Noname manuscript No.
(will be inserted by the editor)

An Empirical Investigation of Relevant Changes and Automation Needs in Modern Code Review

Sebastiano Panichella and Nik Zaugg

Received: date / Accepted: date

Abstract Recent research has shown that available tools for Modern Code Review (MCR) are still far from meeting the current expectations of developers. The objective of this paper is to investigate the approaches and tools that, from a *developer's point of view*, are still needed to facilitate MCR activities. To that end, we first empirically elicited a taxonomy of recurrent review change types that characterize MCR. The taxonomy was designed by performing three steps: (i) we generated an initial version of the taxonomy by qualitatively and quantitatively analyzing 211 review changes/commits and 648 review comments of ten open-source projects; then (ii) we integrated into this initial taxonomy, topics, and MCR change types of an existing taxonomy available from the literature; finally, (iii) we surveyed 52 developers to integrate eventually missing change types in the taxonomy. Results of our study highlight that the availability of new emerging development technologies (e.g., cloud-based technologies) and practices (e.g., continuous delivery) has pushed developers to perform additional activities during MCR and that additional types of feedback are expected by reviewers. Our participants provided recommendations, specified techniques to employ, and highlighted the data to analyze for building recommender systems able to automate the code review activities composing our taxonomy. We surveyed 14 additional participants (12 developers and 2 researchers), not involved in the previous survey, to qualitatively assess the relevance and completeness of the identified MCR change types as well as assess how critical and feasible to implement are some of the identified techniques to support MCR activities. Thus, with a study involving 21 additional developers, we qualitatively assess the feasibility and usefulness

Sebastiano Panichella
Zurich University of Applied Science, Switzerland
E-mail: panc@zhaw.ch

Nik Zaugg
University of Zurich, Switzerland
E-mail: nik.zaugg@bf.uzh.ch

of leveraging natural language feedback (automation considered critical/feasible to implement) in supporting developers during MCR activities. In summary, this study sheds some more light on the approaches and tools that are still needed to facilitate MCR activities, confirming the feasibility and usefulness of using summarization techniques during MCR activities. We believe that the results of our work represent an essential step for meeting the expectations of developers and supporting the vision of full or partial automation in MCR.

Keywords Code Review Process and Practices, Empirical Study, Automated Software Engineering.

1 Introduction

Modern Code Review (MCR) [14] represents a variant of the traditional code review (CR) process, whose main characteristic is to be informal and supported by tools. Nowadays, MCR is a widely applied practice in both open-source and industrial systems [14], and recent work empirically investigated its process outcomes [52,58], its available tools [14,24], or proposed solutions to automate some of its activities [16,26,71,85,96].

During MCR, developers are usually interested in improving the quality of submitted patches [22,78] by fixing bugs [58], adhering to conventions/coding styles or by making the source code easier to be maintained [15,77], to meet user expectations [40,72,98]. In this process, a developer, author of the code under review, asks other developers (i.e., the reviewers) to inspect her/his code. In this context, studies performed in the past demonstrated that inspections are also useful for improving the quality of further artifacts (different from the testing and production code [83]) such as requirements [37,75] and design [74].

MCR is generally supported by tools aiding developers during various activities. For example, the *Gerrit* [10] tool is widely used by open-source projects to support the management of the MCR process, while *CheckStyle* [3] and *PMD* [13] are popular tools used for detecting defects (e.g., vulnerabilities [29]) and design issues (e.g., the high coupling between objects) in the code under review.

Recent research produced further tools to support, in different ways, decisions and actions of MCR: recommender systems (i) selecting appropriate peer reviewers to evaluate a given patch [15,67,95]; and approaches to automatically (ii) decompose code review change-sets [16], recommending the files to focus on during a review [17], or to simply detect potential mistakes [96]. However, according to recent research [14,83], outcomes of available tools and prototypes are still far from meeting the current expectations of developers in modern code review [14,71,83].

The *objective* of this paper is to investigate the approaches and tools that, from a *developer point of view*, are still needed to facilitate MCR activities (in the introduction, we refer to CR, but *the whole work concern MCR challenges*). To the best of our knowledge, very few studies investigated at

the same time (i) the most recurrent or critical *code review changes* (later referred to as *code review change types*) developers have to deal with and (ii) the approaches and/or *tools that are still needed* to automate or accommodate such changes. Indeed, while previous studies mainly investigated the usage and/or the limits of existing tools for code review [14, 22, 71, 83], this paper puts its attention on the specific changes that developers actually perform in code reviews, investigating the potential automation that is needed for supporting such changes. In this context, it is important to clarify that with *code review changes* (or *code review change types*) we explicitly refer to the actual *"changes that developers perform to address the received code review comments"*.

We believe that this investigation has the potential to fill the gap between the current needs of practitioners and the available research tools and prototypes for MCR. To that end, in this paper we address the following research questions:

1. ***RQ*$_1$*: What types of changes occur during MCR?***
   We first empirically elicited a taxonomy of the most critical and recurrent *MCR change types* that characterize reviews and investigated the types of MCR changes that, according to the developers involved in our study, could (or should) be automated. The taxonomy was designed by performing three steps: (i) we generated an initial version of the taxonomy by qualitatively and quantitatively analyzing 211 review changes and 648 review comments of 10 open-source projects; then (ii) we integrated into this initial taxonomy, topics and MCR change types from an existing taxonomy available from the literature; finally, (iii) we surveyed 52 developers to integrate eventual missing MCR change types.
2. ***RQ*$_2$*: What are the emerging automation needs of developers in MCR?*** This research question is a follow-up of the previous one. However, while in **RQ**$_1$ we look at the types of changes that occur during the MCR process, here we investigate the data, approaches and tools that developers would need to accommodate the identified MCR change types. Hence, we asked our survey participants (52 developers) to specify (i) the most critical and/or important review change types they usually perform in MCR; and the (ii) type of automation that they would need (or envision) to accommodate these review change types.

Results of our study highlight that the availability of new emerging development technologies (e.g., cloud-based technologies) and practices (e.g., Continuous Delivery and Continuous Integration) has pushed developers to perform additional activities or tasks during MCR (e.g., the need to fix licensing and security issues). As a consequence, additional types of feedback are expected by reviewers, and novel approaches and tools are needed by developers acting as authors of changed code during code inspection activities and tasks.

Most (98%) surveyed developers believe that (i) certain code review activities or tasks (e.g., defects detection) are difficult to automate, while others (e.g., license header generation) could (or should) be possibly automated by

novel recommender systems for MCR. 96% of our study participants provided insights on the types of approaches and tools they would need in the context of MCR, sharing recommendations, specifying techniques to employ, and highlighting the data to analyze for building recommenders able to automate code review activities. Interestingly, potential automation is needed for example to handle licensing and security issues, or supporting changes in non-source code artifacts (e.g., continuous delivery and integration configuration files, files for runtime configuration, static analysis tools configuration files, etc.).
To complement the results of the previous analysis, we surveyed 14 additional participants (12 developers and 2 researchers), not involved in the aforementioned survey, to qualitatively assess the relevance and completeness of the identified MCR change types as well as assessing how critical and feasible to implement are some of the identified techniques to support MCR activities. This study motivated the usage of specific techniques over others to support MCR.
Among the various proposals, most developers also recommended to implement solutions based on customized approaches leveraging machine learning, Natural Language Parsing (NLP) and data mining techniques modeling the MCR problems with the notion of anti-patterns and change metrics. Hence, in the context of our work (RQ2), we also discuss qualitatively, with a study involving 21 additional developers, the perceived usefulness of leveraging summarization techniques for modeling the MCR problems with the notion of anti-patterns and change metrics. This additional study has the only goal to investigate the feasibility of this research direction, to make an initial concrete step toward semi-automated tools for MCR activities.

Paper contributions. The contributions of this paper can be summarized as follows:

1. a qualitative and quantitative investigation on the types of MCR changes performed by developers, this via repository analysis and a survey involving developers.
2. an empirically elicited **C**ode **R**eview ch**A**nges **M**odel (`CRAM`), i.e., a taxonomy of MCR changes grouped in high- and low-level categories.
3. a qualitative and quantitative investigation on the data, approaches, and tools that, from a *developer's point of view*, are still needed to facilitate MCR activities.
4. finally, we also discuss qualitatively the potential (perceived) feasibility/usefulness of using summarization techniques for modeling the MCR problems with the notion of anti-patterns and change metrics, to support MCR activities.

We believe that this work represents a relevant step toward the definition of tools meeting the emerging expectations of authors and reviewers in modern inspection processes, and thus supporting the vision of full or partial automation in MCR [14, 83]

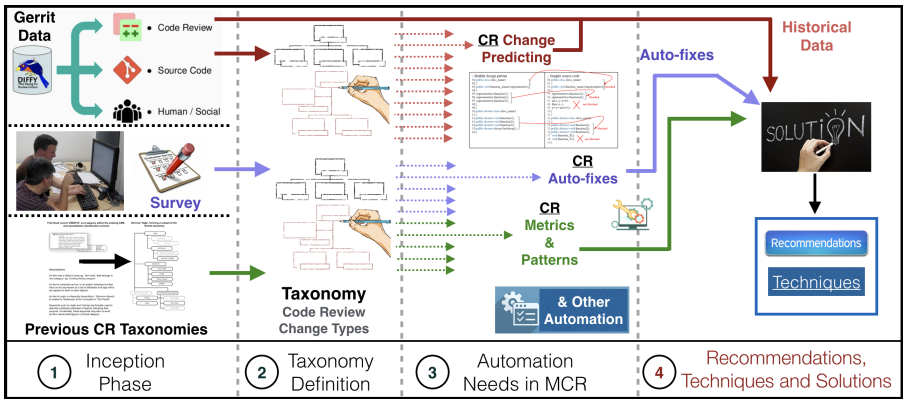

Fig. 1: Overview of Research Approach

Replication package. We make publicly available a replication package[1] with (i) material and working data sets of our study, (ii) complete results of the survey; and (iii) the leveraged raw-data for replication purposes.

Paper structure. Section 2 details the study definition and planning, the data extraction process and the evaluation methodology adopted to answer our research questions. Section 3 discusses the results, while threats to its validity are discussed in Section 4. Section 5 discusses the related literature concerning studies on code review in general and MCR in particular, while Section 6 concludes the paper and outlines directions for future work.

2 Research Methodology

The *goal* of our study is to provide more information on the types of changes developers perform in MCR activities and investigate the approaches and tools that, from a *developer's point of view*, are still needed to facilitate MCR tasks. This section describes the methodology adopted to answer our research questions.

2.1 Approach Overview

Figure 1 depicts the research approach we followed to answer our research questions, which consists of four steps:

1. ***Inception Phase:*** First of all, we performed an inception phase aimed at enriching our knowledge about the studied problem, thus collecting information about the specific MCR change types. We generated an initial taxonomy of MCR change types by (i) analyzing comments and changes

[1] https://doi.org/10.5281/zenodo.3679402

reported by developers during MCR activities of 10 open source projects; and (ii) integrating into this initial taxonomy the change types reported in the validated categorization scheme by Beller *et al.* [22].

2. ***Taxonomy Definition (RQ_1):*** We surveyed 52 developers to validate and get feedback on the taxonomy defined during the inception phase. Specifically, we manually analyzed the feedback gathered from our participants and added, modified and expanded the categories of the initial taxonomy. The output of this phase consisted of the **C**ode **R**eview ch**A**nges **M**odel (`CRAM`), i.e., a taxonomy of MCR changes grouped in high- and low-level categories (described in Table 4). To complement the results of this analysis, we surveyed 14 additional participants (12 developers and 2 researchers), not involved in the aforementioned survey, to qualitatively assess the relevance and completeness of the identified MCR change types.
3. ***Automation Needs in MCR (RQ_2):*** We asked our study participants about the process and practices applied to handle the MCR change types composing `CRAM`, asking at the same time what tools they use or what tools they would need for handling such changes/problems. The output of this phase consisted of the analysis of feedback we received by the participants and the selection of most relevant comments for our investigation (RQ_2).
4. ***Recommendations, Techniques, and Solutions (RQ_2):*** In this phase, we first of all focused our effort on clustering the selected feedback in a formal way, summarizing the most interesting information from participants (RQ_2). Thus, the first output of this phase consisted in the `MCR-Request` (**MCR-RE**commendations, Techni**QUE**s, and **S**olu**T**ions) `model`, i.e., a taxonomy summarizing the current developers' automation needs, the recommendations, the techniques, and the solutions envisioned by developers to automate MCR activities. To complement the results of this analysis, we interviewed 14 additional participants (12 developers and 2 researchers), not involved in the aforementioned survey, to qualitatively assess how critical and feasible to implement are some of the identified techniques to support MCR activities. This study motivated the usage of specific techniques over others to support MCR.

2.2 Inception Phase

Analysis of MCR commits and comments. To gain more understanding on the specific code review change types occurring in MCR we collected the MCR comments and changes from the history of ten Java open source projects, namely Eclipse Acceleo [1], Eclipse CDT [7], Eclipse Amalgam [2], Eclipse BPEL [5], Eclipse Cbi [6], Eclipse EGit [4], Eclipse PDE [8], Egit Training [9] JGit [11], and M2e [12].

The main characteristics of the projects are reported in Table 1. The observed period considered was between 2012 and 2017. There are three reasons that pushed us to select such time window: (i) we wanted to observe the history of projects within a specific time frame, so that the probability that similar

MCR changes and tools used among developers or investigated project was higher; (ii) Observing a reasonable past time window ensure that all review comments are addressed by the authors in MCR; (iii) we wanted to observe at least a period of 5 years for each project (in some cases the projects life was shorter, and in that case we were able to observe/analyze the whole history of them. It is important to mention that the ten projects in Table 1 were mainly chosen by considering the following selection criteria:

- *Projects sample size compared to previous studies*: Compared to our work, Beller *et al.* [22] manually analyzed only two OSS projects. We targeted 10 projects to make our results potentially more generalizable.
- *Availability and diversity*: As first criterion project were selected based on the availability of review information (e.g., evolution of patches stored MCR commits, reviewers comments of patches, amount of reviewers comments > 50 in the history) through Gerrit, and their different domain and size.
- *Projects used in previous studies*: Among the selected projects, we also selected projects that were also considered in previous work (e.g., the one from the Eclipse ecosystems from [71]).

As a first step, we initially selected, among all code review commits (or changes) in our dataset, the ones reporting explicit reviewers' comments on the quality of patches. This step was needed to investigate the actual changes in code reviews that were performed by authors (i.e., developers) of a code change to address the comments of reviewers. Furthermore, we cleaned our dataset; removing review comments contained in abandoned patches. Hence, two authors of this work, by applying grounded theory [90], manually analyzed 648 reviewers' comments related to 211 MCR review commits, focusing on the comments that could be relevant to our study (i.e., the one mentioning the changes to perform to improve the patch code or other artifacts). In doing so, the two coders also verified whether the MCR changes performed by authors of patches actually addressed the reviewers' comments. Thus they shared a spreadsheet to encode, using a short sentence or description, the specific changes proposed by reviewers and implemented in MCR commits. Specifically, while reporting a new sentence, the coders checked whether the sentence matched/fitted a change type reported in sentences previously defined; if not, a new MCR change type was added. In total, 631 reviewers' comments were analyzed and 17 were deemed unhelpful or ambiguous by the authors.

It is important to note that we adopted grounded theory to built the initial taxonomy, acting as if no prior work has built a previous (i.e., we did not know the detail of the taxonomy of Beller *et al.* [22]). This step was needed to investigate the MCR changes that are missing in the taxonomy proposed by Beller *et al.*. Thus, the initial taxonomy was then merged with the one by Beller *et al.*, by adding the new discovered categories/elements (see Section 3.1). This required to perform a further open coding process to make the merge between the two taxonomies.

Table 1: Characteristics of the analyzed projects.

Project	Observed Period	# of review changes	# of reviews comments	# of KLOC
Acceleo	2015-03–2017-03	56	243	622
Amalgam	2015-07–2017-03	4	4	26
Bpel	2012-10–2012-12	1	2	219
Cbi	2015-07–2017-03	1	1	13
Cdt	2012-05–2017-03	70	192	1,600
Egit-github	2012-02–2017-07	25	55	200
Egit-pde	2012-02–2012-03	1	17	531
Egit-training	2012-03–2016-03	3	3	195
JGit	2012-09–2017-03	1	1	212
M2e	2014-03–2017-03	49	130	3
Total	-	**211**	**648**	**3621**

By scrutinizing the aforementioned set of MCR comments and commits the coders identified a total of ***15 initial potential MCR change types***, and logically grouped them in an initial taxonomy of 3 high- and 15 low-level categories of changes. During the validation step, the level of agreement between first and second coders was 82% (disagreements were discussed and fixed). The grouped review commits and comments and this initial version of the taxonomy are shared as one of the appendices of our replication package.

Integration of existing taxonomies. As reported in the related work (Section 5), Beller *et al.* [22] manually analyzed changes taking place in reviewed code from two OSS projects and classified them into *evolvability changes* and *functional changes*, as reported in their validated categorization scheme. To verify the completeness of the initial taxonomy, which emerged via *manual analysis of code review data of the projects* reported in Table 1, we performed a one to one matching between elements in our taxonomy and the one composing the scheme by Beller *et al.* [22]. We observed that some MCR change types composing the initial taxonomy were also present in the one by Beller *et al.*, while others were not. Thus, we split, merged and refactored categories coming from the schema by Beller *et al.* and the previously elicited categories and integrated and combined them into a new taxonomy. Also in this case, this improved version of the taxonomy is available as one of the appendices of our replication package. However, we highlight (with different colors) in Table 4 and Table 5, that report the final taxonomy (obtained by integrating also the feedback received from developers, as explained in Section 2.3), the categories that were present in the scheme by Beller *et al.* and the ones that were not.

2.3 Taxonomy Definition & Automation Needs in MCR

To verify the taxonomy's saturation, i.e., its capability to cover all possible MCR changes, we performed a survey involving 52 developers (we invited more

than 200 participants and around 23% of them participated in the study) to understand (i) whether the taxonomy was considered by them as exhaustive and/or complete (RQ_1); (ii) what type of feedback developers usually receive or expect in code reviews (RQ_1); and (iii) what tools they need or envision to support relevant MCR changes (RQ_2). Our survey was implemented using *Google Forms*[2]. The structure of our questionnaire consisted of 18 questions, which included 6 multiple-choice (MC), and 12 open (O) questions. We decided to have several open questions in the survey to receive less biased answers from the participants, thus allowing the developers to leave further and personalized comments.

Table 2: Survey questions. (MC: Multiple Choice, O: Open answer)

Section	**ID**	**Summarized Question**	**Type**	**# Resp.**
Taxonomy Evaluation	Q1.1	What is a code review?	O	52
	Q1.2	Does the taxonomy covers all changes that occur in code reviews?	MC+O	52
	Q1.3	Which Change categories/Topics occur the most inside code reviews?	O	52
	Q1.4	What kind of feedback do you expect from other developers during code reviews?	O	52
	Q1.5	What kind of feedback do you usually receive from other developers during code reviews?	O	52
Automation Needs	Q2.1	What kind of feedback would you expect from recommender-tools during code review?	O	52
	Q2.2	What kind of automation do you envision for automating code review practices?	O	52
	Q2.3	What kind of automation do you envision for the fixing and detection of Documentation issues?	O	52
	Q2.4	What kind of automation do you envision for the fixing and detection of Style issues?	O	52
	Q2.5	What kind of automation do you envision for the fixing and detection of Structural issues?	O	52
	Q2.6	Which code review change types could be automatically detected and/or fixed by tools?	O	52
	Q2.7	How would you approach the detection and fixing of the code review change types mentioned in Q2.6?	O	52

[2] https://gsuite.google.com/products/forms/

We have grouped the questions reported in Table 2 into three topics: (i) *Background*, (ii) *Taxonomy Evaluation*, and (iii) *Automation Needs*. The **Background** questions provided us with demographic information (reported in Section 3.1). However, for brevity, we omit these questions in the table, providing the full survey information in the replication package. The questions in the other two sections, *Taxonomy Evaluation*, and *Automation Needs*, represent the core part of the survey, aimed at understanding code review practices and related automation needs.

The **Taxonomy Evaluation** section was aimed at assessing the taxonomy completeness (RQ_1) and to investigate the type of feedback developers usually receive/expect in code reviews (RQ_1). To reach this goal, contextually to the five questions of section *Taxonomy Evaluation* (Q1.1-Q1.5), we shared two images of the taxonomy and also a link to the full taxonomy where they could have adjusted it and send it back (described in Section 3.1). Again the shared pictures are available in our replication package. In this stage of the survey, developers could evaluate the taxonomy and suggest further categories to integrate into it (Q1.2), describing also the feedback they usually expect/receive by reviewers in MCR (Q1.3-Q1.5).

To derive the final version of the taxonomy we proceeded as follows. At first, one of the authors (the second author) of the work performed an iterative content analysis [48] of the feedback provided by participants (see *EMSE_MCR_2019/survey_raw_data* in the replication package). Thus, she started with an empty list of MCR change type categories and carefully analyzed each feedback provided by the developers. Each time she found a new *MCR change type category* to add to the taxonomy obtained after the inception phase, a new category was added to the connected list and each feedback as developers often referred to similar types was labeled with the matching categories (we provide this labeled dataset in our replication package). After this step, the initial categorization was refined performing another interaction involving one of the other authors (the first author) of this paper who double-checked each category and removed potential redundant categories in the list. Finally, the new emerged categories were added to the taxonomy obtained from the previous phase. The final version of the taxonomy, that we called (`CRAM` **C**ode **R**eview ch**A**nges **M**odel), is provided in Table 4 and discussed in Section 3.1.

To complement the results reported in these tables, in a second survey, we surveyed 14 additional participants (we invited 20 participants in total considering our direct contact lists, and 12 developers and 2 researchers actually were able to participate in the study), not involved in the aforementioned survey, to qualitatively assess the *relevance* and *completeness* of the identified MCR change types composing CRAM. Among our participants, all of them have > 4 years of development experience and use/used advanced tools for supporting MCR (e.g., Gerrit, static analysis tools). To perform such an evaluation, we shared to the participants the MCR change types composing the designed CRAM and clarified the meaning of them. After this preliminary clarification/explanation stage, we asked the participants to rate the relevance

and completeness of the identified MCR change types composing CRAM, by asking the following questions:

- Q_R: What is the perceived relevance of the following change topic occurring in MCR? Likertscale intensity from 1 (Low) to 5 (High).
- Q_C: What is the perceived completeness of the following change type occurring in MCR? Likertscale intensity from 1 (Low) to 5 (High).

The **Automation Needs** section (Q2.1-Q2.7) was focused on (i) understanding which tools developers would need during MCR (Q2.1 and Q2.2), with a particular focus on *recurrent* [71] or *critical* changes (or problems) occurring in MCR tasks (Q2.3-Q2.5); and (ii) how developers would approach the automatic detection and fixing of MCR change types required to perform in order to improve a submitted patch (Q2.6 and Q2.7).

We performed also, in this case, an iterative content analysis [48] of the feedback provided by the participants. Thus, one of the authors of the work (the first author) started with three empty lists and carefully analyzed each feedback provided by the developers. The three empty lists were respectively related to the *recommendations*, the *techniques*, and the *solutions* envisioned by the developers for automating MCR activities. Thus, each time the author found a new *recommendation*, e.g., on how to collect the data or which data to analyze for automating MCR, the feedback was added to the *recommendations* list. When the developers mentioned a specific *technique* to employ or described how the *solutions* to automate a given change should work (e.g., a specific auto-fixing strategy for detecting and fixing documentation defects [97, 99]), we added elements in the *techniques* and the list of the *solutions*. After this step, the three lists were refined performing another interaction involving one of the other authors (the second author) of this paper who double-checked each emerged category and removed potential redundant categories in the lists. We discuss the results and findings achieved by collecting the feedback of participants and related to Q2.1-Q2.7 in Section 3.2. To complement the results of the previous analysis, we surveyed 14 additional participants (the same we involved in the evaluation of MCR change types), not involved in the aforementioned survey, to qualitatively assess how *critical* and *feasible to implement* are some of the identified techniques to support MCR activities. This study motivated the usage of specific techniques over others to support MCR. Among our participants, all of them have > 4 years of development experience and use/used advanced tools for supporting MCR (e.g., Gerrit, static analysis tools). To perform such an evaluation, we shared to the participants the identified techniques to support MCR activities and clarified the meaning of them. After this preliminary clarification/explanation stage, we asked the participants to rate the *critical* and *feasible to implement* the identified techniques to support MCR activities, by asking the following questions:

- Q_C: How critical is the identified techniques to support MCR activities? Likertscale intensity from 1 (Low) to 5 (High).

Table 3: Information about the Survey Participants

Participants Profile	**Nr. (%)**
Industrial Developer	**50%**
Open Source Developer	11.15%
Senior Researcher	19.2%
CS Student	9.6%
Other	9.6%

Team Size		**Projects Size [LoC]**	
1-5	**38%**	1,000-300,000	**66%**
5-10	14%	300,000-1,000,000	15%
10-15	10%	>1,000,000	19%
>15	**38%**		

Experience (Years)		**Experience (Rate)**	
< 2	1.9%	Poor	1.9%
2-5	11.5%	Fair	0%
5-8	19.2%	Good	19.2%
>8	**67.3%**	Very Good	32.7%
		Excellent	**46.2%**

– Q_F: How feasible to implement is the identified techniques to support MCR activities? Likertscale intensity from 1 (Low) to 5 (High).

Finally, as anticipated before, we discuss the potential of leveraging summarization techniques to support MCR practices.

3 Results

3.1 RQ_1: Types of changes occuring in modern code reviews

To investigate MCR practices and achieve a complete taxonomy of MCR change types, as a first step, we advertised the study on social media channels to acquire study participants. To address more participants outside academia, we also applied opportunistic sampling [39] to find open source contributors performing code inspection in their working practices.

The first survey we made was available for two months to maximize the number of collected answers and we invited more than 200 direct contacts to fill out the questionnaire described in Section 2.3. In total, we received 52 responses, with a return rate of about 23%. Table 3 lists demographic information about our survey participants. Interestingly, among our participants, we had 31 (61.5%) industrial and open-source developers. The self-estimation of their development experience highlight that most of the developers rated themselves as "very good" (32.7%) or "excellent" (46.2%) programmers. Moreover, around 30% of them have 2-8 years of development experience, and around 67% more than 8. It is important to mention that, even if not obliged, all developers that participated in the study filled the non-mandatory open questions.

As reported in Section 4, we asked (Q1.1-2 in Table 2) our study participants to provide us feedback on the initial taxonomy obtained after the

Table 4: **C**ode **R**eview ch**A**nges **M**odel (CRAM) - **Part I**. *Full version of the CRAM model, with all descriptions, available in the appendix at the end of the paper.*

Artifact	Activity	Category	Topic	Detailed Change
Production & Test Code (Modification occurring in production and test code)	**Maintainability & Perfective Maintenance**	**Documentation (D)**	**- Textual Documentation**: Issues concerning the documentation through textual representation, such as naming of classes, method, variables. This also includes license headers, typos in either inline comments or Javadoc	**(D.1) - Naming**. **(D.2) - Comments**. **(D.3) - License Header**: Issues regarding missing or wrong license headers inside source-files. **(D.4) - Typos**: Spelling mistakes in the documentation **(D.5) - Other**.
			- Language Supported Documentation: Documentation through statements/elements that the programming language offers (e.g., java public modifier to document that it is accessible from the outside)	**(D.6) - Immutability**. **(D.7) - Visibility (Modifiers)**.
			Style (S)	**(S.1) - Brackets & Braces**. **(S.2) - Indentation**. **(S.3.) - Blank Lines**. **(S.4) - Long Lines**. **(S.5) - Whitespace Usage**. **(S.6) - Grouping**. **(S.7) - Commented out code**: remove code that is commented out (also TODO and FIXME)
		Structure (STR)	**- Re-implementation**: Structural defects require an alternative implementation method. For example, replacing the program's array data structure with a vector and knowing the existence of prebuilt functionality that could be used instead of a self-programmed implementation would be considered a solution approach defect. Therefore, solution approach defects are not about re-organizing existing code but rethinking the current solution and implementing it in a different way.	**(STR.1) - Semantic Duplication**. **(STR.2) - Semantic Dead Code**. **(STR.3) - Change Function**. **(STR.4) - Standard Coding Conventions**. **(STR.5) - New Functionality**. **(STR.6) - Strings (Wording)**: Issues regarding contents of strings, badly composed strings **(STR.7) - Logging**: Add the ability to methods for logging results or errors **(STR.8) - Testing**: Issues regarding test coverage, wrong/inappropriate tests, additional tests etc.
			- Organization: Defects that can be fixed by applying structural modifications to the software. Moving a piece of functionality from module A to module B is a possible strategy for this.	**(STR.9) - Imports**: Issues with wrong or missing or unused import statements **(STR.10) - Move Functionality**. **(STR.11) - Long Sub Routine**. **(STR.12) - Dead Code**. **(STR.13) - Duplication / Redundant Code**. **(STR.14) - Complex Code / Simplification**. **(STR.15) - Statement Issue**. **(STR.16) - Consistency**. **(STR.17) - Architectural changes**: code reviews often result in a change to the system architecture, like splitting an interface into two distinct interfaces, introducing abstractions, or the inclusion of design patterns
	Functionality/ Corrective Maintenance		**Interface (I)**	**(I.1) - Function Call**. **(I.2) - Parameter**.
			Logic (L)	**(L.1) - Compare**. **(L.2) - Computation**. **(L.3) - Wrong Location**. **(L.4) - Algorithm/Performance**.
			Resource (R)	**(R.1) - Variable Initialization**. **(R.2) - Memory Management**. **(R.3) - Data & Resource Manipulation**. **(R.4) - Security**: Issues related to the application's/software's security aspects **(R.5) - Concurrency**: Issues regarding concurrency
			Check (C)	**(C.1) - Check Function**. **(C.2) - Check Variable**. **(C.3) - Check User Input**.
			Larger Defects (LD)	**(LD.1) - Completeness**. **(LD.2) - GUI**. **(LD.3) - Check outside code / Domino Effects**.

Table 5: **C**ode **R**eview ch**A**nges **M**odel (CRAM) - **Part II**.

Artifact	Activity
Other Changes Changes not typically found in source-code files (.java, .py, .cpp etc.) which are nonetheless essential to the runtime of a project	**(O.1) Commit Message**: Changes in the commit message of a submitted patch. Mostly related to wrong description of the change or not capturing all changes.
	(O.2) Continuous Integration / Continuous Delivery configurations: Changes to configuration files concerning the Continuous Integration or Continuous Delivery pipeline/setup.
	(O.3) Automated Static Analysis Tools configurations: Changes in the configuration of Linters, Checkers, Recommenders used in the project (e.g., Checkstyle, PMD, FindBugs etc.)
	(O.4) Language or Framework specific: Changes to files native to the used programming language. For example MANIFEST for Java.
	(O.5) External Software Documentation: Changes to the external Software Documentation files
	(O.6) Runtime Configurations: docker-configs, ansible playbooks, delivery configs etc.
	(O.7) Other: Includes changes to XML, Scripts, README files, HTML files and Version Control

inception phase (see Section 3.1). As results, only 28% of developers claimed (Q1.2) that the proposed taxonomy was incomplete, reporting a total of 17 sentences related to additional (not reported in the previous version of the taxonomy) tasks, activities or changes occurring during MCR. The encoding of such sentences resulted in the identification of a total of three additional change types (highlighted in **BLUE** in Table 4), not previously found in the *inception phase*. These categories were integrated into the ***final set of MCR change types composing CRAM***. For more information, the intermediate taxonomies, and all codified developers' feedback are available in the replication package.

Table 4 and Table 5 provide an overview of CRAM. It is important to mention that, for reason of space, the full version of Table 4 (i.e., with all descriptions) is provided in the appendix at the end of the paper. To facilitate the understanding of this taxonomy/model, in these tables, we grouped each MCR change type according to different high- and low-level dimensions: (i) *artifact type* involved in the change (e.g., test and production code or configuration files); (ii) *type of MCR activities/changes* performed (e.g., perfective and corrective maintenance); (iii) specific *MCR change categories* associated to each activity (e.g., changes related to artifacts structure, their logic and/or resource utilization); and finally, (iv) the detailed topics and fine-grained *changes* associated with each MCR change category. Moreover, Tables 4 and 5 highlight with different colours the detailed MCR change types emerged during the *inception and the taxonomy definition phases* (described in Section 2). Specifically, (i) in **BLACK** are highlighted MCR changes types that overlapped with the schema by Beller *et al.* [22]; (ii) in **RED** are highlighted the categories emerged during the manual analysis of MCR commits and comments of ten open source

projects and that were not present in the schema by Beller *et al.* (see Section); (iii) in **BLUE** are highlighted the additional change types suggested by the developers and that were not present in the taxonomy emerged after the inception phase.

It is important to mention that, most (78%) developers (Q1.1) reported that MCR practices represent a useful way for facilitating the team knowledge transfer as well as to improve the overall quality and performance of the code under review. This preliminary finding is consistent with the results by Bacchelli and Bird [14]. However, we also discovered that, compared to the schema by Beller *et al.* [22], new emerging change types characterize MCR activities and that novel tools are needed to support such activities.

As reported in Table 4, `CRAM` includes MCR changes related to the *structure*, *documentation* and *style* of the ***test and production code***. Other changes are performed to fix issues related to the way existing or added functionalities are implemented in the patch under review, such as *interface* (issues related to the communication with a different part of the system), the *logic* of the code, its *resource* allocation/consumption, wrong/incomplete *checks* of values assigned to code elements, and different types of *defects*. In addition, in Table 5 are reported further MCR change types related to the modifications made by developers in ***non-source-code artifacts*** which are, in some cases, also essential to the runtime of a project: (i) configuration files related to the *continuous integration and continuous delivery* processes, and *static analysis tools*; (ii) *language or framework specific* files; (iii) *changes to external software documentation*; (iv) other files related to *runtime configurations* (e.g., Docker files); (v) *committed* files; and (vi) *other artifacts* (e.g., README files).

Documentation (D), Style (S) and Structural (STR) changes/issues are, as reported by 60% of our study participants, very recurrent in both traditional and MCR. According to the schema by Beller *et. al.* [22], most documentation changes are needed to fix issues concerning (ii) missing, wrong, incomplete Javadocs and inline comments (D.1); and (ii) inconsistent naming applied in documentation and code (e.g., variables) of the system (D.2). During the inception phase, we also found that developers in MCR also carefully review and change, when needed, the *license headers* (D.3) and fix potential *typos* (D.4) in either inline comments or Javadocs. Interestingly, these MCR change types (D.3-4) were not present in the schema by Beller *et. al.*. Our study participants also claimed that *"tools like PMD, Checkstyle already detect some of such problems (D.4)"*, e.g. typos, *"but are not always so accurate"*. In addition, reviewing and/or updating the header of Java classes represent an *"important task to avoid licensing issues"* [89] and avoid that the software documentation is in general *"not updated or incomplete"*.

Coding Style best practices concern the way the code is written and appear to developers, e.g., code indentation (S.2), the usage of whitespace (S.5), and blank lines (S.3). We found, during the inception phase, changes not present in the schema by Beller *et. al.* [22] and related to the removal of *commented out code* as well as TODO and FIXME comments (S.7). However,

also in this case, our study participants claimed that *"tools for this already exists, like PMD and Checkstyles"* [71].

Structural defects require alternative implementations and/or refactoring operations in both test and production code. As reported by Beller *et. al.* [22], (i) re-implementation changes (STR.1-5) can involve the need to remove or modify semantic dead code (STR.2) and semantic duplications (STR.1) as well as the need to improve the code according to coding conventions (STR.4), to remove function calls to deprecated functions (STR.3) or to facilitate the evolvability (STR.5) of the code under review; instead, (ii) organizational changes (STR.10-13, STR.15-16) are related to defects that can be fixed by applying structural modifications (e.g., refactorings) to the software. Further re-implementation or organizational changes/issues (STR.6-8, STR.9 and STR.17), not included in the schema by Beller *et. al.*, are related to (STR.6) *strings* badly composed, (STR.9) *wrong/missing imports*, and bad *testing practices* (e.g., low test coverage, inappropriate tests, the need of additional tests, etc.). Complementarily, our participants highlighted as additional CR change type the need to add methods for (STR.7) *logging results* or errors. Moreover, they also reported that code reviews often result in (STR.17) *architectural changes* to the system, like splitting an interface into two distinct interfaces, introducing abstractions, or the inclusion of design patterns.

As we can see from Table 4, during the inception phase, we found that developers in MCR try to address (R.5) *concurrency* problems, while in the taxonomy phase, developers strongly highlighted the relevance of (L.4) *performance*, (R.2-3) *resource consumption*, and (R.4) *security* issues (e.g., they claimed that reviewers in MCR should provide answers to questions like *"have added a performance bug in my change? - have I added a security bug in my change?"*). This finding is interesting because in previous work by Bacchelli and Bird [14], security and performance aspects were not relevant aspects to discuss in MCR. As reported by a cloud developer that participated in the study, this result can be explained by the emerging need to ensure in MCR *"the quality of [...] cloud applications, in terms of performance, security and software quality"* [56].

The new emerging MCR changes (or issues) concerning ***test and production code*** *are related to the need to fix (i) licensing and security issues; (ii) strings badly composed and wrong/missing imports; (ii) potential typos in either inline comments or Javadocs; (iii) the removal of commented out code; (iv) the application of bad testing practices; and finally, the handling (iv) of architectural changes to the system.*

Changes in *non-source-code artifacts* reported in Table 5 represent the set of MCR change types that were not present in the schema by Beller *et. al.* [22] and emerged in both inception and taxonomy phases. Some of the emerged problems are related to sub-optimal configuration of continuous delivery and integration files (O2) that led to sub-optimal instantiations of the continuous delivery (CD) and continuous integration (CI) pipelines. Further MCR change types are related to the modifications made by developers on (i) configuration files of (O.3) *static analysis tools* (SATs) to improve their performance and effectiveness; (ii) (O.4) *language or framework specific* files; (iii)

O.5) *external software documentation*; (iv) files responsible for O.6) *runtime configurations* (e.g., Docker files); (v) O.1) *commit messages* to improve their quality; and (vi) O.7) *other artifacts* (e.g., scripting and README files). The aforementioned findings are also very interesting, since, differently from previous research [14], reviewers in MCR also care about CD and CI topics and practices, something that needs further investigations in future research.

The new emerging MCR modifications related to ***non-source-code artifacts*** *concern changes on continuous delivery and integration configuration files, files for runtime configuration, static analysis tools configuration files, and other non-source-code artifacts (e.g., commits and external software documentation).*

In summary, it is interesting to highlight how most of the novel MCR change types (highlighted in BLUE or RED) in Table 4 and Table 5 are related to changes or issues that developers perform or have to deal with because of the availability of new emerging development technologies (e.g., cloud-based technologies) and practices (e.g., Continuous Delivery and Continuous Integration). For instance, the management of CD/CI pipelines [31, 32, 94] and SATs configurations [15, 71] represent an important task to improve both developer productivity and development practices in modern software systems [15, 46, 80, 88]. This has pushed developers to perform additional activities or tasks during code MCR, with the aim at reviewing, re-thinking, and changing software artifacts impacting the CD and CI processes as well as the effectiveness/performance of static analysis tools [88].

Most of novel MCR change types composing `CRAM` *are related to changes or issues that developers perform or have to deal with because of the availability of new* ***emerging development technologies*** *(e.g., cloud-based technologies)* ***and practices*** *(e.g., Continuous Delivery and Continuous Integration).*

In the next section, we discuss the contemporary developers' automation needs to support the activities composing `CRAM`.

3.2 $\mathbf{RQ_2}$: Emerging automation needs in MCR

3.2.1 Emerging Developers' Automation Needs in MCR

Table 6 reports the changes that our participants consider the most frequent during MCR, Table 10 reports the feedback that developers would like to receive from reviewers, and finally, Table 9 summarizes the feedback that they actually receive in code reviews. By looking at Table 6 it is possible to observe that, according to our study participants, the most frequent change topics occurring in MCR are related to the structure (e.g., re-factorings, and reorganizations) of test and production code and the software documentation. Other MCR changes types (e.g., changes in the logic and the style of the code of the patch under review) rarely occur, covering each of them less than 8% of the MCR topics, and all together correspond to around 27% of the total MCR changes performed to a patch.

As mentioned previously, in Table 4 and Table 5 we discuss the MCR change types that emerged during our investigation, while in Table 6 we show the changes that survey participants considered occurring frequently during MCR. To investigate the extent to which there is any inconsistency between the actual change distribution and participants' mental model in Table 6, we selected the 211 review commits and analyzed them manually. Table 7 allows to visually observe whether there is a match between the analyzed MCR changes and the changes that, in our first survey, the participants considered occurring frequently during peer reviews and reported this in the paper. Results in the table show that, consistently to Table 6, most frequent changes occurring in the observed MCR commits are related to the structure (e.g., re-factorings, and reorganizations) of test and production code and the software documentation. Interestingly, *Other Changes* account for 15% of all changes, which represent the top-3 most frequent change type occurring in MCR. Other categories are rarely happening, consistently with what is reported in Table 6.

In Table 6 we discuss the MCR change types that the developers consider the most frequent during MCR, while in Table 7 we report the one that are the most frequent in the 211 manually analyzed commits. To complement the results reported in these tables, in a second survey, we surveyed 14 additional participants (we invited 20 participants in total considering our direct contact lists, and 12 developers and 2 researchers actually were able to participate in the study), not involved in the aforementioned survey, to qualitatively assess the *relevance* and *completeness* of the identified MCR change types composing CRAM. Among our participants, all of them have > 4 years of development experience and use/used advanced tools for supporting MCR (e.g., Gerrit, static analysis tools). To perform such an evaluation, we shared to the participants the MCR change types composing the designed CRAM and clarified the meaning of them. After this preliminary clarification/explanation stage, we asked the participants to rate the relevance and completeness of the identified MCR change types composing CRAM. From Table 8 and Figure 2 it is possible to observe that relevance of MCR change types does not match the frequency of reported changes in Table 6 and Table 7. Specifically, the top relevant MCR change types are *Logic, Structure, Other Changes*, and *Documentation*, with average Likert-scale intensity > 3.14. Other problems are considered less relevant by our participants, with *Check* MCR change type considered as the least important. In terms of CRAM completeness, we can observe, from Table 8 and Figure 2, that most participants consider most of MCR change types exhaustive, with Likert-scale intensity always > 4.43. From the qualitative comments of participants, we learned that *"the elements in "Other changes" are in a high-level complete, but they can be detailed in future investigations"*. On the other side, other participants are *"pretty satisfied by the completeness of the taxonomy"*, however they *"think that companies developing software in other fields (e.g., e-Health) could present rather different MCR challenges"*.

The results in Table 9 highlight how the feedback developers receive from reviewers are highly consistent with the changes they actually perform (Table 6). However, looking at both Table 9 and Table 10, it is also evident that the

Table 6: Q1.3: Frequent change topics occurring in MCR based on developers judgment

Sub-category	**Count**	**%**
Structure	*126*	***48.4%***
Documentation	*64*	***24.6%***
Logic	18	7.1%
Style	19	7.1%
Resource	17	6.7%
Interface	9	3.6%
Check	3	1.2%
Larger Defects	3	1.2%
Other Changes	1	0.2%
Total	**260**	**100%**

Table 7: Frequent change topics occurring in the 211 MCR manually analyzed commits. Note that in the table, the second column has a total count of 632, as 623 are the files changed in the 211 MCR commits

Sub-category	**MCR Changes Count**	**%**
Structure	*251*	***40%***
Documentation	*149*	***24%***
Other Changes	*93*	***15%***
Logic	38	6%
Style	37	6%
Resource	35	6%
Interface	19	3%
Check	6	1%
Larger Defects	4	1%
Total	**632**	**100%**

feedback developers receive from reviewers are often not satisfactory, i.e., rarely meet all the current expectations of developers during MCR, as reported by one of our study participants: *"many of the problems we face during code review are related to the miss-match between expectations and outcomes of a code review [...] reviewers provide feedback that are not exhaustive or timely reported. This often makes code reviews unproductive"*. Interestingly, feedback on structural and documentation aspects are less prevalent in Table 10 than in Table 9, while comments related to the *Functionality* (e.g., performance and resources) and *Other Changes* categories are, nowadays, more important for developers. For example, 8% of participants stated that they would like to receive CD/CI and SATs configuration comments, while only 1% of them receive such feedback. This general result highlights the new emerging activities and expectations characterizing MCR, and that more exhaustive feedback from reviewers is required.

Envisioned approaches. Our survey participants provided more than 400 comments on the automation needs (Q2.1-7) characterizing MCR. Hence, for reason of space, in Table 11 we summarize the top-most recurrent solu-

Table 8: Average Perceived Relevance/Completeness of change topics occurring in MCR. Likert-scale intensity from 1 (Low) to 5 (High) was used to measure perceived relevance and completeness of MCR change types

Sub-category	**Avg. Perceived Relevance**	**Avg. Perceived Completeness**
Logic	***4.50***	**4.57**
Structure	***4.07***	**5**
Other Changes	***3.36***	**4.43**
Documentation	***3.14***	**5**
Larger Defects	2.36	**4.71**
Resource	2.36	**4.86**
Style	1.57	**5**
Interface	1.50	**4.79**
Check	1.00	**4.79**

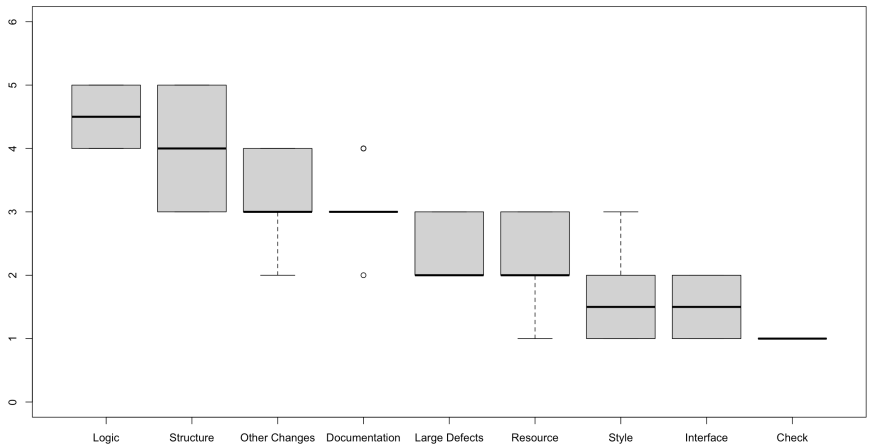

Fig. 2: Average Perceived Relevance of change topics occurring in MCR

Table 9: Q1.5: Feedback received in MCR

Category	**Ranking**	**Sub-categories**
Documentation	20 Cases (21.1%)	Naming (1), Typos (1), no category (18)
Functionality	26 Cases (27.4%)	Check (2), Interface (1), Larger Defects (4), Logic (11), Performance (2), Resources (2), Security (3), no category (1)
Other	4 Cases (4.2%)	-
Other Changes	1 Case (1.1%)	-
Structure	28 Cases (29.5%)	Architectural Changes (4), Complex Code (3), Duplication (1), Standard Coding Conventions (3), Testing (4), no category (13)
Style	16 Cases (16.8%)	no category
Total	**95**	**100%**

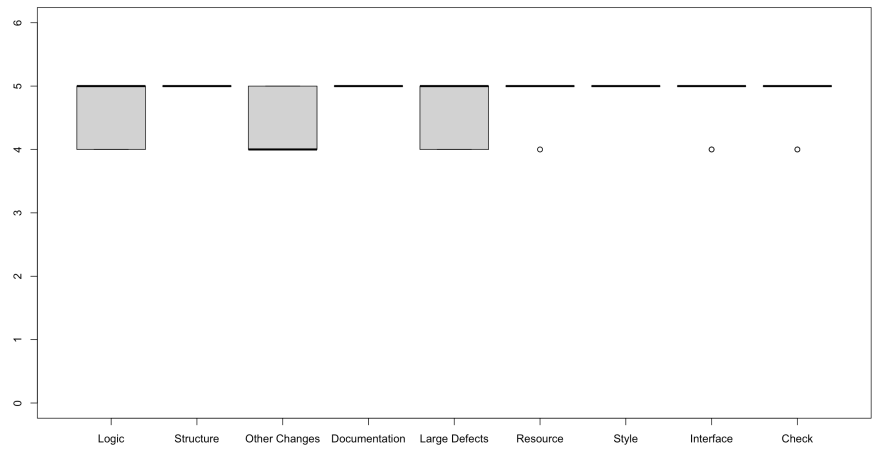

Fig. 3: Average Perceived Completeness of change topics occurring in MCR

tions/approaches proposed by the developers (Q2.1-7), with a particular focus on the new MCR change types emerged in the empirical investigation of RQ_1. Table 11 summarizes the approaches/solutions (Q2.1-7) that are needed to support developers in contemporary MCR activities. We clustered the proposed solutions in column three of Table 11, as developers often referred to similar types as developers often referred to similar types of automated solutions.

The most frequent MCR categories for which further automated approaches would be needed are *Documentation* (56), *Style* (40), *Structure* (29), *Functionality* (19), and *Other Changes* (9). In particular, 46 comments have been provided by developers for the Documentation category. For this category, developers believe that to facilitate MCR would be very helpful to conceive advanced automation able to ***detect and fix issues related to the incomplete/inconsistent documentation*** with respect to the source code. Researchers in the literature are recently exploring this problem [97], but still no automation was tested in the context of code reviews. In other cases, our participants would need approaches that ***generate directly the required documentation/comments*** (including the license header), integrating into the automation also the detection/fixing of potential (also grammar) typos. More fine-grained recommender systems are expected for both Documentation and Style categories, and are related to ***renaming recommendations***, and the ***identification and fixing of coding style*** issues of the patch under review. However, some developers also reported that tools like Checkstyle could be sufficient to handle some of the style issues.

Regarding structural MCR changes, developers would be interested in ***refactoring recommendations for both test and production code***. For instance, a developer mentioned in the survey the need for refactoring recom-

mendations *"of tests not based only on coupling concepts but also encapsulating the need of having explicitly separated testing performance from functional testing*, this to facilitate for example the test of *"different properties of microservices of a cloud application"*. Moreover, another participant mentioned the need to have timely feedback about the *"test/code smells (bad design choices) added"* in the patch under review, e.g., ***"feedback auto-generated based on test/code smells notions, providing an overview on overall test/code quality and readability"***. It is interesting to observe also that, automated tools are also needed for the detection of duplicated, unused, (semantic) dead, deprecated code (*"highlight dead code, unreachable code, and suggest refactoring options [...] for duplicates"*), and architecture violations (*"did I have introduced imperfections at the level of Architecture?"*).

As summarized in Table 11, there is a huge demand from developers of tools able to detect (and possibly generate patches for fixing) ***performance, resource consumption and security issues***. For instance, a participant of our study reported: *"... a company producing self-driving cars, in [...] code review will require also to observe potential security and or testing issues"*. Moreover, from the results of RQ_1, we observe how most of novel MCR change types composing `CRAM` are related to changes or issues that developers perform or have to deal with because of the availability of new emerging development technologies and practices. This also influenced the type of approaches that developers would need for the category *Other changes*: tools for ***recommending, improving, monitoring*** CD/CI, runtime and SATs configurations.

In Table 11, we discuss the MCR tools that developers consider important to develop to support MCR activities. To complement the results reported in this table, with a second survey, we surveyed 14 additional participants (the same involved in the evaluation of CRAM change types relevance and completeness), to qualitatively assess the *criticality* (or relevance) and *feasibility to implement* the solutions identified in Table 11. To perform such an evaluation, we shared to the participants the MCR change types composing the designed CRAM and the identified solutions to automate them, thus clarified the meaning of them. After this preliminary clarification/explanation stage, we asked the participants to rate the criticality and feasibility to implement the identified solutions in Table 11. Table 12 report the solutions identified in Table 11, highlighting, for each solution, its id, the description, the level of criticality for MCR, the feasibility to implement it, and who of the participants in MCR benefits from it. From the table the most critical solutions to implement to support MCR are S1-2 (*Documentation*), S5 (*License Header*), S11 (*Auto-fix of performance, resource issues*), S12 (*Detect security issues*), and S14 (*Recommend/improve CD/CI configurations*). Other solutions are considered less relevant by our participants, with *S4* (*Automatic spell checking and fixing*) and *S8* (*Detection of duplicated, unused, dead, and deprecated code*) considered as the least important. In terms of feasibility to implement such solutions, most participants consider S1-2 (*Documentation*), S3 (*Renaming suggestions based on standard naming used in the codebase*), S4 (*Automatic spell checking and fixing*), and *S8* (*Detection of duplicated, unused, dead, and deprecated code*)

Table 10: Q1.4: Expected feedback in MCR

Category	**Ranking**	**Sub-categories**
Documentation	14 Cases (11.4%)	no sub-category
Functionality	45 Cases (36.6%)	Check(3), Completeness (2), Data & Resource Manipulation (3), Interface (4), Large Defects (3), Logic (12), Performance (7), Resource (7), Security (4)
Other	7 Cases (5.69%)	no subcategory
Other Changes	9 Cases (7.3%)	Automated Static Analysis Tools configurations (3), Continuous Integration/Continuous Delivery configurations (2), Runtime Configurations (2), no subcategory (2)
Structure	34 Cases (27.6%)	Architectural Changes (5), Complex Code (7), Logging (1), Duplication (1), Standard Coding Conventions (2), Testing (6), no subcategory (12)
Style	14 Cases (11.4%)	no subcategory
Total	**123**	**100%**

with Likert-scale intensity always > 3.93. This means that only S1-2 (*Documentation*) solutions are considered enough relevant and, at the same time, feasible to implement.
From the qualitative comments of participants, we learned that *"Some of the problem here, header, doc, renaming, etc. could be easily fixed and grouped together. Asats (Automated static analysis tools) is by far the one in configuration that can be addressed"*. From the other side, other participants claim that *"Resourse and security are the most difficult, there rest could require work, but still can be addressed"*, however, they also believe that *"we are far from making automated configurations"*.

Recommendations, techniques, and data. Tables 14, 15, and 13 report the techniques to adopt, the MCR data to analyze and the recommendations to follow to implement the approaches/solutions described in Table 11. We got further concrete recommendations from developers. We decided to not stress too much the discussion on this part, as not surprisingly findings we achieved from the analysis of the developers' recommendations (they are available in the replication package).

Table 13, most participants suggest studying potential patterns and anti-patterns characterizing *Documentation* changes, and checking for inconsistencies between documentation and code. Similar recommendations are made for non-source code files modified according to the *Other Changes* high-level category.

For what concern *Documentation* issues, most developers recommend to perform a manual analysis to investigate patterns and anti-patterns and change/documentation metrics, then leverage NLP techniques or machine learning techniques (in combination with static code analysis) to model and

Table 11: RQ_2: Developers' Envisioned Solutions.

Category	**Detailed Change**	**Abstracted Solution**
Documentation (56)	- General (32) - Comments (3)	31 - ***Automatically detecting and fixing*** documentation issues (documentation incomplete or inconsistent with the source code) 4 - ***Generation*** and replacement of inconsistent documentation/comments
	Naming (12)	12 - ***Renaming suggestions*** based on standard naming used in the codebase
	Typos (5)	5 - ***Automatic*** spell checking (also grammar) and fixing
	License Header (1)	1 - ***Generating*** License Header
Style (40)		27 - ***Evaluate Style Consistency*** with the style adapted by the team and auto-fix the style issues 13 - ***Use existing tools*** for these issues, e.g., PMD and CheckStyle
Structure (29)	- Refactoring (8) - Duplicated, (Semantic) Dead, Unused, and Deprecated Code (19) - Architecture violations (2)	19 - ***Detection of duplicated***, unused, (semantic) dead, and deprecated code 8 - ***Refactoring suggestions*** for test and production code 2 - ***Detect architectural violations***
Functionality (19)	- Performance (4) - Resource (11) - Security (4)	12 - ***Auto-fix*** of performance, resource issues 5 - ***Detect security issues*** 3 - ***Performance and resource analysis***
Other Changes (9)	- CD/CI configurations (4) - SATs configurations (2) - Runtime configurations (3)	4 - ***Recommend/improve CD/CI*** configurations 3 - ***Recommend/improve runtime*** configurations 2 - ***Recommend/improve SATs*** configurations

Table 12: RQ_2: Criticality and Feasibility of Proposed Solutions.

ID Solution	**Abstracted Solution**	**Criticality**	**Feasibility**	**Used by**
S1	Automatically detecting and fixing documentation issues	**3.71**	**4.36**	Author/Reviewer
S2	Generation and replacement of inconsistent documentation/comments	**3.79**	**4.07**	Author/Reviewer
S3	Renaming suggestions based on standard naming used in the codebase	1.93	**4.00**	Author
S4	Automatic spell checking and fixing	1.64	**3.93**	Author
S5	Generating License Header	**3.57**	3.07	Author
S6	Evaluate Style Consistency with the style adopted by the team and autofix the style issues	2.21	3.36	Author
S7	Use existing tools for these issues	2.50	2.50	Author
S8	Detection of duplicated, unused, (semantic) dead, and deprecated code	1.43	**4.29**	Author/Reviewer
S9	Refactoring suggestions for test and production code	2.64	2.79	Author
S10	Detect architectural violations	2.57	2.21	Reviewer
S11	Auto-fix of performance, resource issues	**3.79**	2.07	Author
S12	Detect security issues	**4.50**	2.36	Author/Reviewer
S13	Performance and resource analysis	2.43	2.36	Author
S14	Recommend/improve CD/CI configurations	**4.00**	3.00	Author/Reviewer
S15	Recommend/improve runtime configurations	3.07	2.14	Author/Reviewer
S16	Recommend/improve SATs configurations	3.29	3.64	Author/Reviewer

Table 13: RQ_2: Developers' Recommendations.

	Recommendations		
Taxonomy high-level	Learn from past data (code review changes)	Find patterns (antipatterns)	Check against codebase
#mentions by participants	**68**	**134**	**11**
Documentation	0	7	3
Functionality	18	19	0
Other	25	54	4
Other Changes	3	27	2
Structure	14	16	1
Style	8	11	1

Table 14: RQ_2: Developers' Techniques.

			Taxonomy high-level					
		#Mentions by participants	**Documentation**	**Functionality**	**Other**	**Other Changes**	**Structure**	**Style**
Techniques	*Machine Learning (predictions)*	**48**	4	8	18	9	5	4
	NLP (Text Mining)	**31**	4	7	10	5	2	3
	Data Mining	**24**	1	3	10	5	3	2
	Static Code Analysis	**35**	3	8	8	8	5	3
	Dynamic Code Analysis	**18**	1	7	4	3	2	1
	Summarization Techniques	**9**	0	5	2	1	1	0
	Regex parsing	**9**	1	0	4	2	0	2
	Manual Analysis	**4**	0	0	2	1	0	1
	Literature (state of the art)	**0**	0	0	0	0	0	0
	Integrate into IDE	**11**	2	1	4	2	1	1
	Use existing Tools	**47**	3	0	22	11	2	9
	Rely on Compiler	**3**	0	3	0	0	0	0

Table 15: RQ_2: Developers' Data.

			Taxonomy high-level					
		# mentions by participants	**Documentation**	**Functionality**	**Other**	**Other Changes**	**Structure**	**Style**
Data	**Metrics**	**38**	5	13	10	2	4	4
	Change metrics	**24**	1	3	10	5	3	2
	Code metrics	**16**	0	0	8	4	3	1
	OO-metrics	**4**	0	0	2	1	1	0
	Natural language	**0**	0	0	0	0	0	0
	Code documentation	**9**	2	3	2	1	0	1
	No data specified	**126**	14	12	14	43	21	22

find/predict incomplete or inconsistent documentation with respect to the source code.

Our participants recommended for *Other Changes* issues (i) to study patterns and anti-patterns characterizing non-source code artifacts from historical data, then (ii) observe with data mining and machine learning techniques the impact of such anti-patterns in the development process and practices (e.g., trends in change/code metrics, build failures, etc.), and finally (iii) leveraging NLP and summarization techniques [41, 65, 70] to provide more context about the detected issues, and recommending the changes to perform to fix them. For other categories in our taxonomy, we also received many interesting recommendations, and it is interesting to observe that *most participants mentioned the need to implement solutions based on customized approaches leveraging machine learning, NLP and data mining techniques modeling the problems with the notion of anti-patterns, and change metrics.* More important, as reported in Table 12, they are also ***the most critical and feasible to implement***. Thus, in the next section, we discuss the feasibility and the potential of using NLP-based techniques namely summarization techniques, to facilitate MCR activities.

3.2.2 ***The Role of Summarization Techniques in MCR activities***

It is interesting to observe that for all categories in Table 14 none of the participants mentioned the possibility to use an existing technique from the literature, but rather implement solutions based on customized approaches leveraging machine learning, NLP and data mining techniques modeling the MCR problems with the notion of anti-patterns, and change metrics. For instance, as reported in the previous section, one of the participant mentioned the need to have timely feedback about the:

> *"... test/code smells (bad design choices) added" in the patch under review, e.g.,* ***"feedback auto-generated based on test/code smells notions, providing an overview on overall test/code quality and readability"****.*

In this context, it is important to mention that open source tools for MCR management such as Gerrit, allow adding inline comments to source or test code, so that authors of code under inspection can actually improve it more easily.[3] We argue that summarization techniques [70, 73] can complement current techniques related to the analysis and detection of test smells [28, 69, 87] in the context of MCR, thus enhancing such a feature. In particular, we believe that combining test/code smells analysis [28, 69, 87] and summarization techniques [64] can help developers to have a better awareness of test suites quality, with inline comments automatically generated by tools, instead of humans (the reviewers). In the next section, we qualitatively validate/evaluate the feasibility of this research direction, proposing a trivial

[3] See for example the Gerrit Review UI https://gerrit-review.googlesource.com/Documentation/user-review-ui.html

```
1|    /**
2| a) * Some problems were detected:
3| b) * - This test contains a method that does too many things
4|     * at once. This makes the code hard to understand and
5|     * maintain.
6| c) * This method accounts for 50% of all found problems
7|    * in this test class. This smell represents 28.85%
8|    * of all found problems in the project with 6.67%
9|    * occurring in this test.
10|    **/
```

Fig. 4: Part of Test Suite Level Summaries for *UtilCacheTest.java*

```
1|   /*
2| a) * - This method requires too many parameters.
3|     */
```

Fig. 5: Part of Test method-level Summaries (for *UtilCacheTest.assertKey()*)

approach to address this challenge.

3.2.1.1 An Approach for unit tests Quality assessment in MCR

In this section, we elaborate an approach designed to automatically generate test case summaries [64, 73] of the portion of code of each individual test that is affected by structural [18, 28, 87] and textual [69] smells. This approach can be used to generate MCR comments automatically and integrated in MCR management tools such as Gerrit. We notice that existing approaches on code or test summarization [27, 57, 63, 73, 84] generate static summaries of the source or test code without taking into account which part of the code is affected by test/code smells, and these techniques have been never used in the context of MCR activities. The approach we designed consists of three steps, elaborated later in this section: (1) *Smell Detection*, (2) *Summary Generation*, and (3) *Description Augmentation*.

Smell Detection

In this step, the proposed approach takes as input the production code and the test code of a given project and detects (a task performed by DECOR [62] and TACO [69]) the smells affecting the analyzed project. During the detection phase, DECOR first finds the list of files that are to be examined. These are either all *JAR* files or all *test class* files of the project. After this preparation step, DECOR goes through the list of detected classes and examines the model for anti-patterns using a set of structural rules and metrics [61]. Differently from DECOR, which analyzes a system at the structural level, TACO detects smells in the code by leveraging techniques based on textual analysis. TACO detects smells by evaluating textual information that is contained in various elements of the source code and by computing the textual similarity between such code elements. It is important to mention that in our preliminary evaluation, we focus on the generation and qualitative evaluation of summaries related to two types of smells [61]:
- LongParameterList: a method with more than 3-4 parameters. This

smell might be introduced after the merging of several types of algorithms in a single method and can be fixed with various refactoring operations, e.g., ReplaceParameterWithMethodCall, IntroduceParameterObject [36].
- LongMethod: A method (or a test method) contains too many LOC. Generally, any method longer than ten lines of code is a symptom of a bad design choice. This smell can be fixed with various refactoring operations, e.g., ExtractMethod, IntroduceParameterObject., etc. [36].

Summary Generation

The proposed approach generates natural language phrases for describing the underlying portion of the code affected by smells by implementing an approach inspired by the well-known Software Word Usage Model (SWUM) proposed by Hill *et al.* [44]. The basic idea behind the SWUM is that *actions*, *themes*, and *secondary arguments* can be derived from an arbitrary portion of test and production code, this information can be used to link linguistic information to programming language structure and semantics. For instance, method signatures (including method name, type, and parameters) usually contain *verbs*, *nouns*, and *prepositional phrases* that can be expanded in order to generate readable natural language sentences. For example, *verbs* in method names are considered by SWUM as the *actions* while the *theme* can be found in the rest of the name. The descriptions are generated, as done in previous work [41], with *natural language templates* [41] (shared in our replication package) that are augmented by the information that is gathered from the smell detection process.

Smell Description. The summaries generated by our approach are composed of the smell specifications and categorizations by Fowler [36], van Deursen [28], Mäntylä [54] and Meszaros [60]. The long *smell descriptions* are used at the class level, while short smell descriptions are for method-level comments. The smell descriptions have the purpose of highlighting the design problems to the developer, by providing a detailed description of the detected smells. We believe that this can facilitate developers during the test/code quality assessment steps of MCR, thus, spotting the potential problems caused by the smell as well as the localization of the cause of the smell. The shorter method descriptions further assist in localizing the cause of a smell.

Quantitative Description. We provide to the developer with quantitative descriptions related to the occurrences of the smells in the project. First of all, our approach reports how dominant a type of the smell is in the test class compared to all types of smells detected in that test class, this according to the following formula: $D_{smell} = 100 \times \frac{smellOccurrencesOfTypeA}{allSmellOccurences}$. Then, it provides information on how often this smell is frequent compared to all the smells found in the project: $F_{smell} = 100 \times \frac{smellOccurencesInProject}{allSmellOccurencesInProject}$. Finally, it displays how frequent is this smell in the test class compared to all the smell occurrences in the project: $C_{smell} = 100 \times \frac{smellOccurrencesOfTypeAInClass}{smellOccurencesOfTypeAInProject}$.

The following example shows the template we used to display to developers the quantitative description:

"This method accounts/These methods account for $< D_{smell} >$ *% of all found problems in this test class. This smell represents* $< F_{smell} >$ *% of all found problems in the project with* $< C_{smell} >$ *% occurring in this test."*

Description Augmentation

In this final step, the original JUnit test classes are enriched with the above-generated descriptions, which are aggregated at the test class and test method-levels.

Test Suite Level Summaries consist of four elements: **a)** a description concerning the found smells; **b)** a detailed description of the smell(s); **c)** and a quantitative description of the frequency of the smell in the test class and the whole project. Figure 4 displays part of the smell descriptions generated for the class UtilCacheTest from Apache OFBiz. The different elements of the descriptions outlined above are highlighted with appropriate colors.

Test method-level Summaries. method-level comments are used to narrow down the root of the problem. Those comments are generated for Method Smells, i.e., problems whose source of the smell is a method. Method descriptions consist of one element, i.e., the short description of the smell, observed in Figure 5, which presents the method descriptions for UtilCacheTest::asserKey() from Apache OFBiz.

3.2.1.2 Qualitative Evaluation of the Approach in the context of MCR

Evaluation. To evaluate the (perceived) usefulness of the proposed approach we formulated the following question:

- **RQ2**$_1$ *Are test case summaries enriched by test smell information considered useful by developers?* Our objective is to investigate whether test smell summaries are considered useful by developers to better understand test case quality during MCR activities.

Study Context. The *context* of this exploratory study consists of (i) *objects*, i.e., Java classes and Test Cases extracted from a Java open-source project, and (ii) *participants* analyzing the selected objects, i.e., professional developers, researchers, and students. Specifically, the object system is *Apache OFBiz* [4]. From this project, we selected four Java classes: (i) `FlexibleStringExpander` that expands String values that contain Unified Expression Language syntax; (ii) `TimeDuration`, which implements an immutable representation of a period of time; (iii) `FlexibleMapAccessor` that can be used to flexibly access Map values; and (iv) `UtilCache`, which consists in a generalized caching utility.

[4] https://ofbiz.apache.org/

Table 16: Java classes of *Apache OFBiz*

Class	LOC	Methods
FlexibleStringExpander	728	51
TimeDuration	399	24
UtilCache	792	58
FlexibleMapAccessor	235	14

Table 17: Test cases of *Apache OFBiz*

Class	LOC	Tot. Test Smells	LongParameterList	LongMethod
FlexibleStringExpanderTests	332	2	1	1
TimeDurationTests	177	2	1	1
UtilCacheTests	429	2	1	1
FlexibleMapAccessorTest	189	2	1	1

Clearly, we considered also the related test cases: `FlexibleStringExpanderTests`, `TimeDurationTests`, `UtilCacheTests` and `FlexibleMapAccessorTest`. We selected the aforementioned Java/test classes since they are non-trivial, but it is feasible to analyze them within 30 minutes. Moreover, they do not require to examine (too many) other classes in the project. Table 16 and Table 17 detail the characteristics of the Java/test classes used in the experiment.

To recruit participants for our study we sent email invitations to developers and researchers in our contacts list. In total, we sent out 53 invitations (25 researchers and 28 developers). As reported in Table 18, 21 subjects (40%) decided to perform the experiment: 8 were professional developers, 13 were students or senior researchers. Considering all participants, most (71%) of them had at least 2-5 years (up to 10 years) of prior experience in software testing and Java programming. Among the 13 involved students or senior researchers, 5 were Master students, 6 PhD students, and 2 senior researchers.

Experimental Procedure. The experiment was conducted offline, i.e., we have sent via email to the participants the required experimental material with instructions about the tasks to perform. During the tasks the participants were guided via Google Forms [5], this also to collect information about the performed activities. The emails, surveys and experimental material we shared to the participants can be found in our replication package. Specifically, we send to each participant an experiment package composed by (i) a pre-questionnaire (to collect information about the profile and experience of each participant), (ii) surveys with instructions and materials to perform the tasks, and (iii) a post-questionnaire. Before the study, we explained to the participants the expected tasks: two code review tasks, each involving two pairs of Java and test classes.

Tasks assignment. Each participant received two tasks: (i) one task included two Java class and the two corresponding JUnit test cases (one of them enriched with the test smell summaries); (ii) the second task consisted

[5] `https://docs.google.com/forms`

Table 18: Experience of Study Participants

Programming Exp.	Absolute #	Testing Exp.	Absolute #
5 months-2 years	3 (14.30%)	5 months-2 years	6 (28.60%)
2-5 years	8 (38.10%)	2-5 years	6 (28.60 %)
5-7 years	2 (9.50%)	5-7 years	3 (14.30%)
7-10 years	6 (28.60%)	7-10 years	4 (19.00%)
$>$ 10 years	2 (9.50%)	$>$ 10 years	2 (9.50%)
Σ	9 (100%)		9 (100%)

of two Java class and the two corresponding JUnit test cases (one of them enriched with the test smell summaries). To evaluate the usefulness of the proposed approach, *for each task*, we provide the summaries to a balanced set of participants:

- group A (with 10 participants) received the first test class with summaries while for the second test class the summaries were not provided;
- group B (with 11 participants) received the first test classes without summaries while the second one was enriched with test case summaries.

Tasks description. Before starting the experiment, each participant filled a pre-study questionnaire to collect information about their programming and testing experience. After filling the questionnaire, they could start performing the first task by relying on the workspace (provided via email) containing the required project data (i.e., the Java and test classes). The stated goals were (i) *to inspect the test cases*, and (ii) *to detect*, with a special focus on the removal of LongParameterList and LongMethod smells. To facilitate this task we provided a document (included in the replication package) describing the notion of test/code smells, the types of smells potentially affecting test cases and the recommended refactoring operations to remove them.

In the instructions, we accurately explain that the generated JUnit test cases need to be maintained and updated according to the provided notion of test/code smells. Hence, participants were asked to read the available test suite and to change the test cases to (eventually) remove the detected test smells. For each pair of Java and test classes participants were instructed to spend no more than 30 minutes. In total the expected duration of the experiment is a bit longer of two hours, including the completion of the two tasks and the filling of all questionnaires.

The participants had the possibility to finish earlier each task if they believed that all smells were detected. After the experiment, we asked the subjects to fill a post-experiment survey. We used it for collecting qualitative insights and feedback.

Research Method. At the end of each task the participants filled the *post-task surveys* while, after the whole experiment, they filled also the *post-experiment questionnaire*, providing us information about the perceived usefulness and relevance of the provided test smell summaries during the performed MCR tasks.

Perceived test cases summaries usefulness and comprehensibility. At the end of the experiment, we asked specific questions to our study

Table 19: Raw data of the post-experiment questionnaire

Questions	Disagree		No Strong	Agree	
	Fully	Partial	Opinion	Partially	Fully
Q1: Do you easily understand and relate the generated descriptions with the code?	0%	9.50%	9.5%	14.30%	**66.70%**
Q2: Is it difficult to understand the test method-level descriptions?	**57.10%**	**33.30%**	0%	4.80%	4.80%
Q3: Is it difficult to understand the test suite level descriptions?	**61.90%**	**33.30%**	0%	4.80%	0%
Q4: Are the generated Test Smell Summaries useful to be more aware of the general test quality?	4.80%	0%	0%	**47.60%**	**47.60%**
Q5: The task without the generated comments/descriptions is prohibitively difficult?	4.80%	0%	9.50%	**57.10%**	**28.60%**

participants with the aim to investigate the perceived comprehensibility and usefulness of provided summaries during the performed code review tasks. It is important to mention that the majority of participants believe that the tasks were reasonably difficult to perform (95% of participants) but they had enough time to complete them (71.40% of participants).

As reported in Table 19 the participants evaluated (in $Q1-3$) the comprehensibility of descriptions provided by our approach using a Likert scale intensity from very-low to very-high. Results of Q1 highlight that 81% of participants believe that in general, the provided descriptions are easy to read and understand. Moreover, when asking the same question regarding descriptions at the test method and test suite levels ($Q2-3$), the perceived readability of generated summaries is *high* or *very high* for $90\%-95\%$ of them. Interestingly, looking at the results of $Q4-5$ (see Table 19), around 95% of developers considered the test smell summaries (when available) as a relevant source of information to perform the tasks (Q5) and to be more aware on the analyzed test suite quality (Q4). In addition, around 85% of participants also believe that performing the tasks without the generated comments would be prohibitively difficult.

> ***RQ2***$_1$*: According to human judgments the generated test smell summaries are (i) easy to understand and are (ii) perceived as a useful source of information to perform code review tasks aimed at improving the test suite quality.*

As confirmation of this general finding, we received positive feedback from many participants, such as *"the combination of class and method descriptions are useful"* and *"the descriptions at the class level provide a good overview of the test suite problems."*

Even if the overall judgment of participants was positive, we also got several suggestions for improvement:

- ***The relevance of the test suite and method-level summaries***: our participants believe that it *"useful to have the comment in the actual place where the smells are located"* and that *" descriptions at both levels serve important purposes"*. However, they also think that *" it was a bit of a nuisance having to scroll back to the top to see the Suggestion"*.
- ***Unnecessary or redundant information***: developers of our study were concerned by the fact that *"the descriptions are a little bit redundant in general"* and that in some cases the *"description of the method arguments is unnecessary"*.
- ***Information to integrate into the summaries***: as important feedback, some participants suggested to *"leverage the extracted static information and descriptions for guiding the fixing with potential patches"* and to provide *"suggestions on how to split the code to reduce the size of the method. For instance, if some (parameter) values are redundant and may be deduced from other parameter values"*.

4 Threats to Validity

Threats to *construct validity* concern the design of our study. We advertised the survey through social media channels and by opportunistic sampling, and thus we could not avoid the lack of conscientious responses. Also, given the evaluation of the survey, some responses included imprecisions: in fact, some answers given were superficial or incomplete. In order to mitigate these threats, ambiguous and incomplete answers were discarded during the evaluation of the survey. In particular, in the replication package, folder *RQ2_automation_needs/* (files *Q2.1-Q2.5_evaluation_survey.xlsx* and *Q2.6-Q2.7_evaluation_survey.xlsx*), we provide information about the number of discarded answers.
Another threat to construct validity are the steps involved in the development of `CRAM`, as this involved manual classification of code review changes and the qualitative analysis of the feedback gathered in the survey. Indeed, there is a level of subjectivity involved when deciding if a feedback or review change belongs to a certain category. To alleviate some of these threats we based `CRAM` on three different sources of change type information: (i) manual classification of commits/comments of ten different Java open source projects, where each of them was double-checked by two authors of the paper (case of disagreements were further discussed and resolved); (ii) integration of an existing taxonomy from literature; and (iii) the feedback from developers, which was again reviewed by one other author.

Threats to *internal validity* concern factors that could have influenced the results of our study. A primary threat exists concerning the definition of our taxonomy, as some categories of review changes could be missing or even overlap with others. To mitigate this threat we grouped the taxonomy into high and low-level categories in order to minimize the risk of an incomplete taxonomy.

Threats to *external validity* concern the generalization of our findings. Indeed, our investigation of review changes is limited to ten Java open source projects (all within the Eclipse ecosystem), and 52 developers participants. We alleviated some of these threats by choosing projects with different domains and sizes. However, we also observe that the dataset consists in projects having in some cases 1 review change because of the filtering step described in the design section. Thus, for future work, we plan to extend the study with further projects, to further limit this identified threat. Moreover, participants in our study have different backgrounds and most of them have more than 8 years of programming experience and more than 60% of them have an industrial profile. Finally, the dataset we studied was limited, consisting of less than 700 review comments obtained from Gerrit, which might restrict the generalisability of our findings in settings such as other programming languages, projects and reviews. However, MCR comments, changes and developers' comments were complementary sources, combined to provide a more complete view of MCR practices.

5 Related Work

This section discusses related literature investigating modern code review process and practices, as well as approaches and tools to support code review activities and tasks.

5.1 Modern Code Review Process and Practices

To the best of our knowledge, Rigby *et al.* [77–79] are the first that empirically investigated the use of code reviews in open-source projects. In this context, Weißgerber *et al.* [92] found that, in general, the probability of a patch to be accepted is about 40%, while Baysal *et al.* [20] discovered that patches submitted by casual contributors have a higher probability to be not reviewed compared to the patches submitted by core contributors. Nurolahzade *et al.* [66] confirmed such findings and showed that reviewers also try to identify and eliminate immature patches.

Other work focused on how developers perform code reviews in industrial and FLOSS projects [14, 55]. Mäntylä *et al.* [55] analyzed the code review activities of commercial and FLOSS projects, discovering that the type of defects fixed in code reviews are related in most cases to non-functional aspects of the software. Bacchelli and Bird [14] studied the code review process across different teams at Microsoft and found that the available tools for code review do not always meet developers' expectations. Our work is, in principle, very close to the one of Bacchelli *et al.* [14], as we are interested in filling the gap between expectations and outcomes of code review tools, (i) by studying the types of changes addressed during a code review; (ii) investigating the automated support that developers need or expect during code review activities.

Recent work studied the relevant social dynamics characterizing the code review process [19, 24, 52, 58]. First of all, McIntosh *et al.* [58] studied developer participation during code review and discovered that the degree of freedom that reviewers have impacted both reviewing environments and software quality. Following this line of research, Kononenko *et al.* [52] confirmed the importance of code review participation, highlighting that reviewer workload-/experience, and participation impact the quality of the code review process. Other work identified important aspects impacting software quality during code review activities, separating them in technical and non-technical factors [21, 47].

Finally, researchers investigated the characteristics of high quality [33, 76] or fair reviews [25, 38, 51] as well as the actual defects and problems developers actually fix during code reviews [22, 53]. A very close work to ours is the one by Beller *et al.* [22] where the authors manually classified over 1,400 changes taking place in reviewed code from two OSS projects into a validated categorization scheme, classifying them into *evolvability changes* and *functional changes*. Our taxonomy is not only more fine-grained compared to the one proposed in previous work, but according to our study participants, is more complete. It is important to mention that other less recent works have developed approaches for analyzing and classifying change types based on code revisions [35], analyze API change evolution [30], or more in general the project history of projects [50]. Similar tools could be used in the future to develop some of the envisioned solutions.

5.2 Automation in Modern Code Review

Recent research proposed tools, and or strategies to automate some decisions and actions during code reviews [15, 16, 26, 42, 67, 71, 81, 85, 88, 95, 96], as well as proposed methods to evaluate them [45].

The use of static analysis SATs to find defects (whether or not they may cause failure) is a common practice for software developers [34, 49, 86, 91], and recent research investigated its usage in the context of code review [71, 88] compared to other development contexts [23, 93]. Advanced approaches have been proposed to support coding or collaborative activities concerning the code review process [15–17, 59, 68, 95]. First of all, to help authors improving their patches, researchers proposed techniques based on textual, static and/or historical analysis to recommend appropriate peer reviewer(s) for evaluating a given patch [15, 42, 67, 85, 95] In addition, to help both reviewers and authors coding/reviewing activities, Barnett *et al.* proposed an approach to automatically decompose code review change-sets [16], while Baum *et al.* proposed a strategy to recommend the files to focus on during a review [17]. The Human-computer interaction (HCI) community also has done some studies that investigate the effectiveness of static analysis tools to peer code reviews from developers' perspective [43, 82], which complement the view of the aforementioned works. Finally, Zang *et al.* [96] presented an interactive approach

for inspecting systematic changes that, by matching a generalized template against the codebase, summarizes similar changes and detects potential mistakes.

In summary, similarly to previous empirical research, this work investigates MCR-practices. Compared to our work, Beller *et al.* [22] manually analyzed only two OSS projects, which makes our work more generalizable. Differently by Beller *et al.* [22], we also validated and extended the taxonomy by surveying developers, discovering further unexplored MCR changes (see Section 3) influenced by new emerging development technologies (e.g., cloud-based technologies) and practices (e.g., continuous delivery). Moreover, differently from Beller *et al.* [22], we investigated, via content analysis of responses from survey participants, (i) the types of feedback developers usually accept/receive in MCR, (ii) the types of tools they need or envision to automate contemporary MCR practices/tasks, and (iii) the data to use and the recommendations to follow in building such tools. Finally, we also propose an automated tool to support MCR practices, which was inspired by the study participants' feedback.

6 Conclusions

This paper empirically investigated approaches and tools that, from a developer's point of view, are still needed to facilitate MCR activities. In a first step, we elicited a taxonomy, called `CRAM`, characterizing the most critical and recurrent change types in MCR by: (i) quantitatively and qualitatively analyzing code review changes in ten Java open-source projects; (ii) integrating an existing taxonomy from literature by Beller *et al.* [22] and (iii) conducting a survey with 52 developers to find missing change types in our taxonomy (`CRAM`), investigating also current developer's automation needs regarding newly emerged review changes and activities.

Results of our study indicate that `CRAM` captures code review changes that were not considered in previous taxonomies, and that most of them are related to the availability of new emerging technologies (e.g., cloud-based technologies) and practices (e.g., Continuous Delivery and Continuous Integration). In addition, our study provides valuable insights on ways MCR activities can be facilitated by novel tools and approaches.

As future work, we plan to experiment with further automated approaches supporting MCR activities, by considering other developers' insights found in our empirical investigation.

Acknowledgements The authors would like to thank Antonello Reale (Fifth Beat[6]) and all developers and researchers that participated to the qualitative investigation of this study. We also thank all reviewers and the editors for the useful feedback, addressing their comments allowed us to make the contributions of this work more coherent and complete.

[6] https://fifthbeat.com/

References

1. Aacceleo. https://www.eclipse.org/acceleo. Accessed: 2018-08-01
2. Amalgam. http://www.eclipse.org/modeling/amalgam/. Accessed: 2018-08-01
3. CheckStyle. http://checkstyle.sourceforge.net. Accessed: 2014-10-27
4. Eclipe EGit. http://www.eclipse.org/egit/. Accessed: 2018-08-01
5. Eclipse BPEL. http://www.eclipse.org/bpel/. Accessed: 2018-08-01
6. Eclipse Cbi. https://git.eclipse.org/r/cbi/org.eclipse.cbi. Accessed: 2018-08-01
7. Eclipse CDT. http://www.eclipse.org/cdt/. Accessed: 2018-08-01
8. Eclipse PDE. http://www.eclipse.org/pde/. Accessed: 2018-08-01
9. Egit-training. https://git.eclipse.org/r/sandbox/egit-training. Accessed: 2018-08-01
10. Gerrit. https://code.google.com/p/gerrit/. Accessed: 2014-10-27
11. JGit. http://www.eclipse.org/jgit/. Accessed: 2018-08-01
12. M2e. https://git.eclipse.org/r/m2e/m2e-core. Accessed: 2018-08-01
13. PMD. http://pmd.sourceforge.net. Accessed: 2014-10-27
14. Bacchelli, A., Bird, C.: Expectations, outcomes, and challenges of modern code review. In: Proceedings of the International Conference on Software Engineering (ICSE), pp. 712–721 (2013)
15. Balachandran, V.: Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation. In: 35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013, pp. 931–940 (2013). DOI 10.1109/ICSE.2013.6606642. URL https://doi.org/10.1109/ICSE.2013.6606642
16. Barnett, M., Bird, C., Brunet, J., Lahiri, S.K.: Helping developers help themselves: Automatic decomposition of code review changesets. In: 37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1, pp. 134–144 (2015)
17. Baum, T., Schneider, K., Bacchelli, A.: On the optimal order of reading source code changes for review. In: 2017 IEEE International Conference on Software Maintenance and Evolution, ICSME 2017, Shanghai, China, September 17-22, 2017, pp. 329–340 (2017). DOI 10.1109/ICSME.2017.28. URL https://doi.org/10.1109/ICSME.2017.28
18. Bavota, G., Qusef, A., Oliveto, R., Lucia, A.D., Binkley, D.W.: Are test smells really harmful? an empirical study. Empirical Software Engineering **20**(4), 1052–1094 (2015)
19. Bavota, G., Russo, B.: Four eyes are better than two: On the impact of code reviews on software quality. In: 2015 IEEE International Conference on Software Maintenance and Evolution, ICSME 2015, Bremen, Germany, September 29 - October 1, 2015, pp. 81–90 (2015)
20. Baysal, O., Kononenko, O., Holmes, R., Godfrey, M.W.: The secret life of patches: A firefox case study. In: Proceedings of the Working Conference on Reverse Engineering (WCRE), pp. 447–455 (2012)
21. Baysal, O., Kononenko, O., Holmes, R., Godfrey, M.W.: Investigating technical and non-technical factors influencing modern code review. Empirical Software Engineering **21**(3), 932–959 (2016)
22. Beller, M., Bacchelli, A., Zaidman, A., Jürgens, E.: Modern code reviews in open-source projects: which problems do they fix? In: 11th Working Conference on Mining Software Repositories, MSR 2014, Proceedings, May 31 - June 1, 2014, Hyderabad, India, pp. 202–211 (2014)
23. Beller, M., Bholanath, R., McIntosh, S., Zaidman, A.: Analyzing the state of static analysis: A large-scale evaluation in open source software. In: IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016, Suita, Osaka, Japan, March 14-18, 2016 - Volume 1, pp. 470–481. IEEE Computer Society (2016)
24. Bosu, A., Carver, J.C., Bird, C., Orbeck, J.D., Chockley, C.: Process aspects and social dynamics of contemporary code review: Insights from open source development and industrial practice at microsoft. IEEE Trans. Software Eng. **43**(1), 56–75 (2017). DOI 10.1109/TSE.2016.2576451. URL https://doi.org/10.1109/TSE.2016.2576451

25. Bosu, A., Greiler, M., Bird, C.: Characteristics of useful code reviews: An empirical study at microsoft. In: 12th IEEE/ACM Working Conference on Mining Software Repositories, MSR 2015, Florence, Italy, May 16-17, 2015, pp. 146–156 (2015)
26. Chatley, R., Jones, L.: Diggit: Automated code review via software repository mining. In: 25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campobasso, Italy, March 20-23, 2018, pp. 567–571 (2018)
27. De Lucia, A., Di Penta, M., Oliveto, R., Panichella, A., Panichella, S.: Using IR methods for labeling source code artifacts: Is it worthwhile? In: IEEE 20th International Conference on Program Comprehension, ICPC 2012, Passau, Germany, June 11-13, 2012, pp. 193–202 (2012)
28. Deursen, A., Moonen, L., Bergh, A., Kok, G.: Refactoring test code. In: Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes (XP2001), pp. 92–95 (2001)
29. Di Penta, M., Cerulo, L., Aversano, L.: The life and death of statically detected vulnerabilities: An empirical study. Information & Software Technology **51**(10), 1469–1484 (2009)
30. Dig, D., Johnson, R.E.: How do apis evolve? A story of refactoring. Journal of Software Maintenance **18**(2), 83–107 (2006). DOI 10.1002/smr.328. URL `https://doi.org/10.1002/smr.328`
31. Duvall, P., Matyas, S.M., Glover, A.: Continuous Integration: Improving Software Quality and Reducing Risk. Addison-Wesley (2007)
32. Duvall, P.M.: Continuous integration. patterns and antipatterns. DZone refcard #84 (2010). URL `http://bit.ly/l8rfVS`
33. Efstathiou, V., Spinellis, D.: Code review comments: language matters. In: Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results, ICSE (NIER) 2018, Gothenburg, Sweden, May 27 - June 03, 2018, pp. 69–72 (2018)
34. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for java. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), pp. 234–245 (2002)
35. Fluri, B., Gall, H.C.: Classifying change types for qualifying change couplings. In: 14th International Conference on Program Comprehension (ICPC 2006), 14-16 June 2006, Athens, Greece, pp. 35–45. IEEE Computer Society (2006). DOI 10.1109/ICPC.2006.16. URL `https://doi.org/10.1109/ICPC.2006.16`
36. Fowler, M.: Refactoring: Improving the design of existing code. In: Extreme Programming and Agile Methods - XP/Agile Universe 2002, Second XP Universe and First Agile Universe Conference Chicago, IL, USA, August, 2002, p. 256 (2002)
37. Fusaro, P., Lanubile, F., Visaggio, G.: A replicated experiment to assess requirements inspection techniques. Empirical Software Engineering **2**(1), 39–57 (1997)
38. Germán, D.M., Robles, G., Poo-Caamaño, G., Yang, X., Iida, H., Inoue, K.: "was my contribution fairly reviewed?": a framework to study the perception of fairness in modern code reviews. In: Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018, pp. 523–534 (2018). DOI 10.1145/3180155.3180217. URL `http://doi.acm.org/10.1145/3180155.3180217`
39. Gibbs, L., Kealy, M., Willis, K., Green, J., Welch, N., Daly, J.: What have sampling and data collection got to do with good qualitative research? Australian and New Zealand journal of public health **31**(6), 540–544 (2007)
40. Grano, G., Ciurumelea, A., Panichella, S., Palomba, F., Gall, H.C.: Exploring the integration of user feedback in automated testing of android applications. In: 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 72–83 (2018)
41. Haiduc, S., Aponte, J., Moreno, L., Marcus, A.: On the use of automated text summarization techniques for summarizing source code. In: 17th Working Conference on Reverse Engineering (WCRE), October 2010, Beverly, MA, USA, pp. 35–44 (2010)
42. Hannebauer, C., Patalas, M., Stünkel, S., Gruhn, V.: Automatically recommending code reviewers based on their expertise: an empirical comparison. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016, pp. 99–110 (2016)

43. Henley, A.Z., Muçlu, K., Christakis, M., Fleming, S.D., Bird, C.: Cfar: A tool to increase communication, productivity, and review quality in collaborative code reviews. In: R.L. Mandryk, M. Hancock, M. Perry, A.L. Cox (eds.) Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems, CHI 2018, Montreal, QC, Canada, April 21-26, 2018, p. 157. ACM (2018). DOI 10.1145/3173574.3173731. URL https://doi.org/10.1145/3173574.3173731
44. Hill, E., Pollock, L., Vijay-Shanker, K.: Automatically capturing source code context of nl-queries for software maintenance and reuse. In: International Conference on Software Engineering (ICSE), pp. 232–242. IEEE (2009)
45. Höst, M., Johansson, C.: Evaluation of code review methods through interviews and experimentation. Journal of Systems and Software **52**(2-3), 113–120 (2000)
46. Humble, J., Farley, D.: Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation, 1st edn. Addison-Wesley Professional (2010)
47. Kemerer, C.F., Paulk, M.C.: The impact of design and code reviews on software quality: An empirical study based on PSP data. IEEE Trans. Software Eng. **35**(4), 534–550 (2009). DOI 10.1109/TSE.2009.27. URL https://doi.org/10.1109/TSE.2009.27
48. Khalid, H., Shihab, E., Nagappan, M., Hassan, A.E.: What do mobile app users complain about? IEEE Software **32**(3), 70–77 (2015)
49. Kim, S., Ernst, M.D.: Which warnings should I fix first? In: Proceedings of the joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE), pp. 45–54 (2007)
50. Kim, S., Pan, K., Jr., E.J.W.: Micro pattern evolution. In: S. Diehl, H.C. Gall, A.E. Hassan (eds.) Proceedings of the 2006 International Workshop on Mining Software Repositories, MSR 2006, Shanghai, China, May 22-23, 2006, pp. 40–46. ACM (2006). DOI 10.1145/1137983.1137995. URL https://doi.org/10.1145/1137983.1137995
51. Kononenko, O., Baysal, O., Godfrey, M.W.: Code review quality: how developers see it. In: Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016, pp. 1028–1038 (2016)
52. Kononenko, O., Baysal, O., Guerrouj, L., Cao, Y., Godfrey, M.W.: Investigating code review quality: Do people and participation matter? In: 2015 IEEE International Conference on Software Maintenance and Evolution, ICSME 2015, Bremen, Germany, September 29 - October 1, 2015, pp. 111–120 (2015)
53. Mäntylä, M., Lassenius, C.: What types of defects are really discovered in code reviews? IEEE Trans. Software Eng. **35**(3), 430–448 (2009). DOI 10.1109/TSE.2008.71. URL https://doi.org/10.1109/TSE.2008.71
54. Mäntylä, M., Vanhanen, J., Lassenius, C.: A taxonomy and an initial empirical study of bad smells in code. In: 19th International Conference on Software Maintenance (ICSM, Amsterdam, The Netherlands, pp. 381–384 (2003)
55. Mäntylä, M.V., Lassenius, C.: What types of defects are really discovered in code reviews? IEEE Transactions on Software Engineering (TSE) **35**(3), 430–448 (2009)
56. Martin, D., Panichella, S.: The cloudification perspectives of search-based software testing. In: A. Gorla, J.M. Rojas (eds.) Proceedings of the 12th International Workshop on Search-Based Software Testing, SBST@ICSE 2019, Montreal, QC, Canada, May 27, 2019, pp. 5–6. IEEE / ACM (2019). DOI 10.1109/SBST.2019.00009. URL https://doi.org/10.1109/SBST.2019.00009
57. McBurney, P.W., McMillan, C.: Automatic documentation generation via source code summarization of method context. In: Proceedings of the International Conference on Program Comprehension (ICPC), pp. 279–290. ACM (2014)
58. McIntosh, S., Kamei, Y., Adams, B., Hassan, A.E.: The impact of code review coverage and code review participation on software quality: a case study of the qt, vtk, and ITK projects. In: Proceedings of the Working Conference on Mining Software Repositories (MSR), pp. 192–201 (2014)
59. Menarini, M., Yan, Y., Griswold, W.G.: Semantics-assisted code review: an efficient toolchain and a user study. In: Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017, pp. 554–565 (2017)

60. Meszaros, G.: Xunit test patterns and smells: improving the ROI of test code. In: Companion to the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, SPLASH/OOPSLA 2010, October, Reno/Tahoe, Nevada, USA, pp. 299–300 (2010)
61. Moha, N., Guéhéneuc, Y., Duchien, L., Meur, A.L.: DECOR: A method for the specification and detection of code and design smells. IEEE Trans. Software Eng. **36**(1), 20–36 (2010)
62. Moha, N., Gueheneuc, Y.G., Duchien, L., Le Meur, A.F.: Decor: A method for the specification and detection of code and design smells. IEEE Transactions on Software Engineering **36**(1), 20–36 (2010)
63. Moreno, L., Aponte, J., Sridhara, G., Marcus, A., Pollock, L., Vijay-Shanker, K.: Automatic generation of natural language summaries for java classes. In: International Conference on Program Comprehension (ICPC), pp. 23–32. IEEE (2013)
64. Moreno, L., Marcus, A.: Automatic software summarization: the state of the art. In: 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017, pp. 511–512 (2017)
65. Moreno, L., Marcus, A.: Automatic software summarization: the state of the art. In: Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018, pp. 530–531 (2018)
66. Nurolahzade, M., Nasehi, S.M., Khandkar, S.H., Rawal, S.: The role of patch review in software evolution: An analysis of the mozilla firefox. In: Proceedings of the Joint International and Annual ERCIM Workshops on Principles of Software Evolution (IWPSE) and Software Evolution (Evol) Workshops, pp. 9–18 (2009)
67. Ouni, A., Kula, R.G., Inoue, K.: Search-based peer reviewers recommendation in modern code review. In: 2016 IEEE International Conference on Software Maintenance and Evolution, ICSME 2016, Raleigh, NC, USA, October 2-7, 2016, pp. 367–377 (2016). DOI 10.1109/ICSME.2016.65. URL `https://doi.org/10.1109/ICSME.2016.65`
68. Paixão, M., Krinke, J., Han, D., Harman, M.: CROP: linking code reviews to source code changes. In: Proceedings of the 15th International Conference on Mining Software Repositories, MSR 2018, Gothenburg, Sweden, May 28-29, 2018, pp. 46–49 (2018)
69. Palomba, F., Panichella, A., Lucia, A.D., Oliveto, R., Zaidman, A.: A textual-based technique for smell detection. In: 24th International Conference on Program Comprehension, Austin, TX, USA, May, 2016, pp. 1–10 (2016)
70. Panichella, S.: Summarization techniques for code, change, testing, and user feedback (invited paper). In: C. Artho, R. Ramler (eds.) 2018 IEEE Workshop on Validation, Analysis and Evolution of Software Tests, VST@SANER 2018, Campobasso, Italy, March 20, 2018, pp. 1–5. IEEE (2018). DOI 10.1109/VST.2018.8327148. URL `https://doi.org/10.1109/VST.2018.8327148`
71. Panichella, S., Arnaoudova, V., Penta, M.D., Antoniol, G.: Would static analysis tools help developers with code reviews? In: 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering, SANER 2015, Montreal, QC, Canada, March 2-6, 2015, pp. 161–170 (2015). DOI 10.1109/SANER.2015.7081826. URL `https://doi.org/10.1109/SANER.2015.7081826`
72. Panichella, S., Di Sorbo, A., Guzman, E., Visaggio, C.A., Canfora, G., Gall, H.C.: How can i improve my app? classifying user reviews for software maintenance and evolution. In: 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 281–290 (2015)
73. Panichella, S., Panichella, A., Beller, M., Zaidman, A., Gall, H.C.: The impact of test case summaries on bug fixing performance: an empirical investigation. In: 38th International Conference on Software Engineering, Austin, TX, USA, May, 2016, pp. 547–558 (2016)
74. Parnas, D.L., Weiss, D.M.: Active design reviews: Principles and practices. In: Proceedings, 8th International Conference on Software Engineering, London, UK, August 28-30, 1985., pp. 132–136 (1985)
75. Porter, A.A., Votta, L.G.: Comparing detection methods for software requirements inspections: A replication using professional subjects. Empirical Software Engineering **3**(4), 355–379 (1998)

76. Rahman, M.M., Roy, C.K., Kula, R.G.: Predicting usefulness of code review comments using textual features and developer experience. In: Proceedings of the 14th International Conference on Mining Software Repositories, MSR 2017, Buenos Aires, Argentina, May 20-28, 2017, pp. 215–226 (2017)
77. Rigby, P.C.: Understanding open source software peer review: Review processes, parameters and statistical models, and underlying behaviours and mechanisms. Ph.D. thesis, University of Victoria, BC, Canada (2011)
78. Rigby, P.C., German, D.M.: A preliminary examination of code review processes in open source projects. Tech. Rep. DCS-305-IR, University of Victoria (2006)
79. Rigby, P.C., German, D.M., Storey, M.D.: Open source software peer review practices: a case study of the apache server. In: Proceedings of the International Conference on Software Engineering (ICSE), pp. 541–550 (2008)
80. Savor, T., Douglas, M., Gentili, M., Williams, L., Beck, K., Stumm, M.: Continuous deployment at facebook and OANDA. In: Companion proceedings of the 38th International Conference on Software Engineering (ICSE Companion), pp. 21–30
81. Shi, S., Li, M., Lo, D., Thung, F., Huo, X.: Automatic code review by learning the revision of source code. In: The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019, pp. 4910–4917. AAAI Press (2019). DOI 10.1609/aaai.v33i01.33014910. URL https://doi.org/10.1609/aaai.v33i01.33014910
82. Singh, D., Sekar, V.R., Stolee, K.T., Johnson, B.: Evaluating how static analysis tools can reduce code review effort. In: A.Z. Henley, P. Rogers, A. Sarma (eds.) 2017 IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2017, Raleigh, NC, USA, October 11-14, 2017, pp. 101–105. IEEE Computer Society (2017). DOI 10.1109/VLHCC.2017.8103456. URL https://doi.org/10.1109/VLHCC.2017.8103456
83. Spadini, D., Aniche, M.F., Storey, M.D., Bruntink, M., Bacchelli, A.: When testing meets code review: why and how developers review tests. In: Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018, pp. 677–687 (2018)
84. Sridhara, G., Hill, E., Muppaneni, D., Pollock, L., Vijay-Shanker, K.: Towards automatically generating summary comments for java methods. In: International Conference on Automated Software Engineering, pp. 43–52 (2010)
85. Thongtanunam, P., Tantithamthavorn, C., Kula, R.G., Yoshida, N., Iida, H., Matsumoto, K.: Who should review my code? A file location-based code-reviewer recommendation approach for modern code review. In: 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering, SANER 2015, Montreal, QC, Canada, March 2-6, 2015, pp. 141–150 (2015)
86. Thung, F., Lucia, Lo, D., Jiang, L., Rahman, F., Devanbu, P.T.: To what extent could we detect field defects? an empirical study of false negatives in static bug finding tools. In: Proceedings of the International Conference on Automated Software Engineering (ASE), pp. 50–59 (2012)
87. Tsantalis, N., Chatzigeorgiou, A.: Identification of move method refactoring opportunities. IEEE Trans. Software Eng. pp. 347–367 (2009)
88. Vassallo, C., Panichella, S., Palomba, F., Proksch, S., Zaidman, A., Gall, H.C.: Context is king: The developer perspective on the usage of static analysis tools. In: 25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campobasso, Italy, March 20-23, 2018, pp. 38–49 (2018)
89. Vendome, C., Germán, D.M., Penta, M.D., Bavota, G., Vásquez, M.L., Poshyvanyk, D.: To distribute or not to distribute?: why licensing bugs matter. In: Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018, pp. 268–279 (2018)
90. Wagner, H.R.: The discovery of grounded theory: Strategies for qualitative research. Social Forces **46**(4), 555 (1968)
91. Wagner, S., Jurjens, J., Koller, C., Trischberger, P.: Comparing bug finding tools with reviews and tests. In: Proceedings of the 17th IFIP TC6/WG 6.1 International Conference on Testing of Communicating Systems, pp. 40–55 (2005)

92. Weißgerber, P., Neu, D., Diehl, S.: Small patches get in! In: Proceedings of the Working Conference on Mining Software Repositories (MSR), pp. 67–76 (2008)
93. Zampetti, F., Scalabrino, S., Oliveto, R., Canfora, G., Di Penta, M.: How open source projects use static code analysis tools in continuous integration pipelines. In: Proceedings of the 14th International Conference on Mining Software Repositories, pp. 334–344. IEEE Press (2017)
94. Zampetti Fiorella, V.C.P.S.C.G.G.H.D.P.M.: An empirical characterization of bad practices in continuous integration. Empirical Software Engineering (2020)
95. Zanjani, M.B., Kagdi, H.H., Bird, C.: Automatically recommending peer reviewers in modern code review. IEEE Trans. Software Eng. **42**(6), 530–543 (2016). DOI 10.1109/TSE.2015.2500238. URL `https://doi.org/10.1109/TSE.2015.2500238`
96. Zhang, T., Song, M., Pinedo, J., Kim, M.: Interactive code review for systematic changes. In: 37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1, pp. 111–122 (2015)
97. Zhou, Y., Gu, R., Chen, T., Huang, Z., Panichella, S., Gall, H.C.: Analyzing apis documentation and code to detect directive defects. In: Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017, pp. 27–37 (2017). DOI 10.1109/ICSE.2017.11. URL `https://doi.org/10.1109/ICSE.2017.11`
98. Zhou, Y., Su, Y., Chen, T., Huang, Z., Gall, H.C., Panichella, S.: User review-based change file localization for mobile applications. IEEE Transactions on Software Engineering pp. 1–1 (2020)
99. Zhou, Y., Wang, C., Yan, X., Chen, T., Panichella, S., Gall, H.C.: Automatic detection and repair recommendation of directive defects in java api documentation. IEEE Transactions on Software Engineering pp. 1–1 (2018)

7 Appendix

Table 20: **C**ode **R**eview ch**A**nges **M**odel (CRAM) - Part I.

Artifact	Activity	Category	Topic	Detailed Change
Production & Test Code (Modification occurring in production and test code)	**Maintainability & Perfective Maintenance**	**Documentation (D)**	**- Textual Documentation**: Issues concerning the documentation through textual representation, such as naming of classes, method, variables. This also includes license headers, typos in either inline comments or Javadoc	**(D.1) - Naming**: Problems relating to software element (e.g., methods, classes, variables, etc) names that do not conform to the naming policy of the project. **(D.2) - Comments**: Explanations of complex code fragments, classes, methods. Issues include wrongly placed comments, missing comments, missing or wrong Javadoc etc. **(D.3) - License Header**: Issues regarding missing or wrong license headers inside source-files. **(D.4) - Typos**: Spelling mistakes in the documentation
			- Language Supported Documentation: Documentation through statements/elements that the programming language offers (e.g., java public modifier to document that it is accessible from the outside)	**(D.6) - Immutability**: Not declaring a variable to be immutable when it should have been or declaring it immutable when it should have not been **(D.7) - Visibility (Modifiers)**: Software element (e.g. method, variable) has too much or too restricted visibility -
			Style (S)	**(S.1) - Brackets & Braces**: e.g., single statement after a conditional branch **(S.2) - Indentation**: consistent indentation of the code **(S.3.) - Blank Lines**: excess of blank lines or too few blank lines or wrong split of lines **(S.4) - Long Lines**: code statement too long, over a specific amount of characters **(S.5) - Whitespace Usage**: usages of blank spaces in the code **(S.6) - Grouping**: grouping of methods with related functionality or adding class variables at the beginning of the class **(S.7) - Commented out code**: remove code that is commented out (also TODO and FIXME)
		Structure (STR)	**- Re-implementation**: Structural defects require an alternative implementation method. For example, replacing the program's array data structure with a vector and knowing the existence of prebuilt functionality that could be used instead of a self-programmed implementation would be considered a solution approach defect. Therefore, solution approach defects are not about re-organizing existing code but rethinking the current solution and implementing it in a different way.	**(STR.1) - Semantic Duplication**: Code structures that have a similar intention but are implemented syntactically different **(STR.2) - Semantic Dead Code**: Code fragments that are executed, but they do not serve any meaningful purpose and/or have no effect on the result **(STR.3) - Change Function**: Change function call to another function because it uses old or deprecated functions **(STR.4) - Standard Coding Conventions**: Use exceptions for error messaging instead of return values, use predefined constants instead of magic numbers, built-in data structures instead of own implementation etc. **(STR.5) - New Functionality**: new functionality to ensure evolvability, e.g., create new classes, methods to make code more maintainable **(STR.6) - Strings (Wording)**: Issues regarding contents of strings, badly composed strings **(STR.7) - Logging**: Add the ability to methods for logging results or errors **(STR.8) - Testing**: Issues regarding test coverage, wrong/inappropriate tests, additional tests etc.
			- Organization: Defects that can be fixed by applying structural modifications to the software. Moving a piece of functionality from module A to module B is a possible strategy for this.	**(STR.9) - Imports**: Issues with wrong or missing or unused import statements **(STR.10) - Move Functionality**: move functions, part of functions, or other functional elements to a different class, file, or module **(STR.11) - Long Sub Routine**: split long and complex functions into multiple functions **(STR.12) - Dead Code**: remove code that is never reached and executed **(STR.13) - Duplication / Redundant Code**: remove duplicate code or code that is not used **(STR.14) - Complex Code / Simplification**: restructure or rewrite implementation to make it more understandable **(STR.15) - Statement Issue**: splitting, combining or otherwise reorganizing a statement inside a function **(STR.16) - Consistency**: Means the need to keep code consistent in a sense that similar code elements operate in a similar fashion and are more or less symmetrical. For example, similar tasks in similar classes should have similar implementations **(STR.17) - Architectural changes**: code reviews often result in a change to the system architecture, like splitting an interface into two distinct interfaces, introducing abstractions, or the inclusion of design patterns
	Functionality/ Corrective Maintenance		**Interface (I)**	**(I.1) - Function Call**: call to another part of system or library is incorrect or missing **(I.2) - Parameter**: function call or other interaction has incorrect or missing parameters
			Logic (L)	**(L.1) - Compare**: mistake in a comparison statement **(L.2) - Computation**: computations produce incorrect results **(L.3) - Wrong Location**: correct operation is performed, but it is done too soon or too late **(L.4) - Algorithm/Performance**: inefficient algorithm is used
			Resource (R)	**(R.1) - Variable Initialization**: Variables are left uninitialized prior to use. Uninitialized variables may contain any value and using such variable for comparison or calculation produces arbitrary results. **(R.2) - Memory Management**: Mistake is made in handling the system memory. **(R.3) - Data & Resource Manipulation**: Defects related to manipulating or releasing data or other resources. **(R.4) - Security**: Issues related to the application's/software's security aspects **(R.5) - Concurrency**: Issues regarding concurrency
			Check (C)	**(C.1) - Check Function**: when in a function-call is also a need to check that the value returned is valid and that no error occurred **(C.2) - Check Variable**: there is a need to check variable **(C.3) - Check User Input**: the need to validate user input
			Larger Defects (LD)	**(LD.1) - Completeness**: partially implemented feature **(LD.2) - GUI**: Defects in the user interface code relating to the consistency of the user-interface, and to the options made possible to the user in each situation. **(LD.3) - Check outside code / Domino Effects**: Defects that required that part of the application code that was not under review to be checked, as it was likely to contain incorrect code based on the current review.

Table 21: **C**ode **R**eview ch**A**nges **M**odel (CRAM) - **Part II**.

Artifact	Activity
Other Changes Changes not typically found in source-code files (.java, .py, .cpp etc.) which are nonetheless essential to the runtime of a project	**(O.1) Commit Message**: Changes in the commit message of a submitted patch. Mostly related to wrong description of the change or not capturing all changes.
	(O.2) Continuous Integration / Continuous Delivery configurations: Changes to configuration files concerning the Continuous Integration or Continuous Delivery pipeline/setup.
	(O.3) Automated Static Analysis Tools configurations: Changes in the configuration of Linters, Checkers, Recommenders used in the project (e.g., Checkstyle, PMD, FindBugs etc.)
	(O.4) Language or Framework specific: Changes to files native to the used programming language. For example MANIFEST for Java.
	(O.5) External Software Documentation: Changes to the external Software Documentation files
	(O.6) Runtime Configurations: docker-configs, ansible playbooks, delivery configs etc.
	(O.7) Other: Includes changes to XML, Scripts, README files, HTML files and Version Control