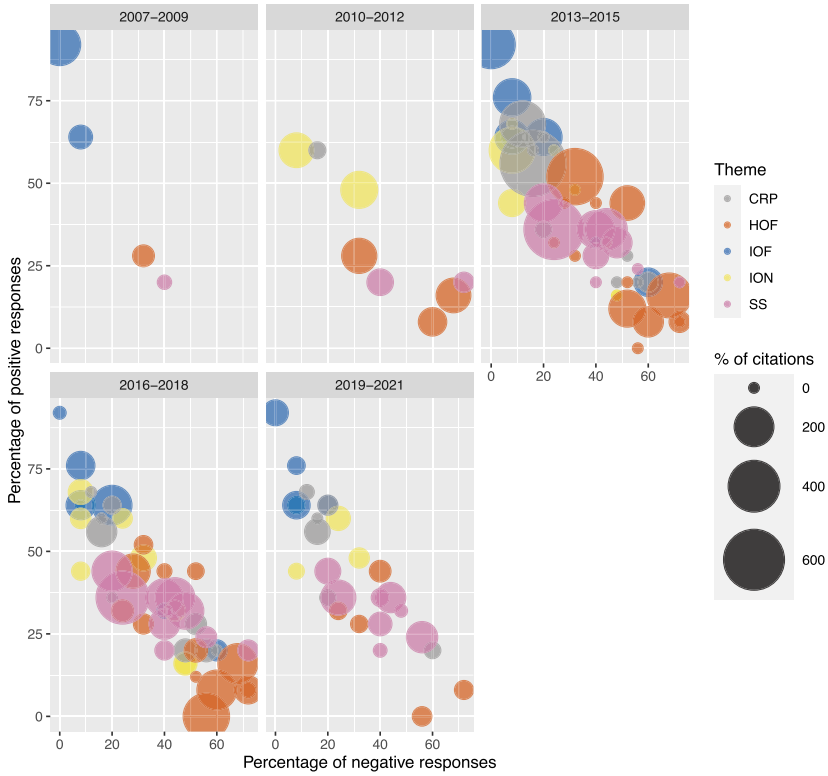


Fig. 10. Comparison of research impact and practitioners' perceptions.

Table 14. Correlation between the Ratings and Research Impact

Pearson correlation	2007–2009		2010–2012		2013–2015		2016–2018		2019–2021	
	p value	*r*	*p* value	*r*	*p* value	*r*	*p* value	*r*	*p* value	*r*
Positive rating - Citations	0.1186	0.8814008	0.8205	0.09631116	**0.01884**	**0.3450907**	0.2793	−0.1629392	0.9237	0.01934015
Negative rating - Citations	0.2149	−0.7850902	0.6684	−0.1807715	**0.0323**	**−0.3161727**	0.6639	0.06581944	0.9631	0.009334502

Table 15. Correlation between the Ratings and Research Impact on Each Theme

Pearson correlation	IOF		ION		CRP		HOF		SS	
	p value	*r*	*p* value	*r*	*p* value	*r*	*p* value	*r*	*p* value	*r*
Positive rating - Citations	0.06555	0.4091079	0.223	0.2776709	0.3124	0.2105235	0.9791	0.004592727	**0.001982**	**0.5502959**
Negative rating - Citations	0.2306	−0.2733161	0.1797	−0.3044363	0.2766	−0.2263241	0.5221	0.1119356	**0.004827**	**−0.5087684**

negative and positive responses for each statement is shown on the x- and y-axes, and the colors represent the different themes. Each bubble represents a statement from the survey and its size indicates the total number of citations of all primary studies in each statement.

In addition, we evaluated if there is a statistical correlation between the research impact and practitioners' perceptions. Using Shapiro–Wilk normality test we determined that our data are normally distributed. Then we conducted a Pearson correlation test to evaluate if there is a significant relation between the ratings and the research impact in different years. Table 14 shows the results of Pearson's correlation test for the different years. We also evaluated the correlation between the ratings and the research impact of papers in each theme (see Table 15).

Although the overall positive ratings are low for the SS theme, the papers with high impact have higher positive rating compared to low-impact papers. When considering all years together,

the SS theme exhibits a significant negative correlation between negative ratings and research impact ($r = -0.5087684$, $p = 0.004827$), indicating that when impact is high, the negative ratings are low. Similarly, the correlation between positive ratings and research impact is significant as well ($r = 0.5502959$, $p = 0.001982$). In the HOF theme we can see from Figure 10 that some of the statements that have high impact were perceived negatively by practitioners, particularly in the time frame between 2016 and 2018. However, we did not find any statistical significant relation between the ratings and statements in the HOF theme. In IOF, we can see that statements that have high impact also received more positive ratings. We also observed a statistically significant correlation between the positive ratings and impact in the time frame between 2013 and 2015 ($r = 0.7670108$, $p = 0.04419$). We did not find any interesting patterns in the other themes.

7 DISCUSSION

In this section, we summarize the research directions that emerged from analyzing the state of the art and the practitioner survey. Furthermore, we illustrate that our findings align with the observations made by Davila and Nunes [18], strengthening our common conclusions, since our respective reviews cover a non-overlapping set of primary studies. Finally, we discuss the threats to validity associated with our research.

7.1 MCR Research Directions

We propose future MCR research directions based on current trends and our reflections, both obtained through our mapping study and survey with practitioners. We anchor this discussion in the MCR process steps shown in Figure 1.[6] Next, we propose relevant research topics, along with research questions that still remain to be answered.

7.1.1 Preparation for Code Review. Understanding code to be reviewed - Understanding the code was perceived as important by the survey respondents (cf. Figure 7(e)). The solutions reported in the primary studies focus on a subset of patch characteristics that affect review readiness (such as References [94, 118, 260]). However, is it possible to combine all patch characteristics into an overall score that can inform the submitter so they can improve the patch before sending it out for review?

Review goal - After understanding the code to be reviewed, the next step is to decide the review goal. The survey respondents are positive about investigating the impact of code review on code quality in general and, more specifically, security (cf. Figure 7(a)). Our primary studies findings indicate that most issues found in code reviews are related to a subset of code quality attributes such as evolvability [68, 171]. Does that imply that only certain quality attributes can or should be evaluated with code reviews?

Review scope - Another aspect of preparing for code review is to decide which artifacts to review, i.e., the review scope. Test code is seldom reviewed and is not considered worthy of review [224, 226]. However, the survey respondents consider test code review one of the most important research topics (cf. Figure 7(c)). In addition to test and production code, the popularity of **third-party libraries (3pps)** in software development is increasing, which leads to an important area of 3pp review. How to use risk-based assessment to scope the review target and the review goals to achieve an acceptable tradeoff between effort and benefits?

Optimizing review order - Factors influencing the review order in OSS projects can differ from proprietary projects or may have different importance. For example, as mentioned in our primary

[6]Note that we did not identify any research path for *Step 4 - Reviewer Notification*, indicating that this activity is already well understood.

studies [53, 105, 213], acceptance probability is one of the determinants in ordering OSS reviews (important to attract contributors). However, in proprietary projects, other determinants such as merge conflict probability may have more importance in determining the review order.

Key Future Questions – Preparation

Understanding code to be reviewed

Q1 What are the overall factors that affect review readiness?

Q2 How can code (patch) preparation be automated?

Review goal

Q3 Which code quality attributes are better addressed in code reviews than other means (e.g., testing)?

Review scope

Q4 What artifacts other than code should be reviewed, and how much importance should be given to these reviews?

Q5 How can risk assessment be used to determine when to review the security of 3pps?

Optimizing review order

Q6 How do factors determining review order relate to project type (OSS/proprietary)?

7.1.2 Reviewer Selection. Appropriate reviewer selection - The primary studies focus on identifying "good reviewers" based on certain predictors such as pull request content similarity [145, 211, 268, 275]. However, how much do "good reviewers" differ in review performance from "bad reviewers"?

Number of reviewers - The primary studies establish a correlation between the number of reviewers and the review performance [95, 155, 174]. However, what are the factors determining the optimal number of reviewers?

Key Future Questions – Reviewer selection

Appropriate reviewer selection

Q1 What is the impact of selecting non-recommended reviewers?

Q2 Does it matter to choose the highest ranked reviewer or follow the recommended review order?

Number of reviewers

Q3 What are the factors that determine the optimal number of reviewers for a given project, and what should be their responsibilities (security, test code review, or requirements review) when multiple reviewers are involved?

7.1.3 Code Checking. After the preparation and reviewer selection step, the reviewers are notified and the actual review takes place. It is valuable to monitor the review process to learn new insights that can be codified in guidelines. It is known that code review can identify design issues [120, 178, 184, 189]. How can this identification be used as an input to create or update design guidelines? In addition, the primary studies found that good reviewers exhibit different code scanning patterns to less-good reviewers [67, 83, 216, 252]. Such findings should be used to propose/develop solutions that harvest this expertise from reviewers.

Key Future Questions – Code checking

Design support

Q1 How can design rules/guidelines be extracted and updated based on the design issues identified and corrected through code reviews?

Scanning patterns

Q2 How can reviewers' focus patterns, extracted by eye tracking from code reviewers, be used to model and eventually transfer reviewing expertise?

7.1.4 Reviewer Interaction. Review comment usefulness - Studies have investigated reviewer interaction through review comments. According to the primary studies, the usefulness of comments is determined by the changes caused by them [192, 205]. For example, a useless comment can be an inquiry about the code that leads to no change. However, such discussions can lead to knowledge sharing. Therefore, the semantics of comments should also be considered when determining their usefulness.

Knowledge sharing - Experienced reviewers possess tacit knowledge that is difficult to formalize and convey to novice programmers. Systems that could mine this knowledge from reviews would be an interesting avenue for research. It could be good if reviewers get a good mix of familiar and unknown code reviews to expand their expertise over time. Existing studies [90, 154, 155, 174, 208, 240] show that the expertise of reviewers is codified in reviews, but making that expertise tangible and accessible is still an open question. In addition, a review comment can sometimes contain links that provide additional information that could contribute to knowledge sharing.

Human aspects - When discussing reviewer interactions, human aspects received much attention in the reviewed primary studies. However, the investigation of review dynamics, social interactions, and review performance is focused on OSS projects. It is not known if such interactions differ in proprietary projects.

Key Future Questions – Reviewer interaction

Review comment usefulness

Q1 How could semantics to identify the usefulness of code review comments?

Knowledge sharing

Q2 How could the expertise of reviewers be mined through code review comments and made accessible to less experienced reviewers?

Q3 How can links between actors and information objects created in MCR be used to understand the review process better or extract useful information about the developed product?

Human aspects

Q4 To what extent is the impact of social interactions on review performance specific to project type (OSS/proprietary)?

7.1.5 Review Decision. We have identified studies that automate code reviews. When static code analysis tools are used for automation, the rules are visible and explicit (white box). In contrast, the decision rules are not easily interpretable when using deep learning approaches (black box).

Key Future Questions – Reviewer decision

Automation

Q1 How can code review automation, based on deep learning approaches integrate explainability and transparency of accept/reject decisions?

7.1.6 Overall Code Review Process. The future research directions and research questions that could not be directly mapped to any specific step in the code review process are categorized as overall topics.

Human aspects - The difference between core and irregular reviewers has been studied mostly in the context of OSS [65, 73, 75, 150, 199]. In our survey, the respondents perceived the difference between core and irregular reviewers as unimportant. However, the survey respondents were mostly from companies working with proprietary projects. It would be interesting to investigate if human factors matter more in OSS than in proprietary software development. The survey respondents considered the research on human aspects not as relevant as the research on other themes. We pose that, since most of the research on organizations and human factors is performed in OSS projects, research on human aspects in companies mainly working with proprietary projects is still very relevant as the human factors might differ between OSS and proprietary contexts.

Innersourcing - We have seen MCR research in OSS and in commercial projects. However, we do know very little about MCR in Innersourcing. Similarly to the open source way of working, Innersourcing promotes contributions across different projects within an organization. A delay or incomplete review may discourage an Innersource contributor. Human factors may be more relevant for commercial projects in the context of Innersourcing.

Code reuse - Common assets are code that is reused within an organization. It would be interesting to investigate how changes made to common assets should be reviewed. For example, should reviewers from teams that use the common assets be invited to review new changes?

Key Future Questions – Overall

Human aspects

Q1 Are the OSS studies' findings not applicable to proprietary software development?

Innersourcing

Q2 How does code review performance (review time, thoroughness, etc.) impact innersource contributions?

Common assets review

Q3 How should changes in common assets be reviewed?

7.2 A Comparison with Other Reviews and Surveys

Figure 11 illustrates the overlap and differences between the primary studies identified in this article and the works by Wang et al. and Davila and Nunes. Interestingly, even though the search period and aims of the three studies are comparable in scope, the included primary studies are quite diverse. Wohlin et al. [40] have made similar observations when comparing the results of two comparable mapping studies. While a detailed analysis, as done by Wohlin et al. is out of the scope of this article, we speculate that the main reason for the divergence of primary studies is the emphasis on different keywords in the respective searches. While Wang et al. and Davila and Nunes included "inspection" in their search string, we explicitly did not. The term is associated with traditional code inspections, which is a different process than MCR, as explained in Section 2.2.

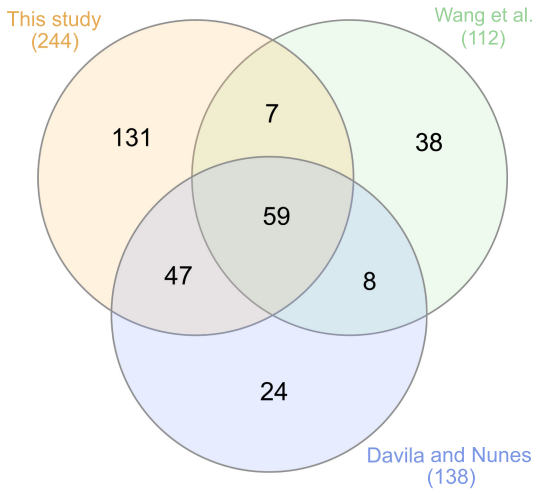


Fig. 11. Unique and common primary studies in MCR reviews with wide scope.

Rather, we included terms that are associated with MCR, like "pull request" and "patch acceptance." Wang et al. excluded also papers that were not published in a high-impact venue, likely leading to the lower number of included primary studies. We also explicitly did not exclude any papers based on their quality. Instead, we provide an assessment of the studies' rigor and relevance [23].

As described in Section 4.1, there is a lack of involvement of human subjects in code review research. As code review is a human-centric process, researchers should involve more human subjects to evaluate the feasibility of code review interventions. This research gap has also been observed by Davila and Nunes' review [18], who called for more user studies on MCR. Moreover, researchers should consider more data sources in addition to analyzing repositories to achieve triangulation, strengthening the conclusions that can be drawn on the developed interventions. This is analogous to the observation by Davila and Nunes [18], who found that a majority of MCR research focuses on particular open source projects (Qt, Openstack, and Android) or is conducted in large companies such as Microsoft and Google. MCR practices and motivations in companies working primarily with proprietary projects may be different than in open source projects. In companies, the contributions and reviews are not voluntary work; furthermore, it is more likely that the reviewers and contributors known each other. Moreover, there may be domain or organization-specific requirements that may not be present in open source projects. Therefore, there is a need to investigate more MCR in the context of proprietary software projects. It is also worth mentioning that there is a lack of studies in small and medium-sized companies. Such new studies could shed light on the question if the knowledge accumulated through MCR is also present in small and medium companies' projects [18].

The surveyed practitioners were most positive about the IOF and CRP. Therefore, given the relatively low number of studies in these themes, we suggest conducting more research investigating how MCR affects and is affected. Davila and Nunes [18] share a similar insight, calling for more research on MCR process improvement.

Previous studies that surveyed practitioners and analyzed the impact of software engineering research [13, 27] could not establish any correlation (positive or negative) between research impact and practitioner interest. These observations are to a large extent in line with our results (Section 6.2). There is only one statistical significant correlation between the Support Systems

theme and research impact, indicating that statements containing publications with high citation count were considered as important to investigate (and vice versa). In Section 2.4, we report also that the past surveys on practitioner perception found that 67–71% of the research was seen positively. Looking at the results in Figure 7, we observe that 24 statements have more negative than positive ratings and 22 statements have more positive than negative ratings (48% positive). This is considerably lower than in the previous surveys. However, it could be attributed to the difference in the administered survey instruments. In our survey, the participants *had* to distribute negative, neutral, and positive perception according to a predefined distribution, i.e., they could not find all research negative or positive.

7.3 Validity Threats

Our philosophical worldview is pragmatic, which is often adopted by software engineering researchers [31]. We use therefore a mixed-method approach (systematic mapping study and survey) to answer our research questions. The commonly used categorizations to analyze and report validity threats, such as internal, external, conclusion, and construct validity of the post-positivist worldview (such as Wohlin et al. [39]), are adequate for quantitative research. However, they either do not capture all relevant threats for qualitative research or are formulated in a way that is not compatible with an interpretative worldview. The same argument can be made for threat categorizations originating from the interpretative worldview (such as Runeson et al. [33]). Petersen and Gencel [31] have therefore proposed a complementary validity threat categorization, based on work by Maxwell [29], that is adequate for a pragmatic worldview. We structure the remainder of the section according to their categorization and discuss validity threats w.r.t. each research method. Please note that repeatability (reproducibility/dependability) of this research is a function of all the threat categories together [31] and therefore not discussed individually.

7.3.1 Descriptive Validity. Threats in this category concern the factual accuracy of the made observations [31]. We have designed and used a data extraction form to collect the information from the reviewed studies. We copied contribution statements that can be traced back to the original studies. Furthermore, we have piloted the survey instrument with three practitioners to identify functional defects and usability issues. Extraction form, survey instrument, and collected data are available in an online repository [6].

7.3.2 Theoretical Validity. Threats in this category concern the ability to capture practically the intended theoretical concepts, which includes the control of confounding factors and biases [31].

Study identification. During the search we could have missed papers that could have been relevant but were not identified by the search string. We addressed this threat with a careful selection of keywords and not limiting the scope of the search to a particular population, comparison or outcome [24]. For the intervention criterion, we used variants of terms that we deemed relevant and associated with modern code reviews. Due to this association, we did not choose keywords for the population criterion (e.g., "software engineering") as they could have potentially reduced the number of relevant search hits. We have compared our primary studies with the set of other systematic literature studies (see Figure 11). We have identified 136 studies that the other reviews missed, while not identifying 69 studies that were found by the other reviews. Hence, there is a moderate threat that we missed relevant studies.

Study selection. Researcher bias could have led to the wrongful exclusion of relevant papers. We addressed this threat by including all three authors in the selection process who reviewed an independent set of studies. To align our selection criteria, we established objective inclusion and exclusion criteria, which we piloted and refined when we found divergences in our selection

(see *Selection* in Section 3.1). Furthermore, we adopted an inclusive selection process, postponing the decision on exclusion for unsure papers to the data extraction step, when it would be clear that the study did not contain the information we required to answer our research questions. When we excluded a paper, we documented the decision with the particular exclusion criterion.

Data extraction. Researcher bias could have led to a incorrect extraction of data. All three authors were involved in the data extraction as well. We also conducted two pilot extractions to gain a common interpretation of the extraction items. We revised the description of rigor and relevance criteria based on the pilot process. After the pilot process, we continued to extract data from primary studies with an overlap of 20% where we achieved high consensus.

Statement order. All survey participants received the statements in the same order. The participants may have tended to agree more to statements listed at the beginning than at the end of the survey. Our survey instrument was designed in such a way that the participants could change their rating anytime, i.e., also when they have seen all statements. Looking at our results, the themes that were judged early in the survey (IOF) seem to have received more agreement than later themes (SS). However, participants have provided a consistent rating for the human factor–related statements, independently of whether they appear in early or late positions in the survey. Therefore we assess the likelihood of this risk as low.

7.3.3 Internal Generalizability. Threats in this category concern the degree to which inferences from the collected data (quantitative or qualitative) are reasonable [31].

Statement ranking and factor analysis. We followed the recommendations for conducting the Q-Methodology [41], including factor analysis and interpretation. In addition, we report in detail how we interpret the quantitative results of the survey, providing a chain of evidence for our argumentation and conclusions.

Research amount and impact, and practitioners perceptions. There is a risk that practitioners understood that they need to judge if the topic in a statement affects them rather than whether research on the topic is important. We counteracted this threat by designing the survey instrument in a way that reminds the respondents what the purpose of the ranking of statements is. Furthermore, the free-text answers in the survey provide a good indication that the respondents correctly understood the ranking task.

Identification of research roadmap. We have based our analysis on the contributions reported by the original authors of the studies. In contrast, the gaps we highlighted are based on what has *not* been reported (i.e., researched). As such, the proposed research agenda contain speculation on what might be fruitful to research. However, we do provide argumentation and references to the original studies, allowing readers to follow our reasoning.

7.3.4 External Generalizability. Threats in this category concern the degree to which inferences from the collected data (quantitative or qualitative) can be extended to the population from which the sample was drawn.

Definition of statements. There is a risk that the statements were formulated either too generic or too specific, not reflecting all aspects of the research studies they represent. The statements were defined based on the primary studies collected in the 2018 SMS. We then extended the SMS to include papers published until 2021, classifying all new studies, except one, under the existing statements, which indicates that the initially defined statements were still useful after four years of research. It is, however, likely that with the advancement of MCR practice, new research emerges that requires also an update of the statements if they survey is replicated in the future.

Survey sample. As discussed in Section 2.2, the main difference between proprietary and open source software development, in relation to MCR, is the purpose of the MCR practice. The respondents of our survey mainly work in companies that primarily work in proprietary projects. In this

context, the main purpose of MCR is knowledge dissemination rather than building relationships to core developers [76]. Indeed, we observe in our survey results that research aspects of human factors in relation to MCR are perceived as less important (see Section 5.2). However, the factor analysis in Section 5.3 provides a more differentiated view, based on the profile of the survey respondents. For example, senior roles are more favourable toward human factors research in MCR than respondents with less professional experience. Nevertheless, future work could replicate the survey in open source communities, allowing a differential analysis.

Identification of research roadmap. While the inferences we draw from the reviewed studies may be sound *within* the sample we studied, there is a moderate threat that the future research we propose has already been conducted in the studies we did not review (see discussion on study identification in Section 7.3.2).

7.3.5 Interpretive Validity. Threats in this category concern the validity of the conclusions that are drawn by the researchers by interpreting the data [31].

Definition of themes. Researcher bias could have lead to an inaccurate classification of the studies in the SMS. We divided the primary studies for analysis among the authors. Each paper that was analyzed by one author was reviewed by the two other authors, and disagreements were discussed until a change in the classification was made or consensus was reached.

Definition of statements. Also the formulation of statements representing a study, used in the survey, could have been affected by researcher bias. To address this treat, we followed an iterative process in which we revised the statements and the association of papers to these statements. All three authors were involved in this process and reviewed each others formulations and classifications to check for consistency as well as allow for different perspectives on the material. There are two statements where more than one aspect is introduced: in the HOF theme, "age and experience" and "performance and quality" in the SS theme. A respondent may have an opinion on just one of the aspects and therefore misrepresent the rating of the other aspect. We asked the practitioners to provide explanations for extreme ratings, i.e., −3 and +3, which makes it possible to know on which aspects the practitioners focused. However, such explanations are not available for ratings other than −3 to +3. Since only 2 of 46 statements are affected, we judge the risk of misrepresentation as low.

Identification of research roadmap. Finally, the identification of gaps in the MCR research corpus could have been affected by researcher bias. The first and second authors conducted a workshop in which they independently read the MCR contributions in Section 4.2. Then, they discussed their ideas of what questions have *not* been answered by the reviewed research and which questions would be interesting to find an answer for, especially if there is some support from the survey results that a particular statement was perceived important to investigate by the practitioners. This initial formulation of research gaps was reviewed by the third author.

8 CONCLUSIONS

In this article, we conducted a systematic mapping study and a survey to provide an overview of the different research themes on MCR and analyze the practitioners' opinions on the importance of those themes. Based on the juxtaposition of these two perspectives on MCR research, we outline an agenda for future research on MCR that is based on the identified research gaps and the perceived importance by practitioners.

We have extracted the research contributions from 244 primary studies and summarized 15 years of MCR research in evidence briefings that can contribute to the knowledge transfer from academic research to practitioners. The five main themes of MCR research are as follows: (1) SS, (2) IOF, (3) CRP, (4) ION, and (5) HOF.

We conducted a survey to collect practitioners' opinions about 46 statements representing the research in the identified themes. As a result, we learned that practitioners are most positive about the CRP and IOF theme, with special focus on the impact of code reviews on product quality. However, these themes represent a minority of the reviewed MCR research (66 primary studies). At the same time, the respondents are most negative about HOF- and SS-related research, which constitute together a majority of the reviewed research (108 primary studies). These results indicate that there is a misalignment between the state of the art and the themes deemed important by most respondents of our survey. In addition, we found that the research that has been perceived positively by practitioners is generally also more frequently cited, i.e., has a larger research impact.

Finally, as there has been an increased interest in reviewing MCR research in recent years, we analyzed other systematic literature reviews and mapping studies. Due to the different research questions of the respective studies, there is a varying overlap of the reviewed primary studies. Still, we find our observations on the potential gaps in MCR research corroborated. Analyzing the data extracted from the reviewed primary studies and guided by the answers from the survey, we propose 19 new research questions we deem worth investigating in future MCR research.

ACKNOWLEDGMENTS

We acknowledge all practitioners who contributed to our investigation.

REFERENCES

[1] Everton L. G. Alves, Myoungkyu Song, Tiago Massoni, Patricia D. L. Machado, and Miryung Kim. 2017. Refactoring inspection support for manual refactoring edits. *IEEE Trans. Softw. Eng.* 44, 4 (2017), 365–383.

[2] Aybuke Aurum, Håkan Petersson, and Claes Wohlin. 2002. State-of-the-art: Software inspections after 25 years. *Softw. Test. Verif. Reliabil.* 12, 3 (2002), 133–154.

[3] Alberto Bacchelli and Christian Bird. 2013. Expectations, outcomes, and challenges of modern code review. In *Proceedings of the International Conference on Software Engineering (ICSE'13)*. IEEE, 712–721.

[4] Deepika Badampudi, Ricardo Britto, and Michael Unterkalmsteiner. 2019. Modern code reviews - preliminary results of a systematic mapping study. In *Proceedings of the Evaluation and Assessment on Software Engineering (EASE'19)*. ACM, 340–345.

[5] Deepika Badampudi, Michael Unterkalmsteiner, and Ricardo Britto. 2021. Evidence briefings on modern code reviews. https://doi.org/10.5281/zenodo.5093742

[6] Deepika Badampudi, Michael Unterkalmsteiner, and Ricardo Britto. 2022. Data used in modern code review mapping study and survey. https://doi.org/10.5281/zenodo.7464947

[7] Gabriele Bavota and Barbara Russo. 2015. Four eyes are better than two: On the impact of code reviews on software quality. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME'15)*. IEEE, 81–90.

[8] Marten Brouwer. 1999. Q is accounting for tastes. *J. Advert. Res.* 39, 2 (1999), 35–35.

[9] Steven R. Brown. 1993. A primer on Q methodology. *Oper. Subject.* 16, 3/4 (1993), 91–138.

[10] Bill Brykczynski. 1999. A survey of software inspection checklists. *ACM SIGSOFT Softw. Eng. Not.* 24, 1 (1999), 82.

[11] Bruno Cartaxo, Gustavo Pinto, Baldoino Fonseca, Márcio Ribeiro, Pedro Pinheiro, Sergio Soares, and Maria Teresa Baldassarre. 2019. Software engineering research community viewpoints on rapid reviews. In *Proceedings of the 13th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM'19)*.

[12] Bruno Cartaxo, Gustavo Pinto, Elton Vieira, and Sérgio Soares. 2016. Evidence briefings: Towards a medium to transfer knowledge from systematic reviews to practitioners. In *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. 1–10.

[13] Jeffrey C. Carver, Oscar Dieste, Nicholas A. Kraft, David Lo, and Thomas Zimmermann. 2016. How practitioners perceive the relevance of esem research. In *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. 1–10.

[14] H. Alperen Çetin, Emre Doğan, and Eray Tüzün. 2021. A review of code reviewer recommendation studies: Challenges and future directions. *Sci. Comput. Program.* 208 (2021), 102652.

[15] Zhiyuan Chen, Young-Woo Kwon, and Myoungkyu Song. 2018. Clone refactoring inspection by summarizing clone refactorings and detecting inconsistent changes during software evolution. *J. Softw.: Evol. Process* 30, 10 (2018), e1951.

[16] Flavia Coelho, Tiago Massoni, and Everton L. G. Alves. 2019. Refactoring-aware code review: A systematic mapping study. In *Proceedings of the IEEE/ACM 3rd International Workshop on Refactoring (IWoR'19)*. IEEE, 63–66.

[17] D. S. Cruzes and T. Dyba. 2011. Recommended steps for thematic synthesis in software engineering. In *Proceedings of the International Symposium on Empirical Software Engineering and Measurement*. 275–284.
[18] Nicole Davila and Ingrid Nunes. 2021. A systematic literature review and taxonomy of modern code review. *J. Syst. Softw.* 177 (2021), 110951.
[19] Charles H. Davis and Carolyn Michelle. 2011. Q methodology in audience research: Bridging the qualitative/quantitative "divide." *J. Aud. Recept. Stud.* 8, 2 (2011), 559–593.
[20] M. E. Fagan. 1976. Design and code inspections to reduce errors in program development. *IBM Syst. J.* 15, 3 (1976), 182–211.
[21] Xavier Franch, Daniel Mendez, Andreas Vogelsang, Rogardt Heldal, Eric Knauss, Marc Oriol, Guilherme Travassos, Jeffrey Clark Carver, and Thomas Zimmermann. 2020. How do practitioners perceive the relevance of requirements engineering research? *IEEE Trans. Softw. Eng.* 48, 6 (2020).
[22] Theresia Devi Indriasari, Andrew Luxton-Reilly, and Paul Denny. 2020. A review of peer code review in higher education. *ACM Trans. Comput. Educ.* 20, 3 (2020), 1–25.
[23] Martin Ivarsson and Tony Gorschek. 2011. A method for evaluating rigor and industrial relevance of technology evaluations. *Empir. Softw. Eng.* 16, 3 (2011), 365–395.
[24] Barbara A. Kitchenham and Stuart Charters. 2007. *Guidelines for Performing Systematic Literature Reviews in Software Engineering*. Technical Report EBSE-2007-01. Software Engineering Group, Keele University and Department of Computer Science, University of Durham, United Kingdom.
[25] Sami Kollanus and Jussi Koskinen. 2009. Survey of software inspection research. *Open Softw. Eng. J.* 3, 1 (2009).
[26] Oliver Laitenberger and Jean-Marc DeBaud. 2000. An encompassing life cycle centric survey of software inspection. *J. Syst. Softw.* 50, 1 (2000), 5–31.
[27] David Lo, Nachiappan Nagappan, and Thomas Zimmermann. 2015. How practitioners perceive the relevance of software engineering research. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*. 415–425.
[28] F. Macdonald, J. Miller, A. Brooks, M. Roper, and M. Wood. 1995. A review of tool support for software inspection. In *Proceedings of the 7th International Workshop on Computer-Aided Software Engineering*. IEEE, 340–349.
[29] Joseph Maxwell. 1992. Understanding and validity in qualitative research. *Harv. Educ. Rev.* 62, 3 (1992), 279–301.
[30] Sumaira Nazir, Nargis Fatima, and Suriayati Chuprat. 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*. ACM, 210–215.
[31] Kai Petersen and Cigdem Gencel. 2013. Worldviews, research methods, and their relationship to validity in empirical software engineering research. In *Proceedings of the Joint Conference of the 23nd International Workshop on Software Measurement (IWSM'13) and the 8th International Conference on Software Process and Product Measurement*. 81–89.
[32] Kai Petersen, Sairam Vakkalanka, and Ludwik Kuzniarz. 2015. Guidelines for conducting systematic mapping studies in software engineering: An update. *Inf. Softw. Technol.* 64 (2015), 1–18.
[33] Per Runeson and Martin Höst. 2009. Guidelines for conducting and reporting case study research in software engineering. *Empir. Softw. Eng.* 14, 2 (2009), 131–164.
[34] Caitlin Sadowski, Emma Söderberg, Luke Church, Michal Sipko, and Alberto Bacchelli. 2018. Modern code review: A case study at Google. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP'18)*. ACM, New York, NY, 181–190.
[35] Mojtaba Shahin, Muhammad Ali Babar, and Liming Zhu. 2017. Continuous integration, delivery and deployment: A systematic review on approaches, tools, challenges and practices. *IEEE Access* 5 (2017), 3909–3943.
[36] Dong Wang, Yuki Ueda, Raula Gaikovina Kula, Takashi Ishio, and Kenichi Matsumoto. 2019. The evolution of code review research: A systematic mapping study. arXiv:1911.08816 [cs.SE]
[37] Dong Wang, Yuki Ueda, Raula Gaikovina Kula, Takashi Ishio, and Kenichi Matsumoto. 2021. Can we benchmark Code Review studies? A systematic mapping study of methodology, dataset, and metric. *J. Syst. Softw.* 180 (2021), 111009.
[38] Roel Wieringa, Neil Maiden, Nancy Mead, and Colette Rolland. 2005. Requirements engineering paper classification and evaluation criteria: A proposal and a discussion. *Requir. Eng.* 11, 1 (December 2005), 102–107.
[39] Claes Wohlin. 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 (EASE'14)*. ACM, 1–10.
[40] Claes Wohlin, Per Runeson, Paulo Anselmo da Mota Silveira Neto, Emelie Engström, Ivan do Carmo Machado, and Eduardo Santana De Almeida. 2013. On the reliability of mapping studies in software engineering. *J. Syst. Softw.* 86, 10 (2013), 2594–2610.
[41] Aiora Zabala and Unai Pascual. 2016. Bootstrapping Q methodology to improve the understanding of human perspectives. *PLoS One* 11, 2 (2016), e0148087.

MAPPING STUDY REFERENCES

[42] Toufique Ahmed, Amiangshu Bosu, Anindya Iqbal, and Shahram Rahimi. 2017. SentiCR: A customized sentiment analysis tool for code review interactions. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE'17)*. IEEE, 106–111.

[43] Wisam Haitham Abbood Al-Zubaidi, Patanamon Thongtanunam, Hoa Khanh Dam, Chakkrit Tantithamthavorn, and Aditya Ghose. 2020. *Workload-Aware Reviewer Recommendation Using a Multi-Objective Search-Based Approach*. Association for Computing Machinery, New York, NY, 21–30. https://doi.org/10.1145/3416508.3417115

[44] Adam Alami, Marisa Leavitt Cohn, and Andrzej Wąsowski. 2019. Why does code review work for open source software communities? In *Proceedings of the IEEE/ACM 41st International Conference on Software Engineering (ICSE'19)*. IEEE, 1073–1083.

[45] Eman Abdullah AlOmar, Hussein AlRubaye, Mohamed Wiem Mkaouer, Ali Ouni, and Marouane Kessentini. 2021. Refactoring practices in the context of modern code review: An industrial case study at Xerox. In *Proceedings of the IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP'21)*. IEEE, 348–357.

[46] Everton L. G. Alves, Myoungkyu Song, and Miryung Kim. 2014. RefDistiller: A refactoring aware code review tool for inspecting manual refactoring edits. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 751–754.

[47] Hirohisa Aman. 2013. 0-1 programming model-based method for planning code review using bug fix history. In *Proceedings of the 20th Asia-Pacific Software Engineering Conference (APSEC'13)*, Vol. 2. IEEE, 37–42.

[48] F. Armstrong, F. Khomh, and B. Adams. 2017. Broadcast vs. unicast review technology: Does it matter? In *Proceedings of the IEEE International Conference on Software Testing, Verification and Validation (ICST)*. 219–229.

[49] Sumit Asthana, Rahul Kumar, Ranjita Bhagwan, Christian Bird, Chetan Bansal, Chandra Maddila, Sonu Mehta, and B. Ashok. 2019. WhoDo: Automating reviewer suggestions at scale. In *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'19)*. Association for Computing Machinery, New York, NY, 937–945. https://doi.org/10.1145/3338906.3340449

[50] Jai Asundi and Rajiv Jayant. 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 (HICSS'07)*. IEEE, 166c–166c.

[51] Krishna Teja Ayinala, Kwok Sun Cheng, Kwangsung Oh, and Myoungkyu Song. 2020. Tool support for code change inspection with deep learning in evolving software. In *Proceedings of the IEEE International Conference on Electro Information Technology (EIT'20)*. IEEE, 013–017.

[52] Krishna Teja Ayinala, Kwok Sun Cheng, Kwangsung Oh, Teukseob Song, and Myoungkyu Song. 2020. Code inspection support for recurring changes with deep learning in evolving software. In *Proceedings of the IEEE 44th Annual Computers, Software, and Applications Conference (COMPSAC'20)*. 931–942. https://doi.org/10.1109/COMPSAC48688.2020.0-149

[53] Muhammad Ilyas Azeem, Qiang Peng, and Qing Wang. 2020. Pull request prioritization algorithm based on acceptance and response probability. In *Proceedings of the IEEE 20th International Conference on Software Quality, Reliability and Security (QRS'20)*. 231–242. https://doi.org/10.1109/QRS51102.2020.00041

[54] Alberto Bacchelli and Christian Bird. 2013. Expectations, outcomes, and challenges of modern code review. In *Proceedings International Conference on Software Engineering (ICSE'13)*. IEEE, 712–721.

[55] Vipin Balachandran. 2013. Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation. In *Proceedings of the International Conference on Software Engineering*. IEEE, 931–940.

[56] Vipin Balachandran. 2020. Reducing accidental clones using instant clone search in automatic code review. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME'20)*. 781–783. https://doi.org/10.1109/ICSME46990.2020.00089

[57] Faruk Balcı, Dilruba Sultan Haliloğlu, Onur Şahin, Cankat Tilki, Mehmet Ata Yurtsever, and Eray Tüzün. 2021. Augmenting code review experience through visualization. In *Proceedings of the Working Conference on Software Visualization (VISSOFT'21)*. IEEE, 110–114.

[58] Mike Barnett, Christian Bird, Joao Brunet, and Shuvendu K. Lahiri. 2015. Helping developers help themselves: Automatic decomposition of code review changesets. In *Proceedings of the IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 134–144.

[59] Tobias Baum, Fabian Kortum, Kurt Schneider, Arthur Brack, and Jens Schauder. 2017. Comparing pre-commit reviews and post-commit reviews using process simulation. *J. Softw.: Evol. Process* 29, 11 (2017), e1865.

[60] Tobias Baum, Olga Liskin, Kai Niklas, and Kurt Schneider. 2016. A faceted classification scheme for change-based industrial code review processes. In *Proceedings of the IEEE International Conference on Software Quality, Reliability and Security (QRS'16)*. IEEE, 74–85.

[61] Tobias Baum, Olga Liskin, Kai Niklas, and Kurt Schneider. 2016. Factors influencing code review processes in industry. In *Proceedings of the 24th ACM Sigsoft International Symposium on Foundations of Software Engineering*. 85–96.

[62] Tobias Baum and Kurt Schneider. 2016. On the need for a new generation of code review tools. In *International Conference on Product-Focused Software Process Improvement*. Springer, 301–308.

[63] Tobias Baum, Kurt Schneider, and Alberto Bacchelli. 2017. On the optimal order of reading source code changes for review. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME'17)*. IEEE, 329–340.

[64] Tobias Baum, Kurt Schneider, and Alberto Bacchelli. 2019. Associating working memory capacity and code change ordering with code review performance. *Empir. Softw. Eng.* 24, 4 (2019), 1762–1798.

[65] O. Baysal, O. Kononenko, R. Holmes, and M. W. Godfrey. 2012. The secret life of patches: A firefox case study. In *Proceedings of the 19th Working Conference on Reverse Engineering*. 447–455.

[66] Olga Baysal, Oleksii Kononenko, Reid Holmes, and Michael W. Godfrey. 2016. Investigating technical and non-technical factors influencing modern code review. *Empir. Softw. Eng.* 21, 3 (2016), 932–959.

[67] Andrew Begel and Hana Vrzakova. 2018. Eye movements in code review. In *Proceedings of the Workshop on Eye Movements in Programming (EMIP'18)*. ACM, New York, NY.

[68] Moritz Beller, Alberto Bacchelli, Andy Zaidman, and Elmar Juergens. 2014. Modern code reviews in open-source projects: Which problems do they fix? In *Proceedings of the 11th Working Conference on Mining Software Repositories*. 202–211.

[69] Mario Bernhart and Thomas Grechenig. 2013. On the understanding of programs with continuous code reviews. In *Proceedings of the 21st International Conference on Program Comprehension (ICPC'13)*. IEEE, 192–198.

[70] Mario Bernhart, Andreas Mauczka, and Thomas Grechenig. 2010. Adopting code reviews for agile software development. In *Proceedings of the Agile Conference*. IEEE, 44–47.

[71] M. Bernhart, S. Strobl, A. Mauczka, and T. Grechenig. 2012. Applying continuous code reviews in airport operations software. In *Proceedings of the 12th International Conference on Quality Software*. 214–219.

[72] Christian Bird, Trevor Carnahan, and Michaela Greiler. 2015. Lessons learned from building and deploying a code review analytics platform. In *Proceedings of the 12th Working Conference on Mining Software Repositories*. IEEE, 191–201.

[73] Amiangshu Bosu and Jeffrey C. Carver. 2012. Peer code review in open source communities using reviewboard. In *Proceedings of the ACM 4th Annual Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU'12)*. ACM, New York, NY, 17–24.

[74] Amiangshu Bosu and Jeffrey C. Carver. 2013. Impact of peer code review on peer impression formation: A survey. In *Proceedings of the ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. IEEE, 133–142.

[75] Amiangshu Bosu and Jeffrey C. Carver. 2014. Impact of developer reputation on code review outcomes in OSS projects: An empirical investigation. In *Proceedings of the 8th International Symposium on Empirical Software Engineering and Measurement*. ACM, 33.

[76] Amiangshu Bosu, Jeffrey C. Carver, Christian Bird, Jonathan Orbeck, and Christopher Chockley. 2016. 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 (2016), 56–75.

[77] Amiangshu Bosu, Jeffrey C. Carver, Munawar Hafiz, Patrick Hilley, and Derek Janni. 2014. Identifying the characteristics of vulnerable code changes: An empirical study. In *Proceedings of the 22nd International Symposium on Foundations of Software Engineering*. ACM, 257–268.

[78] Amiangshu Bosu, Michaela Greiler, and Christian Bird. 2015. Characteristics of useful code reviews: An empirical study at microsoft. In *Proceedings of the IEEE/ACM 12th Working Conference on Mining Software Repositories*. IEEE, 146–156.

[79] Rodrigo Brito and Marco Tulio Valente. 2021. RAID: Tool support for refactoring-aware code reviews. In *Proceedings of the IEEE/ACM 29th International Conference on Program Comprehension (ICPC'21)*. IEEE, 265–275.

[80] Fabio Calefato, Filippo Lanubile, Federico Maiorano, and Nicole Novielli. 2018. Sentiment polarity detection for software development. *Empir. Softw. Eng.* 23, 3 (2018), 1352–1382.

[81] Nathan Cassee, Bogdan Vasilescu, and Alexander Serebrenik. 2020. The silent helper: The impact of continuous integration on code reviews. In *Proceedings of the IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER'20)*. IEEE, 423–434.

[82] Maria Caulo, Bin Lin, Gabriele Bavota, Giuseppe Scanniello, and Michele Lanza. 2020. Knowledge transfer in modern code review. In *Proceedings of the 28th International Conference on Program Comprehension*. 230–240.

[83] K. R. Amudha Chandrika and J. Amudha. 2018. A fuzzy inference system to recommend skills for source code review using eye movement data. *J. Intell. Fuzzy Syst.* 34, 3 (2018), 1743–1754.

[84] Ashish Chopra, Morgan Mo, Samuel Dodson, Ivan Beschastnikh, Sidney S. Fels, and Dongwook Yoon. 2021. "@ alex, this fixes# 9": Analysis of referencing patterns in pull request discussions. *Proc. ACM Hum.-Comput. Interact.* 5, CSCW2 (2021), 1–25.

[85] Moataz Chouchen, Ali Ouni, Raula Gaikovina Kula, Dong Wang, Patanamon Thongtanunam, Mohamed Wiem Mkaouer, and Kenichi Matsumoto. 2021. Anti-patterns in modern code review: Symptoms and prevalence. In *Proceedings of the IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER'21)*. IEEE, 531–535.

[86] Moataz Chouchen, Ali Ouni, Mohamed Wiem Mkaouer, Raula Gaikovina Kula, and Katsuro Inoue. 2021. WhoReview: A multi-objective search-based approach for code reviewers recommendation in modern code review. *Appl. Soft Comput.* 100 (2021), 106908. https://doi.org/10.1016/j.asoc.2020.106908

[87] Aleksandr Chueshev, Julia Lawall, Reda Bendraou, and Tewfik Ziadi. 2020. Expanding the number of reviewers in open-source projects by recommending appropriate developers. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME'20)*. 499–510. https://doi.org/10.1109/ICSME46990.2020.00054

[88] Flávia Coelho, Nikolaos Tsantalis, Tiago Massoni, and Everton L. G. Alves. 2021. An empirical study on refactoring-inducing pull requests. In *Proceedings of the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM'21)*. 1–12.

[89] Atacílio Cunha, Tayana Conte, and Bruno Gadelha. 2021. Code review is just reviewing code? A qualitative study with practitioners in industry. In *Proceedings of the Brazilian Symposium on Software Engineering*. 269–274.

[90] Jacek Czerwonka, Michaela Greiler, and Jack Tilford. 2015. Code reviews do not find bugs: How the current code review best practice slows us down. In *Proceedings of the 37th International Conference on Software Engineering, Volume 2 (ICSE'15)*. IEEE Press, 27–28.

[91] Anastasia Danilova, Alena Naiakshina, Anna Rasgauski, and Matthew Smith. 2021. Code reviewing as methodology for online security studies with developers-a case study with freelancers on password storage. In *Proceedings of the 17th Symposium on Usable Privacy and Security (SOUPS'21)*. 397–416.

[92] Manoel Limeira de Lima Júnior, Daricélio Moreira Soares, Alexandre Plastino, and Leonardo Murta. 2015. Developers assignment for analyzing pull requests. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*. ACM, 1567–1572.

[93] Marco di Biase, Magiel Bruntink, and Alberto Bacchelli. 2016. A security perspective on code review: The case of chromium. In *Proceedings of the IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM'16)*. IEEE, 21–30.

[94] Marco di Biase, Magiel Bruntink, Arie van Deursen, and Alberto Bacchelli. 2019. The effects of change decomposition on code review–a controlled experiment. *PeerJ Comput. Sci.* 5 (2019), e193.

[95] Eduardo Witter dos Santos and Ingrid Nunes. 2017. Investigating the effectiveness of peer code review in distributed software development. In *Proceedings of the 31st Brazilian Symposium on Software Engineering (SBES'17)*. ACM, New York, NY, 84–93.

[96] Tobias Dürschmid. 2017. Continuous code reviews: A social coding tool for code reviews inside the IDE. In *Companion to the 1st International Conference on the Art, Science and Engineering of Programming*. ACM, 41.

[97] Felipe Ebert, Fernando Castor, Nicole Novielli, and Alexander Serebrenik. 2017. Confusion detection in code reviews. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME'17)*. IEEE, 549–553.

[98] Felipe Ebert, Fernando Castor, Nicole Novielli, and Alexander Serebrenik. 2018. Communicative intention in code review questions. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME'18)*. IEEE, 519–523.

[99] Felipe Ebert, Fernando Castor, Nicole Novielli, and Alexander Serebrenik. 2021. An exploratory study on confusion in code reviews. *Empir. Softw. Eng.* 26, 1 (2021), 1–48.

[100] Vasiliki Efstathiou and Diomidis Spinellis. 2018. Code review comments: Language matters. In *Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results*. ACM, 69–72.

[101] Carolyn D. Egelman, Emerson Murphy-Hill, Elizabeth Kammer, Margaret Morrow Hodges, Collin Green, Ciera Jaspan, and James Lin. 2020. Predicting developers' negative feelings about code review. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering (ICSE'20)*. IEEE, 174–185.

[102] Ikram El Asri, Noureddine Kerzazi, Gias Uddin, Foutse Khomh, and M. A. Janati Idrissi. 2019. An empirical study of sentiments in code reviews. *Inf. Softw. Technol.* 114 (2019), 37–54.

[103] Muntazir Fadhel and Emil Sekerinski. 2021. Striffs: Architectural component diagrams for code reviews. In *Proceedings of the International Conference on Code Quality (ICCQ'21)*. IEEE, 69–78.

[104] George Fairbanks. 2019. Better code reviews with design by contract. *IEEE Softw.* 36, 6 (2019), 53–56. https://doi.org/10.1109/MS.2019.2934192

[105] Yuanrui Fan, Xin Xia, David Lo, and Shanping Li. 2018. Early prediction of merged code changes to prioritize reviewing tasks. *Empir. Softw. Eng.* (2018), 1–48.

[106] Mikołaj Fejzer, Piotr Przymus, and Krzysztof Stencel. 2018. Profile based recommendation of code reviewers. *J. Intell. Inf. Syst.* 50, 3 (2018), 597–619.
[107] Isabella Ferreira, Jinghui Cheng, and Bram Adams. 2021. The "Shut the f** k up" phenomenon: Characterizing incivility in open source code review discussions. *Proc. ACM Hum.-Comput. Interact.* 5, CSCW2 (2021), 1–35.
[108] Wojciech Frącz and Jacek Dajda. 2017. Experimental validation of source code reviews on mobile devices. In *International Conference on Computational Science and Its Applications*. Springer, 533–547.
[109] Leonardo B. Furtado, Bruno Cartaxo, Christoph Treude, and Gustavo Pinto. 2020. How successful are open source contributions from countries with different levels of human development? *IEEE Softw.* 38, 2 (2020), 58–63.
[110] Lorenzo Gasparini, Enrico Fregnan, Larissa Braz, Tobias Baum, and Alberto Bacchelli. 2021. ChangeViz: Enhancing the GitHub pull request interface with method call information. In *Proceedings of the Working Conference on Software Visualization (VISSOFT'21)*. IEEE, 115–119.
[111] Xi Ge, Saurabh Sarkar, and Emerson Murphy-Hill. 2014. Towards refactoring-aware code review. In *Proceedings of the 7th International Workshop on Cooperative and Human Aspects of Software Engineering*. ACM, 99–102.
[112] Çağdaş Evren Gerede and Zeki Mazan. 2018. Will it pass? Predicting the outcome of a source code review. *Turk. J. Electr. Eng. Comput. Sci.* 26, 3 (2018), 1343–1353.
[113] Daniel M. German, Gregorio Robles, Germán Poo-Caamaño, Xin Yang, Hajimu Iida, and Katsuro Inoue. 2018. "Was my contribution fairly reviewed?" A framework to study the perception of fairness in modern code reviews. In *Proceedings of the IEEE/ACM 40th International Conference on Software Engineering (ICSE'18)*. IEEE, 523–534.
[114] Mehdi Golzadeh, Alexandre Decan, and Tom Mens. 2019. On the effect of discussions on pull request decisions. In *Proceedings of the 18th Belgium-Netherlands Software Evolution Workshop (BENEVOL'19)*.
[115] Jesus M. Gonzalez-Barahona, Daniel Izquierdo-Cortazar, Gregorio Robles, and Alvaro del Castillo. 2014. Analyzing gerrit code review parameters with bicho. *Electr. Commun. EASST* (2014).
[116] Jesús M. González-Barahona, Daniel Izquierdo-Cortázar, Gregorio Robles, and Mario Gallegos. 2014. Code review analytics: WebKit as case study. In *Open Source Software: Mobile Open Source Technologies*. Springer, 1–10.
[117] Tanay Gottigundala, Siriwan Sereesathien, and Bruno da Silva. 2021. Qualitatively analyzing PR rejection reasons from conversations in open-source projects. In *Proceedings of the 13th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE'21)*. IEEE, 109–112.
[118] Bo Guo, Young-Woo Kwon, and Myoungkyu Song. 2019. Decomposing composite changes for code review and regression test selection in evolving software. *J. Comput. Sci. Technol.* 34, 2 (2019), 416–436.
[119] DongGyun Han, Chaiyong Ragkhitwetsagul, Jens Krinke, Matheus Paixao, and Giovanni Rosa. 2020. Does code review really remove coding convention violations?. In *Proceedings of the IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM'20)*. IEEE, 43–53.
[120] Xiaofeng Han, Amjed Tahir, Peng Liang, Steve Counsell, and Yajing Luo. 2021. Understanding code smell detection via code review: A study of the openstack community. In *Proceedings of the IEEE/ACM 29th International Conference on Program Comprehension (ICPC'21)*. IEEE, 323–334.
[121] Quinn Hanam, Ali Mesbah, and Reid Holmes. 2019. Aiding code change understanding with semantic change impact analysis. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME'19)*. IEEE, 202–212.
[122] Christoph Hannebauer, Michael Patalas, Sebastian Stünkel, and Volker Gruhn. 2016. Automatically recommending code reviewers based on their expertise: An empirical comparison. In *Proceedings of the 31st International Conference on Automated Software Engineering*. ACM, 99–110.
[123] Masum Hasan, Anindya Iqbal, Mohammad Rafid Ul Islam, A. J. M. Imtiajur Rahman, and Amiangshu Bosu. 2021. Using a balanced scorecard to identify opportunities to improve code review effectiveness: An industrial experience report. 26, 6 (2021). https://doi.org/10.1007/s10664-021-10038-w
[124] Florian Hauser, Stefan Schreistter, Rebecca Reuter, Jurgen Horst Mottok, Hans Gruber, Kenneth Holmqvist, and Nick Schorr. 2020. Code reviews in C++ preliminary results from an eye tracking study. In *Proceedings of the ACM Symposium on Eye Tracking Research and Applications*. 1–5.
[125] V. J. Hellendoorn, P. T. Devanbu, and A. Bacchelli. 2015. Will they like this? Evaluating code contributions with language models. In *Proceedings of the IEEE/ACM 12th Working Conference on Mining Software Repositories*. 157–167.
[126] Vincent J. Hellendoorn, Jason Tsay, Manisha Mukherjee, and Martin Hirzel. 2021. *Towards Automating Code Review at Scale*. Association for Computing Machinery, New York, NY, 1479–1482. https://doi.org/10.1145/3468264.3473134
[127] Austin Z. Henley, KIvanç Muçlu, Maria Christakis, Scott D. Fleming, and Christian Bird. 2018. CFar: A tool to increase communication, productivity, and review quality in collaborative code reviews. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*. ACM, 157.
[128] Martin Hentschel, Reiner Hähnle, and Richard Bubel. 2016. Can formal methods improve the efficiency of code reviews? In *International Conference on Integrated Formal Methods*. Springer, 3–19.

[129] Toshiki Hirao, Akinori Ihara, and Ken-ichi Matsumoto. 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*. IBM, 248–251.

[130] Toshiki Hirao, Akinori Ihara, Yuki Ueda, Passakorn Phannachitta, and Ken-ichi Matsumoto. 2016. The impact of a low level of agreement among reviewers in a code review process. In *IFIP International Conference on Open Source Systems*. Springer, 97–110.

[131] Toshiki Hirao, Raula Gaikovina Kula, Akinori Ihara, and Kenichi Matsumoto. 2019. Understanding developer commenting in code reviews. *IEICE Trans. Inf. Syst.* 102, 12 (2019), 2423–2432.

[132] Toshiki Hirao, Shane McIntosh, Akinori Ihara, and Kenichi Matsumoto. 2019. The review linkage graph for code review analytics: A recovery approach and empirical study. In *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'19)*. Association for Computing Machinery, New York, NY, 578–589. https://doi.org/10.1145/3338906.3338949

[133] Gerard J. Holzmann. 2010. SCRUB: A tool for code reviews. *Innov. Syst. Softw. Eng.* 6, 4 (2010), 311–318.

[134] Syeda Sumbul Hossain, Yeasir Arafat, Md Hossain, Md Arman, Anik Islam, et al. 2020. Measuring the effectiveness of software code review comments. In *International Conference on Advances in Computing and Data Sciences*. Springer, 247–257.

[135] Dongyang Hu, Yang Zhang, Junsheng Chang, Gang Yin, Yue Yu, and Tao Wang. 2019. Multi-reviewing pull-requests: An exploratory study on GitHub OSS projects. *Inf. Softw. Technol.* 115 (2019), 1–4.

[136] Yuan Huang, Nan Jia, Xiangping Chen, Kai Hong, and Zibin Zheng. 2018. Salient-class location: Help developers understand code change in code review. In *Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 770–774.

[137] Yuan Huang, Nan Jia, Xiangping Chen, Kai Hong, and Zibin Zheng. 2020. Code review knowledge perception: Fusing multi-features for salient-class location. *IEEE Trans. Softw. Eng.* (2020).

[138] Yu Huang, Kevin Leach, Zohreh Sharafi, Nicholas McKay, Tyler Santander, and Westley Weimer. 2020. Biases and differences in code review using medical imaging and eye-tracking: Genders, humans, and machines. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 456–468.

[139] M. Ichinco. 2014. Towards crowdsourced large-scale feedback for novice programmers. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'14)*. 189–190.

[140] Daniel Izquierdo, Jesus Gonzalez-Barahona, Lars Kurth, and Gregorio Robles. 2018. Software development analytics for Xen: Why and how. *IEEE Softw.* (2018).

[141] Daniel Izquierdo-Cortazar, Lars Kurth, Jesus M. Gonzalez-Barahona, Santiago Dueñas, and Nelson Sekitoleko. 2016. Characterization of the Xen project code review process: An experience report. In *Proceedings of the IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR'16)*. IEEE, 386–390.

[142] Daniel Izquierdo-Cortazar, Nelson Sekitoleko, Jesus M. Gonzalez-Barahona, and Lars Kurth. 2017. Using metrics to track code review performance. In *Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering*. ACM, 214–223.

[143] Jing Jiang, Jin Cao, and Li Zhang. 2017. An empirical study of link sharing in review comments. In *Software Engineering and Methodology for Emerging Domains*. Springer, 101–114.

[144] Jing Jiang, Jia-Huan He, and Xue-Yuan Chen. 2015. Coredevrec: Automatic core member recommendation for contribution evaluation. *J. Comput. Sci. Technol.* 30, 5 (2015), 998–1016.

[145] Jing Jiang, Yun Yang, Jiahuan He, Xavier Blanc, and Li Zhang. 2017. Who should comment on this pull request? Analyzing attributes for more accurate commenter recommendation in pull-based development. *Inf. Softw. Technol.* 84 (2017), 48–62.

[146] Marian Jureczko, Łukasz Kajda, and Paweł Górecki. 2020. Code review effectiveness: An empirical study on selected factors influence. *IET Softw.* 14, 7 (2020), 794–805.

[147] Akshay Kalyan, Matthew Chiam, Jing Sun, and Sathiamoorthy Manoharan. 2016. A collaborative code review platform for github. In *Proceedings of the 21st International Conference on Engineering of Complex Computer Systems (ICECCS'16)*. IEEE, 191–196.

[148] Ritu Kapur, Balwinder Sodhi, Poojith U. Rao, and Shipra Sharma. 2021. Using paragraph vectors to improve our existing code review assisting tool-CRUSO. In *Proceedings of the 14th Innovations in Software Engineering Conference (formerly known as India Software Engineering Conference)*. 1–11.

[149] David Kavaler, Premkumar Devanbu, and Vladimir Filkov. 2019. Whom are you going to call? Determinants of@-mentions in github discussions. *Empir. Softw. Eng.* 24, 6 (2019), 3904–3932.

[150] Noureddine Kerzazi and Ikram El Asri. 2016. Who can help to review this piece of code? In *Collaboration in a Hyperconnected World*, Hamideh Afsarmanesh, Luis M. Camarinha-Matos, and António Lucas Soares (Eds.). Springer, 289–301.

[151] Shivam Khandelwal, Sai Krishna Sripada, and Y. Raghu Reddy. 2017. Impact of gamification on code review process: An experimental study. In *Proceedings of the 10th Innovations in Software Engineering Conference (ISEC'17)*. ACM, New York, NY, 122–126.

[152] Jungil Kim and Eunjoo Lee. 2018. Understanding review expertise of developers: A reviewer recommendation approach based on latent dirichlet allocation. *Symmetry* 10, 4 (2018), 114.

[153] N. Kitagawa, H. Hata, A. Ihara, K. Kogiso, and K. Matsumoto. 2016. Code review participation: Game theoretical modeling of reviewers in gerrit datasets. In *Proceedings of the IEEE/ACM Cooperative and Human Aspects of Software Engineering (CHASE'16)*. 64–67.

[154] O. Kononenko, O. Baysal, and M. W. Godfrey. 2016. Code review quality: How developers see it. In *Proceedings of the IEEE/ACM 38th International Conference on Software Engineering (ICSE'16)*. 1028–1038.

[155] Oleksii Kononenko, Olga Baysal, Latifa Guerrouj, Yaxin Cao, and Michael W. Godfrey. 2015. Investigating code review quality: Do people and participation matter? In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME'15)*. IEEE, 111–120.

[156] O. Kononenko, T. Rose, O. Baysal, M. Godfrey, D. Theisen, and B. de Water. 2018. Studying pull request merges: A case study of Shopify's active merchant. In *Proceedings of the IEEE/ACM 40th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP'18)*. 124–133.

[157] V. Kovalenko and A. Bacchelli. 2018. Code review for newcomers: Is it different? In *Proceedings of the IEEE/ACM 11th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE'18)*. 29–32.

[158] Vladimir Kovalenko, Nava Tintarev, Evgeny Pasynkov, Christian Bird, and Alberto Bacchelli. 2018. Does reviewer recommendation help developers? *IEEE Trans. Softw. Eng.* (2018).

[159] Andrey Krutauz, Tapajit Dey, Peter C. Rigby, and Audris Mockus. 2020. Do code review measures explain the incidence of post-release defects? *Empir. Softw. Eng.* 25, 5 (2020), 3323–3356.

[160] Harsh Lal and Gaurav Pahwa. 2017. Code review analysis of software system using machine learning techniques. In *Proceedings of the 11th International Conference on Intelligent Systems and Control (ISCO'17)*. IEEE, 8–13.

[161] Samuel Lehtonen and Timo Poranen. 2015. Metrics for Gerrit code review. In *Proceedings of the 14th Symposium on Programming Languages and Software Tools (SPLST'15), CEUR Workshop Proceedings*, Vol. 1525. CEUR-WS.org, 31–45.

[162] Heng-Yi Li, Shu-Ting Shi, Ferdian Thung, Xuan Huo, Bowen Xu, Ming Li, and David Lo. 2019. DeepReview: Automatic code review using deep multi-instance learning. In *Advances in Knowledge Discovery and Data Mining*, Qiang Yang, Zhi-Hua Zhou, Zhiguo Gong, Min-Ling Zhang, and Sheng-Jun Huang (Eds.). Springer International Publishing, Cham, 318–330.

[163] Zhixing Li, Yue Yu, Gang Yin, Tao Wang, Qiang Fan, and Huaimin Wang. 2017. Automatic classification of review comments in pull-based development model. In *Proceedings of the International Conference on Software Engineering and Knowledge Engineering (SEKE'17)*. 572–577.

[164] Zhi-Xing Li, Yue Yu, Gang Yin, Tao Wang, and Huai-Min Wang. 2017. What are they talking about? Analyzing code reviews in pull-based development model. *J. Comput. Sci. Technol.* 32, 6 (2017), 1060–1075.

[165] J. Liang and O. Mizuno. 2011. Analyzing involvements of reviewers through mining a code review repository. In *Proceedings of the Joint Conference of the 21st International Workshop on Software Measurement and the 6th International Conference on Software Process and Product Measurement*. 126–132.

[166] Zhifang Liao, Yanbing Li, Dayu He, Jinsong Wu, Yan Zhang, and Xiaoping Fan. 2017. Topic-based integrator matching for pull request. In *Proceedings of the Global Communications Conference*. IEEE, 1–6.

[167] Jakub Lipcak and Bruno Rossi. 2018. A large-scale study on source code reviewer recommendation. In *Proceedings of the 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA'18)*. IEEE, 378–387.

[168] Mingwei Liu, Xin Peng, Andrian Marcus, Christoph Treude, Xuefang Bai, Gang Lyu, Jiazhan Xie, and Xiaoxin Zhang. 2021. Learning-based extraction of first-order logic representations of API directives. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 491–502.

[169] L. MacLeod, M. Greiler, M. Storey, C. Bird, and J. Czerwonka. 2018. Code reviewing in the trenches: Challenges and best practices. *IEEE Softw.* 35, 4 (2018), 34–42.

[170] Michał Madera and Rafał Tomoń. 2017. A case study on machine learning model for code review expert system in software engineering. In *Proceedings of the Federated Conference on Computer Science and Information Systems (FedCSIS'17)*. IEEE, 1357–1363.

[171] Mika V. Mäntylä and Casper Lassenius. 2008. What types of defects are really discovered in code reviews? *IEEE Trans. Softw. Eng.* 35, 3 (2008), 430–448.

[172] Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E. Hassan. 2016. An empirical study of the impact of modern code review practices on software quality. *Empir. Softw. Eng.* 21, 5 (2016), 2146–2189.

[173] Massimiliano Menarini, Yan Yan, and William G. Griswold. 2017. Semantics-assisted code review: An efficient tool chain and a user study. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE'17)*. IEEE, 554–565.

[174] Andrew Meneely, Alberto C. Rodriguez Tejeda, Brian Spates, Shannon Trudeau, Danielle Neuberger, Katherine Whitlock, Christopher Ketant, and Kayla Davis. 2014. An empirical investigation of socio-technical code review metrics and security vulnerabilities. In *Proceedings of the 6th International Workshop on Social Software Engineering*. ACM, 37–44.

[175] Benjamin S. Meyers, Nuthan Munaiah, Emily Prud'hommeaux, Andrew Meneely, Josephine Wolff, Cecilia Ovesdotter Alm, and Pradeep Murukannaiah. 2018. A dataset for identifying actionable feedback in collaborative software development. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*. Association for Computational Linguistics, 126–131. https://doi.org/10.18653/v1/P18-2021

[176] Ehsan Mirsaeedi and Peter C. Rigby. 2020. Mitigating turnover with code review recommendation: Balancing expertise, workload, and knowledge distribution. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE'20)*. Association for Computing Machinery, New York, NY, 1183–1195. https://doi.org/10.1145/3377811.3380335

[177] Rahul Mishra and Ashish Sureka. 2014. Mining peer code review system for computing effort and contribution metrics for patch reviewers. In *Proceedings of the IEEE 4th Workshop on Mining Unstructured Data (MUD'14)*. IEEE, 11–15.

[178] Rodrigo Morales, Shane McIntosh, and Foutse Khomh. 2015. Do code review practices impact design quality? A case study of the qt, vtk, and itk projects. In *Proceedings of the 22nd International Conference on Software Analysis, Evolution and Reengineering (SANER'15)*. IEEE, 171–180.

[179] Sebastian Müller, Michael Würsch, Thomas Fritz, and Harald C. Gall. 2012. An approach for collaborative code reviews using multi-touch technology. In *Proceedings of the 5th International Workshop on Co-operative and Human Aspects of Software Engineering*. IEEE, 93–99.

[180] Nuthan Munaiah, Benjamin S. Meyers, Cecilia O. Alm, Andrew Meneely, Pradeep K. Murukannaiah, Emily Prud'hommeaux, Josephine Wolff, and Yang Yu. 2017. Natural language insights from code reviews that missed a vulnerability. In *International Symposium on Engineering Secure Software and Systems*. Springer, 70–86.

[181] Yukasa Murakami, Masateru Tsunoda, and Hidetake Uwano. 2017. WAP: Does reviewer age affect code review performance? In *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE'17)*. IEEE, 164–169.

[182] Emerson Murphy-Hill, Jillian Dicker, Margaret Morrow Hodges, Carolyn D. Egelman, Ciera Jaspan, Lan Cheng, Elizabeth Kammer, Ben Holtz, Matt Jorde, Andrea Knight, and Collin Green. 2021. Engineering impacts of anonymous author code review: A field experiment. *IEEE Trans. Softw. Eng.* (2021), 1–1. https://doi.org/10.1109/TSE.2021.3061527

[183] Reza Nadri, Gema Rodriguez-Perez, and Meiyappan Nagappan. 2021. Insights into nonmerged pull requests in GitHub: Is there evidence of bias based on perceptible race? *IEEE Softw.* 38, 2 (2021), 51–57.

[184] Aziz Nanthaamornphong and Apatta Chaisutanon. 2016. Empirical evaluation of code smells in open source projects: Preliminary results. In *Proceedings of the 1st International Workshop on Software Refactoring*. ACM, 5–8.

[185] Takuto Norikane, Akinori Ihara, and Kenichi Matsumoto. 2018. Do review feedbacks influence to a contributor's time spent on OSS projects? In *Proceedings of the International Conference on Big Data, Cloud Computing, Data Science & Engineering (BCD'18)*. IEEE, 109–113.

[186] Sebastiaan Oosterwaal, Arie van Deursen, Roberta Coelho, Anand Ashok Sawant, and Alberto Bacchelli. 2016. Visualizing code and coverage changes for code review. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 1038–1041.

[187] Ali Ouni, Raula Gaikovina Kula, and Katsuro Inoue. 2016. Search-based peer reviewers recommendation in modern code review. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME'16)*. IEEE, 367–377.

[188] Matheus Paixao, Jens Krinke, Donggyun Han, and Mark Harman. 2018. CROP: Linking code reviews to source code changes. In *Proceedings of the 15th International Conference on Mining Software Repositories*. 46–49.

[189] Matheus Paixao, Jens Krinke, DongGyun Han, Chaiyong Ragkhitwetsagul, and Mark Harman. 2019. The impact of code review on architectural changes. *IEEE Trans. Softw. Eng.* 47, 5 (2019), 1041–1059.

[190] Matheus Paixao and Paulo Henrique Maia. 2019. Rebasing in code review considered harmful: A large-scale empirical investigation. In *Proceedings of the 19th International Working Conference on Source Code Analysis and Manipulation (SCAM'19)*. IEEE, 45–55.

[191] Matheus Paixão, Anderson Uchôa, Ana Carla Bibiano, Daniel Oliveira, Alessandro Garcia, Jens Krinke, and Emilio Arvonio. 2020. Behind the intents: An in-depth empirical study on software refactoring in modern code review. In *Proceedings of the 17th International Conference on Mining Software Repositories*. 125–136.

[192] Thai Pangsakulyanont, Patanamon Thongtanunam, Daniel Port, and Hajimu Iida. 2014. Assessing MCR discussion usefulness using semantic similarity. In *Proceedings of the 6th International Workshop on Empirical Software Engineering in Practice*. IEEE, 49–54.

[193] Sebastiano Panichella, Venera Arnaoudova, Massimiliano Di Penta, and Giuliano Antoniol. 2015. Would static analysis tools help developers with code reviews? In *Proceedings of the 22nd International Conference on Software Analysis, Evolution and Reengineering (SANER'15)*. IEEE, 161–170.

[194] Sebastiano Panichella and Nik Zaugg. 2020. An empirical investigation of relevant changes and automation needs in modern code review. *Empir. Softw. Eng.* 25, 6 (2020), 4833–4872.

[195] Luca Pascarella, Davide Spadini, Fabio Palomba, Magiel Bruntink, and Alberto Bacchelli. 2018. Information needs in contemporary code review. *Proc. ACM Hum.-Comput. Interact.* 2, CSCW (2018), 135.

[196] Rajshakhar Paul, Amiangshu Bosu, and Kazi Zakia Sultana. 2019. Expressions of sentiments during code reviews: Male vs. female. In *Proceedings of the IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER'19)*. IEEE, 26–37.

[197] Rajshakhar Paul, Asif Kamal Turzo, and Amiangshu Bosu. 2021. Why security defects go unnoticed during code reviews? A case-control study of the chromium os project. In *Proceedings of the IEEE/ACM 43rd International Conference on Software Engineering (ICSE'21)*. IEEE, 1373–1385.

[198] Zhenhui Peng, Jeehoon Yoo, Meng Xia, Sunghun Kim, and Xiaojuan Ma. 2018. Exploring how software developers work with mention bot in GitHub. In *Proceedings of the 6th International Symposium of Chinese CHI*. ACM, 152–155.

[199] Gustavo Pinto, Luiz Felipe Dias, and Igor Steinmacher. 2018. Who gets a patch accepted first? Comparing the contributions of employees and volunteers. In *Proceedings of the IEEE/ACM 11th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE'18)*. IEEE, 110–113.

[200] Felix Raab. 2011. Collaborative code reviews on interactive surfaces. In *Proceedings of the 29th Annual European Conference on Cognitive Ergonomics*. ACM, 263–264.

[201] Janani Raghunathan, Lifei Liu, and Huzefa Hatimbhai Kagdi. 2001. Feedback topics in modern code review: Automatic identification and impact on changes 37, 3 (2001), 324–342

[202] Giuliano Ragusa and Henrique Henriques. 2018. Code review tool for visual programming languages. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'18)*. IEEE, 287–288.

[203] M. M. Rahman and C. K. Roy. 2017. Impact of continuous integration on code reviews. In *Proceedings of the IEEE/ACM 14th International Conference on Mining Software Repositories (MSR'17)*. 499–502. https://doi.org/10.1109/MSR.2017.39

[204] Mohammad Masudur Rahman, Chanchal K. Roy, and Jason A. Collins. 2016. CoRReCT: Code reviewer recommendation in GitHub based on cross-project and technology experience. In *Proceedings of the International Conference on Software Engineering Companion (ICSE-C'16)*. IEEE, 222–231.

[205] Mohammad Masudur Rahman, Chanchal K. Roy, and Raula G. Kula. 2017. Predicting usefulness of code review comments using textual features and developer experience. In *Proceedings of the IEEE/ACM 14th International Conference on Mining Software Repositories (MSR'17)*. IEEE, 215–226.

[206] Achyudh Ram, Anand Ashok Sawant, Marco Castelluccio, and Alberto Bacchelli. 2018. What makes a code change easier to review: An empirical investigation on code change reviewability. In *Proceedings of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 201–212.

[207] Soumaya Rebai, Abderrahmen Amich, Somayeh Molaei, Marouane Kessentini, and Rick Kazman. 2020. Multi-objective code reviewer recommendations: Balancing expertise, availability and collaborations. *Autom. Softw. Eng.* 27, 3–4 (December 2020), 301–328. https://doi.org/10.1007/s10515-020-00275-6

[208] Peter Rigby, Brendan Cleary, Frederic Painchaud, Margaret-Anne Storey, and Daniel German. 2012. Contemporary peer review in action: Lessons from open source development. *IEEE Softw.* 29, 6 (November 2012), 56–61.

[209] Peter C. Rigby and Christian Bird. 2013. Convergent contemporary software peer review practices. In *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE'13)*. ACM, New York, NY, 202–212.

[210] Shade Ruangwan, Patanamon Thongtanunam, Akinori Ihara, and Kenichi Matsumoto. 2018. The impact of human factors on the participation decision of reviewers in modern code review. *Emp. Softw. Eng.* (2018), 1–44.

[211] Nafiz Sadman, Md Manjurul Ahsan, and M. A. Parvez Mahmud. 2020. ADCR: An adaptive TOOL to select "appropriate developer for code review" based on code context. In *Proceedings of the 11th IEEE Annual Ubiquitous Computing, Electronics Mobile Communication Conference (UEMCON'20)*. 0583–0591. https://doi.org/10.1109/UEMCON51285.2020.9298102

[212] Caitlin Sadowski, Emma Söderberg, Luke Church, Michal Sipko, and Alberto Bacchelli. 2018. Modern code review: A case study at Google. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP'18)*. ACM, New York, NY, 181–190.

[213] Nishrith Saini and Ricardo Britto. 2021. *Using Machine Intelligence to Prioritise Code Review Requests*. IEEE Press, 11–20. https://doi.org/10.1109/ICSE-SEIP52600.2021.00010

[214] Ronie Salgado and Alexandre Bergel. 2017. Pharo Git thermite: A visual tool for deciding to weld a pull request. In *Proceedings of the 12th Edition of the International Workshop on Smalltalk Technologies*. ACM, 1–6.

[215] Mateus Santos, Josemar Caetano, Johnatan Oliveira, and Humberto T. Marques-Neto. 2018. Analyzing the impact of feedback in GitHub on the software developer's mood. In *Proceedings of the International Conference on Software Engineering & Knowledge Engineering*. ACM.

[216] Bonita Sharif, Michael Falcone, and Jonathan I. Maletic. 2012. An eye-tracking study on the role of scan time in finding source code defects. In *Proceedings of the Symposium on Eye Tracking Research and Applications (ETRA'12)*. ACM, New York, NY, 381–384.

[217] Shipra Sharma and Balwinder Sodhi. 2019. Using stack overflow content to assist in code review. *Softw.: Pract. Exp.* 49, 8 (2019), 1255–1277.

[218] Shu-Ting Shi, Ming Li, David Lo, Ferdian Thung, and Xuan Huo. 2019. Automatic code review by learning the revision of source code. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 4910–4917. https://doi.org/10.1609/aaai.v33i01.33014910

[219] Junji Shimagaki, Yasutaka Kamei, Shane McIntosh, Ahmed E. Hassan, and Naoyasu Ubayashi. 2016. A study of the quality-impacting practices of modern code review at sony mobile. In *Proceedings of the International Conference on Software Engineering Companion (ICSE-C'16)*. IEEE, 212–221.

[220] Moran Shochat, Orna Raz, and Eitan Farchi. 2008. SeeCode—A code review plug-in for eclipse. In *Haifa Verification Conference*. Springer, 205–209.

[221] Devarshi Singh, Varun Ramachandra Sekar, Kathryn T. Stolee, and Brittany Johnson. 2017. Evaluating how static analysis tools can reduce code review effort. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'17)*. IEEE, 101–105.

[222] J. Siow, C. Gao, L. Fan, S. Chen, and Y. Liu. 2020. CORE: Automating review recommendation for code changes. In *Proceedings of the IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER'20)*. IEEE Computer Society, Los Alamitos, CA, 284–295. https://doi.org/10.1109/SANER48275.2020.9054794

[223] Daricélio M. Soares, Manoel L. de Lima Júnior, Alexandre Plastino, and Leonardo Murta. 2018. What factors influence the reviewer assignment to pull requests? *Inf. Softw. Technol.* 98 (2018), 32–43.

[224] Davide Spadini, Maurício Aniche, Margaret-Anne Storey, Magiel Bruntink, and Alberto Bacchelli. 2018. When testing meets code review: Why and how developers review tests. In *Proceedings of the 40th International Conference on Software Engineering (ICSE'18)*. ACM, New York, NY, 677–687.

[225] Davide Spadini, Gül Çalikli, and Alberto Bacchelli. 2020. Primers or reminders? The effects of existing review comments on code review. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering (ICSE'20)*. IEEE, 1171–1182.

[226] Davide Spadini, Fabio Palomba, Tobias Baum, Stefan Hanenberg, Magiel Bruntink, and Alberto Bacchelli. 2019. Test-driven code review: An empirical study. In *Proceedings of the IEEE/ACM 41st International Conference on Software Engineering (ICSE'19)*. IEEE, 1061–1072.

[227] Kai Spohrer, Thomas Kude, Armin Heinzl, and Christoph Schmidt. 2013. Peer-based quality assurance in information systems development: A transactive memory perspective. In *Proceedings of the International Conference on Information Systems, (ICIS'13, Milano, Italy, December 15-18, 2013)*.

[228] Panyawut Sri-iesaranusorn, Raula Gaikovina Kula, and Takashi Ishio. 2021. Does code review promote conformance? A study of OpenStack patches. In *Proceedings of the IEEE/ACM 18th International Conference on Mining Software Repositories (MSR'21)*. IEEE, 444–448.

[229] Miroslaw Staron, Mirosław Ochodek, Wilhelm Meding, and Ola Söder. 2020. Using machine learning to identify code fragments for manual review. In *Proceedings of the 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA'20)*. IEEE, 513–516.

[230] Anton Strand, Markus Gunnarson, Ricardo Britto, and Muhmmad Usman. 2020. Using a context-aware approach to recommend code reviewers: Findings from an industrial case study. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice*. ACM, 1–10.

[231] Emre Sülün, Eray Tüzün, and Uğur Doğrusöz. 2019. Reviewer recommendation using software artifact traceability graphs. In *Proceedings of the Fifteenth International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE'19)*. Association for Computing Machinery, New York, NY, 66–75. https://doi.org/10.1145/3345629.3345637

[232] Andrew Sutherland and Gina Venolia. 2009. Can peer code reviews be exploited for later information needs? In *Proceedings of the 31st International Conference on Software Engineering-Companion*. IEEE, 259–262.

[233] Rajendran Swamidurai, Brad Dennis, and Uma Kannan. 2014. Investigating the impact of peer code review and pair programming on test-driven development. In *Proceedings of the IEEE SoutheastCon (SoutheastCon'14)*. IEEE, 1–5.

[234] Yida Tao and Sunghun Kim. 2015. Partitioning composite code changes to facilitate code review. In *Proceedings of the 12th Working Conference on Mining Software Repositories*. IEEE, 180–190.

[235] K. Ayberk Tecimer, Eray Tüzün, Hamdi Dibeklioglu, and Hakan Erdogmus. 2021. Detection and elimination of systematic labeling bias in code reviewer recommendation systems. In *Evaluation and Assessment in Software Engineering (EASE'21)*. Association for Computing Machinery, New York, NY, 181–190. https://doi.org/10.1145/3463274.3463336

[236] Christopher Thompson and David Wagner. 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*. 83–92.

[237] Patanamon Thongtanunam and Ahmed E. Hassan. 2020. Review dynamics and their impact on software quality. *IEEE Trans. Softw. Eng.* 47, 12 (2020), 2698–2712.

[238] Patanamon Thongtanunam, Raula Gaikovina Kula, Ana Erika Camargo Cruz, Norihiro Yoshida, and Hajimu Iida. 2014. Improving code review effectiveness through reviewer recommendations. In *Proceedings of the 7th International Workshop on Cooperative and Human Aspects of Software Engineering*. ACM, 119–122.

[239] Patanamon Thongtanunam, Shane McIntosh, Ahmed E. Hassan, and Hajimu Iida. 2015. Investigating code review practices in defective files: An empirical study of the qt system. In *Proceedings of the 12th Working Conference on Mining Software Repositories*. IEEE, 168–179.

[240] Patanamon Thongtanunam, Shane McIntosh, Ahmed E. Hassan, and Hajimu Iida. 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 (ICSE'16)*. ACM, New York, NY, 1039–1050.

[241] Patanamon Thongtanunam, Shane McIntosh, Ahmed E. Hassan, and Hajimu Iida. 2018. Review participation in modern code review: An empirical study of the Android, Qt, and OpenStack projects (journal-first abstract). In *Proceedings of the 25th International Conference on Software Analysis, Evolution and Reengineering (SANER'18)*. IEEE, 475–475.

[242] Patanamon Thongtanunam, Chakkrit Tantithamthavorn, Raula Gaikovina Kula, Norihiro Yoshida, Hajimu Iida, and Ken-ichi Matsumoto. 2015. Who should review my code? A file location-based code-reviewer recommendation approach for modern code review. In *Proceedings of the International Conference on Software Analysis, Evolution and Reengineering (SANER'15)*. IEEE, 141–150.

[243] Patanamon Thongtanunam, Xin Yang, Norihiro Yoshida, Raula Gaikovina Kula, Ana Erika Camargo Cruz, Kenji Fujiwara, and Hajimu Iida. 2014. Reda: A web-based visualization tool for analyzing modern code review dataset. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME'14)*. IEEE, 605–608.

[244] Rosalia Tufano, Luca Pascarella, Michele Tufano, Denys Poshyvanyk, and Gabriele Bavota. 2021. Towards automating code review activities. In *Proceedings of the IEEE/ACM 43rd International Conference on Software Engineering (ICSE'21)*. 163–174. https://doi.org/10.1109/ICSE43902.2021.00027

[245] Yuriy Tymchuk, Andrea Mocci, and Michele Lanza. 2015. Code review: Veni, vidi, vici. In *Proceedings of the IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER'15)*. IEEE, 151–160.

[246] Anderson Uchôa, Caio Barbosa, Daniel Coutinho, Willian Oizumi, Wesley K. G. Assunçao, Silvia Regina Vergilio, Juliana Alves Pereira, Anderson Oliveira, and Alessandro Garcia. 2021. Predicting design impactful changes in modern code review: A large-scale empirical study. In *Proceedings of the IEEE/ACM 18th International Conference on Mining Software Repositories (MSR'21)*. IEEE, 471–482.

[247] Anderson Uchôa, Caio Barbosa, Willian Oizumi, Publio Blenílio, Rafael Lima, Alessandro Garcia, and Carla Bezerra. 2020. How does modern code review impact software design degradation? An in-depth empirical study. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME'20)*. IEEE, 511–522.

[248] Yuki Ueda, Akinori Ihara, Takashi Ishio, Toshiki Hirao, and Kenichi Matsumoto. 2018. How are IF-conditional statements fixed through peer CodeReview? *IEICE Trans. Inf. Syst.* 101, 11 (2018), 2720–2729.

[249] Yuki Ueda, Akinori Ihara, Takashi Ishio, and Kennichi Matsumoto. 2018. Impact of coding style checker on code review-a case study on the openstack projects. In *Proceedings of the 9th International Workshop on Empirical Software Engineering in Practice (IWESEP'18)*. IEEE, 31–36.

[250] Yuki Ueda, Takashi Ishio, Akinori Ihara, and Kenichi Matsumoto. 2019. Mining source code improvement patterns from similar code review works. In *Proceedings of the IEEE 13th International Workshop on Software Clones (IWSC'19)*. IEEE, 13–19.

[251] Naomi Unkelos-Shpigel and Irit Hadar. 2016. Lets make it fun: Gamifying and formalizing code review. In *Proceedings of the 11th International Conference on Evaluation of Novel Software Approaches to Software Engineering*. SCITEPRESS-Science and Technology Publications, Lda, 391–395.

[252] Hidetake Uwano, Masahide Nakamura, Akito Monden, and Ken-ichi Matsumoto. 2007. Exploiting eye movements for evaluating reviewer's performance in software review. *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.* E90-A, 10 (October 2007), 2290–2300.

[253] Erik Van Der Veen, Georgios Gousios, and Andy Zaidman. 2015. Automatically prioritizing pull requests. In *Proceedings of the IEEE/ACM 12th Working Conference on Mining Software Repositories*. IEEE, 357–361.

[254] P. van Wesel, B. Lin, G. Robles, and A. Serebrenik. 2017. Reviewing career paths of the OpenStack developers. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME'17)*. 544–548.

[255] Giovanni Viviani and Gail C. Murphy. 2016. Removing stagnation from modern code review. In *Companion Proceedings of the ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity*. ACM, 43–44.

[256] Hana Vrzakova, Andrew Begel, Lauri Mehtätalo, and Roman Bednarik. 2020. Affect recognition in code review: An in-situ biometric study of reviewer's affect. *J. Syst. Softw.* 159 (2020), 110434.
[257] Chen Wang, Xiaoyuan Xie, Peng Liang, and Jifeng Xuan. 2017. Multi-perspective visualization to assist code change review. In *Proceedings of the 24th Asia-Pacific Software Engineering Conference (APSEC'17)*. IEEE, 564–569.
[258] Dong Wang, Raula Gaikovina Kula, Takashi Ishio, and Kenichi Matsumoto. 2021. Automatic patch linkage detection in code review using textual content and file location features. *Inf. Softw. Technol.* 139 (2021), 106637.
[259] Dong Wang, Tao Xiao, Patanamon Thongtanunam, Raula Gaikovina Kula, and Kenichi Matsumoto. 2021. Understanding shared links and their intentions to meet information needs in modern code review. *Empir. Softw. Eng.* 26, 5 (2021), 1–32.
[260] Min Wang, Zeqi Lin, Yanzhen Zou, and Bing Xie. 2019. Cora: Decomposing and describing tangled code changes for reviewer. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE'19)*. IEEE, 1050–1061.
[261] Qingye Wang, Bowen Xu, Xin Xia, Ting Wang, and Shanping Li. 2019. Duplicate pull request detection: When time matters. In *Proceedings of the 11th Asia-Pacific Symposium on Internetware (Internetware'19)*. Association for Computing Machinery, New York, NY. https://doi.org/10.1145/3361242.3361254
[262] Song Wang, Chetan Bansal, Nachiappan Nagappan, and Adithya Abraham Philip. 2019. Leveraging change intents for characterizing and identifying large-review-effort changes. In *Proceedings of the 15th International Conference on Predictive Models and Data Analytics in Software Engineering*. 46–55.
[263] Yanqing Wang, Xiaolei Wang, Yu Jiang, Yaowen Liang, and Ying Liu. 2016. A code reviewer assignment model incorporating the competence differences and participant preferences. *Found. Comput. Decis. Sci.* 41, 1 (2016), 77–91.
[264] Ruiyin Wen, David Gilbert, Michael G. Roche, and Shane McIntosh. 2018. BLIMP tracer: Integrating build impact analysis with code review. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME'18)*. IEEE, 685–694.
[265] Mairieli Wessel, Alexander Serebrenik, Igor Wiese, Igor Steinmacher, and Marco A. Gerosa. 2020. Effects of adopting code review bots on pull requests to OSS projects. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME'20)*. IEEE, 1–11.
[266] Mairieli Wessel, Alexander Serebrenik, Igor Wiese, Igor Steinmacher, and Marco A. Gerosa. 2020. What to expect from code review bots on GitHub? A survey with OSS maintainers. In *Proceedings of the 34th Brazilian Symposium on Software Engineering (SBES'20)*. Association for Computing Machinery, New York, NY, 457–462. https://doi.org/10.1145/3422392.3422459
[267] Mairieli Wessel, Igor Wiese, Igor Steinmacher, and Marco Aurelio Gerosa. 2021. Don't disturb me: Challenges of interacting with software bots on open source software projects. *Proc. ACM Hum.-Comput. Interact.* 5, CSCW2 (2021), 1–21.
[268] Xin Xia, David Lo, Xinyu Wang, and Xiaohu Yang. 2015. Who should review this change?: Putting text and file location analyses together for more accurate recommendations. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME'15)*. IEEE, 261–270.
[269] Zhenglin Xia, Hailong Sun, Jing Jiang, Xu Wang, and Xudong Liu. 2017. A hybrid approach to code reviewer recommendation with collaborative filtering. In *Proceedings of the 6th International Workshop on Software Mining (SoftwareMining'17)*. IEEE, 24–31.
[270] Cheng Yang, Xunhui Zhang, Lingbin Zeng, Qiang Fan, Gang Yin, and Huaimin Wang. 2017. An empirical study of reviewer recommendation in pull-based development model. In *Proceedings of the 9th Asia-Pacific Symposium on Internetware*. ACM, 14.
[271] Cheng Yang, Xun-hui Zhang, Ling-bin Zeng, Qiang Fan, Tao Wang, Yue Yu, Gang Yin, and Huai-min Wang. 2018. RevRec: A two-layer reviewer recommendation algorithm in pull-based development model. *J. Centr. South Univ.* 25, 5 (2018), 1129–1143.
[272] Xin Yang. 2014. Social network analysis in open source software peer review. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'14)*. ACM, New York, NY, 820–822.
[273] Xin Ye. 2019. Learning to rank reviewers for pull requests. *IEEE Access* 7 (2019), 85382–85391. https://doi.org/10.1109/ACCESS.2019.2925560
[274] Xin Ye, Yongjie Zheng, Wajdi Aljedaani, and Mohamed Wiem Mkaouer. 2021. Recommending pull request reviewers based on code changes. *Soft Comput.* 25, 7 (April 2021), 5619–5632. https://doi.org/10.1007/s00500-020-05559-3
[275] Haochao Ying, Liang Chen, Tingting Liang, and Jian Wu. 2016. EARec: Leveraging expertise and authority for pull-request reviewer recommendation in GitHub. In *Proceedings of the 3rd International Workshop on CrowdSourcing in Software Engineering*. ACM, 29–35.
[276] Yue Yu, Huaimin Wang, Gang Yin, and Charles X. Ling. 2014. Reviewer recommender of pull-requests in GitHub. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME'14)*. IEEE, 609–612.

[277] Fiorella Zampetti, Gabriele Bavota, Gerardo Canfora, and Massimiliano Di Penta. 2019. A study on the interplay between pull request review and continuous integration builds. In *Proceedings of the IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER'19)*. IEEE, 38–48.

[278] Farida El Zanaty, Toshiki Hirao, Shane McIntosh, Akinori Ihara, and Kenichi Matsumoto. 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*. ACM, 1–10.

[279] Motahareh Bahrami Zanjani, Huzefa Kagdi, and Christian Bird. 2016. Automatically recommending peer reviewers in modern code review. *IEEE Trans. Softw. Eng.* 42, 6 (2016), 530–543.

[280] Tianyi Zhang, Myoungkyu Song, Joseph Pinedo, and Miryung Kim. 2015. Interactive code review for systematic changes. In *Proceedings of the IEEE/ACM 37th International Conference on Software Engineering*, Vol. 1. IEEE, 111–122.

[281] Weifeng Zhang, Zhen Pan, and Ziyuan Wang. 2020. Prediction method of code review time based on hidden markov model. In *Web Information Systems and Applications*, Guojun Wang, Xuemin Lin, James Hendler, Wei Song, Zhuoming Xu, and Genggeng Liu (Eds.). Springer International Publishing, Cham, 168–175.

[282] Xuesong Zhang, Bradley Dorn, William Jester, Jason Van Pelt, Guillermo Gaeta, and Daniel Firpo. 2011. Design and implementation of Java sniper: A community-based software code review web solution. In *Proceedings of the 44th Hawaii International Conference on System Sciences*. IEEE, 1–10.

[283] Y. Zhang, G. Yin, Y. Yu, and H. Wang. 2014. A exploratory study of @-mention in GitHub's pull-requests. In *Proceedings of the 21st Asia-Pacific Software Engineering Conference*, Vol. 1. 343–350.

[284] Guoliang Zhao, Daniel Alencar da Costa, and Ying Zou. 2019. Improving the pull requests review process using learning-to-rank algorithms. *Emp. Softw. Eng.* (2019), 1–31.

[285] Zhifang Liao, ZeXuan Wu, Yanbing Li, Yan Zhang, Xiaoping Fan, and Jinsong Wu. 2020. Core-reviewer recommendation based on pull request topic model and collaborator social network. *Soft Comput.* 24 (2020), 5683–5693.

Received 4 October 2021; revised 21 December 2022; accepted 31 January 2023