Information Needs in Contemporary Code Review

LUCA PASCARELLA, Delft University of Technology, The Netherlands
DAVIDE SPADINI, Software Improvement Group, The Netherlands
FABIO PALOMBA, University of Zurich, Switzerland
MAGIEL BRUNTINK, Software Improvement Group, The Netherlands
ALBERTO BACCHELLI, University of Zurich, Switzerland

Contemporary code review is a widespread practice used by software engineers to maintain high software quality and share project knowledge. However, conducting proper code review takes time and developers often have limited time for review. In this paper, we aim at investigating the information that reviewers need to conduct a proper code review, to better understand this process and how research and tool support can make developers become more effective and efficient reviewers.

Previous work has provided evidence that a successful code review process is one in which reviewers and authors actively participate and collaborate. In these cases, the threads of discussions that are saved by code review tools are a precious source of information that can be later exploited for research and practice. In this paper, we focus on this source of information as a way to gather reliable data on the aforementioned reviewers' needs. We manually analyze 900 code review comments from three large open-source projects and organize them in categories by means of a card sort. Our results highlight the presence of seven high-level information needs, such as knowing the uses of methods and variables declared/modified in the code under review. Based on these results we suggest ways in which future code review tools can better support collaboration and the reviewing task. Preprint [https://doi.org/10.5281/zenodo.1405894]. Data and Materials [https://doi.org/10.5281/zenodo.1405902].

CCS Concepts: • **Software and its engineering** → *Software verification and validation*;

Additional Key Words and Phrases: code review; information needs; mining software repositories

ACM Reference Format:
Luca Pascarella, Davide Spadini, Fabio Palomba, Magiel Bruntink, and Alberto Bacchelli. 2018. Information Needs in Contemporary Code Review. *Proceedings of the ACM on Human-Computer Interaction* 2, CSCW, Article 135 (November 2018), 27 pages. https://doi.org/10.1145/3274404

1 INTRODUCTION

Peer code review is a well-established software engineering practice aimed at maintaining and promoting source code quality, as well as sustaining development community by means of knowledge transfer of design and implementation solutions applied by others [2]. Contemporary code review, also known as *Modern Code Review* (MCR) [2, 17], represents a lightweight process that is (1) informal, (2) tool-based, (3) asynchronous, and (4) focused on inspecting new proposed code changes rather than the whole codebase [49]. In a typical code review process, developers (the *reviewers*) other than the code change author manually inspect new committed changes to find as many issues as possible and provide feedback that needs to be addressed by the author of the change before the code is accepted and put into production [6].

Authors' addresses: Luca Pascarella, Delft University of Technology, Delft, The Netherlands, l.pascarella@tudelft.nl; Davide Spadini, Software Improvement Group, Amsterdam, The Netherlands, d.spadini@sig.eu; Fabio Palomba, University of Zurich, Zurich, Switzerland, palomba@ifi.uzh.ch; Magiel Bruntink, Software Improvement Group, Amsterdam, The Netherlands, m.bruntink@sig.eu; Alberto Bacchelli, University of Zurich, Zurich, Switzerland, bacchelli@ifi.uzh.ch.

Modern code review is a collaborative process in which reviewers and authors conduct an asynchronous online discussion to ensure that the proposed code changes are of sufficiently high quality [2] and fit the project's direction [26] before they are accepted. In code reviews, discussions range from low-level concerns (*e.g.*, variable naming and code style) up to high-level considerations (*e.g.*, fit within the scope of the project and future planning) and encompass both functional defects and evolutionary aspects [10]. For example a reviewer may ask questions regarding the structure of the changed code [57] or clarifications about the rationale behind some design decisions [55], another reviewer may respond or continue the thread of questions, and the author can answer the questions (*e.g.*, explaining the motivation that led to a change) and implement changes to the code to address the reviewers' remark.

Even though studies have shown that modern code review has the *potential* to support software quality and dependability [17, 39, 41], researchers have also provided strong empirical evidence that the outcome of this process is rather erratic and often unsatisfying or misaligned with the expectations of participants [2, 10, 37]. This erratic outcome is caused by the cognitive-demanding nature of reviewing [7], whose outcome mostly depends on the time and zeal of the involved reviewers [17].

Based on this, a large portion of the research efforts on tools and processes to help code reviewing is explicitly or implicitly based on the assumption that reducing the *cognitive load* of reviewers improves their code review performance [7]. In the current study, we continue on this line of better supporting the code review process through the reduction of reviewers' cognitive load. Specifically, *our goal is to investigate the information that reviewers need to conduct a proper code review.* We argue that—if this information would be available at hand—reviewers could focus their efforts and time on correctly evaluating and improving the code under review, rather than spending cognitive effort and time on collecting the missing information. By investigating reviewers' information needs, we can better understand the code review process, guide future research efforts, and envision how tool support can make developers become more effective and efficient reviewers.

To gather data about reviewers' information needs we turn to one of the collaborative aspects of code review, namely the discussions among participants that happen during this process. In fact, past research has shown that code review is more successful when there is a functioning collaboration among all the participants. For example, Rigby *et al.* reported that the efficiency and effectiveness of code reviews are most affected by the amount of review participation [50]; Kononenko *et al.* [34] showed that review participation metrics are associated with the quality of the code review process; McIntosh *et al.* found that a lack of review participation can have a negative impact on long-term software quality [39, 60]; and Spadini *et al.* studied review participation in production and test files, presenting a set of identified obstacles limiting the review of code [54]. For this reason, from code review communication, we expect to gather evidence of reviewers' information needs that are solved through the collaborative discussion among the participants.

To that end, we consider three large open-source software projects and manually analyze 900 code review discussion threads that started from a reviewer's question. We focus on what kind of questions are asked in these comments and their answers. As shown in previous research [12, 14, 33, 56], such questions can implicitly represent the information needs of code reviewers. In addition, we conduct four semi-structured interviews with developers from the considered systems and one focus group with developers from a software quality consultancy firm, both to challenge our outcome and to discuss developers' perceptions. Better understanding what reviewers' information needs are can lead to reduced cognitive load for the reviewers, thus leading, in turn, to better and shorter reviews. Furthermore, knowing these needs helps driving the research community toward the definition of methodologies and tools able to properly support code reviewers when verifying newly submitted code changes.

Our analysis led to seven high-level information needs, such as knowing the uses of methods and variables declared/modified in the code under review, and their analysis in the code review lifecycle. Among our results, we found that the needs to know (1) whether a proposed alternative solution is valid and (2) whether the understanding of the reviewer about the code under review is correct are the most prominent ones. Moreover, all the reviewers' information needs are replied to within a median time of seven hours, thus pointing to the large time savings that can be achieved by addressing these needs through automated tools. Based on these results, we discuss how future code review tools can better support collaboration and the reviewing task.

2 BACKGROUND AND RELATED WORK

This section describes the basic components that form a modern code review as well as the literature related to information needs and code review participation.

2.1 Background: The code review process

Figure 1 depicts a code review (pertaining to the OpenStack project) done with a typical code review tool. Although this is one of the many available review tools, their functionalities are largely the same [65]. In the following we briefly describe each of the components of a review as provided by code review tools.

Code review tools provide an ID and a status (part ① in Figure 1) for each code review, which are used to track the code change and know whether it has been *merged* (*i.e.*, put into production) or *abandoned* (*i.e.*, it has been evaluated as not suitable for the project). Code review tools also allow the change author to include a textual description of the code change, with the aim to provide reviewers with more information on the rationale and behavior of the change. However, past research has provided evidence that the quality and level of detail of the descriptions that accompany code changes are often suboptimal [57], thus making it harder for reviewers to properly understand the code change through this support. The fact that the change description is often not optimal strengthens the importance of the goal of our study: An improved analysis of developers' needs in code review can provide benefits in terms of review quality [34].

The second component of a typical code review tool is a view on the technical meta-information on the change under review (part ② in Figure 1). This meta-information include author and committer of the code change, commit ID, parent commit ID, and change ID, which can be used to track the submitted change over the history of the project.

Part ③ of the tool in Figure 1 reports, instead, more information on who are the reviewers assigned for the inspection of the submitted code change, while part ④ lists the source code files modified in the commit (*i.e.*, the files on which the review will be focused).

Finally, part ⑤ is the core component of a code review tool and the one that involves most collaborative aspects. It reports the discussion that author and reviewers are having on the submitted code change. In particular, reviewers can ask clarifications or recommend improvements to the author, who can instead reply to the comments and propose alternative solutions. This mechanism is often accompanied by the upload of new versions of the code change (*i.e.*, revised patches or *iterations*), which lead to an iterative process until all the reviewers are satisfied with the change or decide to not include it into production. Figure 2 shows a different view that contains both reviews and authors comments. In this case, the involved developers discuss about a specific line of code, as opposed to Alice from the previous example who commented on the entire code change (Figure 1, end of part ⑤).

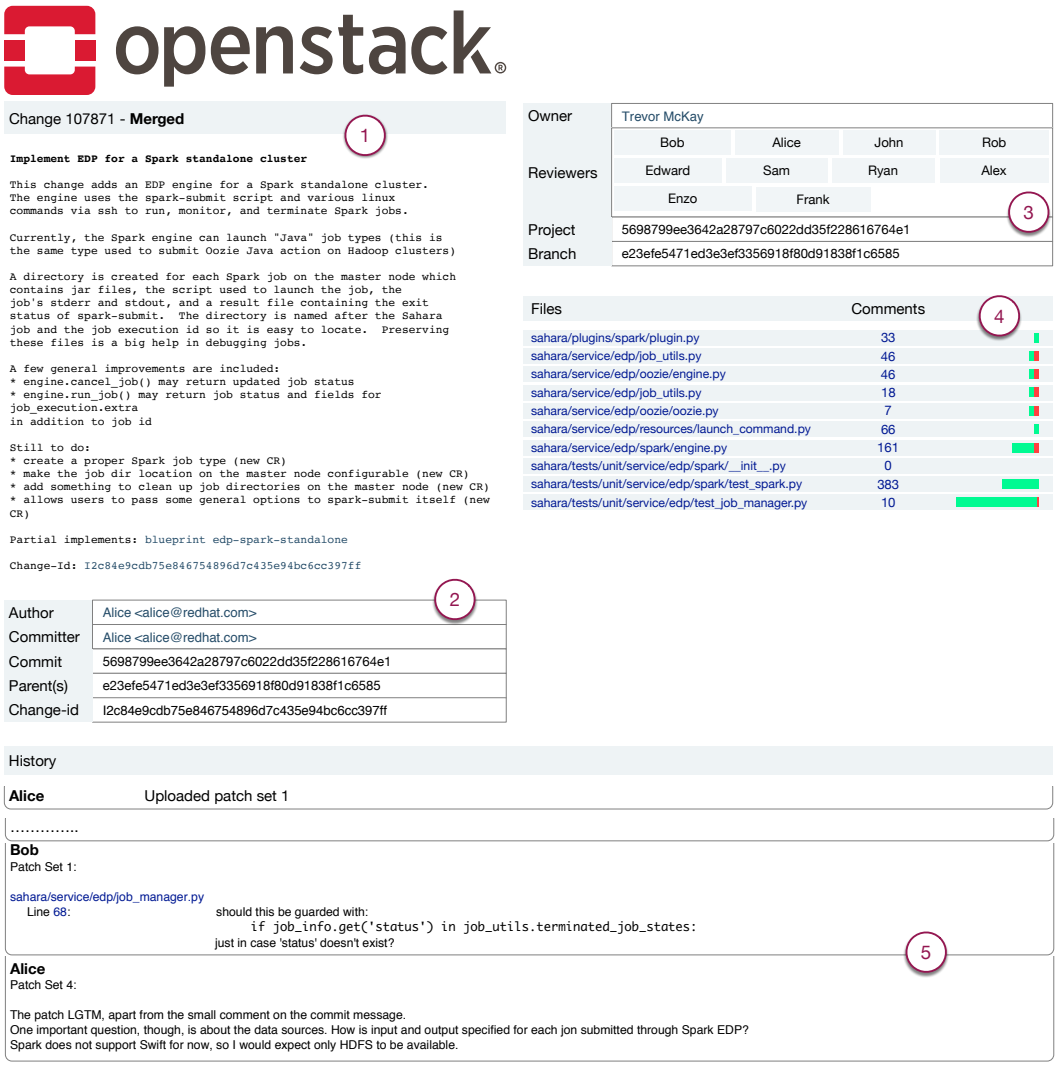

Fig. 1. Example of code review mined from Gerrit.

2.2 Related Work

Over the last decade the research community spent a considerable effort in studying code reviews (*e.g.*, [3, 10, 11, 17, 20, 32, 54]). In this section, we compare and contrast our work to previous research in two areas: first, we consider studies that investigate the information needs of developers in various contexts, then we analyze previous research that focused on code review discussion, participation, and time.

2.2.1 Information needs. Breu *et al.* [12] conducted a study—which has been a great inspiration to the current study we present here—on developers' information needs based on the analysis of collaboration among users of a software egineering tool (*i.e.*, issue tracking system). In their study, the authors have quantitatively and qualitatively analyzed the questions asked in a sample of 600

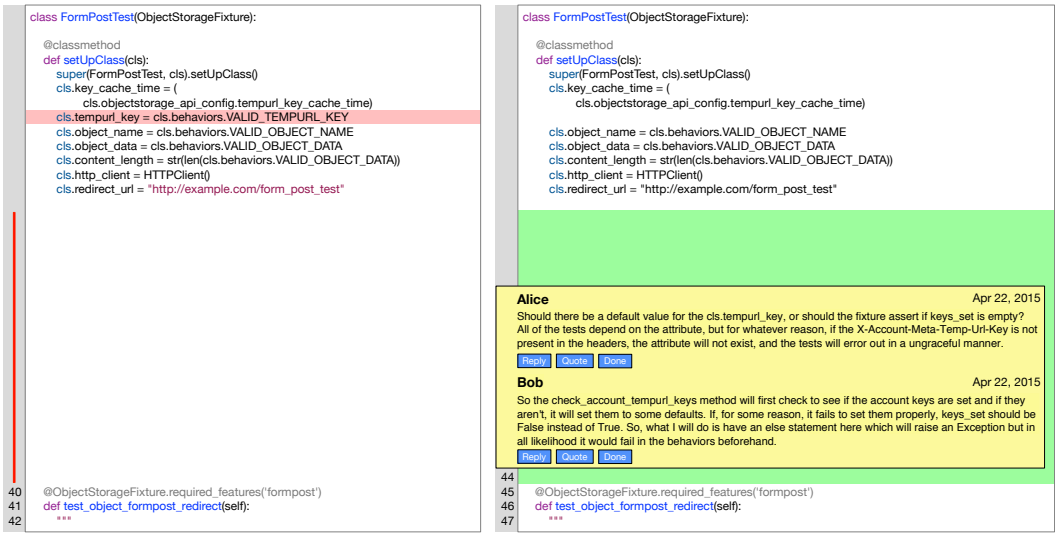

Fig. 2. Example of code review comments mined from Gerrit.

bug reports from two open-source projects, deriving a set of information needs in bug reports. The authors showed that active and ongoing participation were important factors needed for making progress on the bugs reported by users and they suggested a number of actions to be performed by the researchers and tool vendors in order to improve bug tracking systems.

Ko *et al.* [33] studied information needs of developers in collocated development teams. The authors observed the daily work of developers and noted the types of information desired. They identified 21 different information types in the collected data and discussed the implications of their findings for software designers and engineers. Buse and Zimmermann [14] analyzed developers' needs for software development analytics: to that end, they surveyed 110 developers and project managers. With the collected responses, the authors proposed several guidelines for analytics tools in software development.

Sillito *et al.* [53] conducted a qualitative study on the questions that programmers ask when performing change tasks. Their aim was to understand what information a programmer needs to know about a code base while performing a change task and also how they go about discovering that information. The authors categorized and described 44 different kinds of questions asked by the participants. Finally, Herbsleb *et al.* [29] analyzed the types of questions that get asked during design meetings in three organizations. They found that most questions concerned the project requirements, particularly what the software was supposed to do and, somewhat less frequently, scenarios of use. Moreover, they also discussed the implications of the study for design tools and methods.

The work we present in this paper is complementary with respect to the ones discussed so far: indeed, we aim at making a further step ahead investigating the information needs of developers that review code changes with the aim of deepening our understanding of the code review process and of leading to future research and tools to better support reviewers in conducting their tasks.

2.2.2 Code Review Participation and Time. Extensive work has been done by the software engineering research community in the context of code review participation. Abelein *et al.* [1] investigated the effects of user participation and involvement on system success and explored which methods

are available in literature, showing that it can have a significant correlation with system quality. Thongtanunam *et al.* [62] showed that reviewing expertise (which is approximated based on review participation) can reverse the association between authoring expertise and defect-proneness. Even more importantly, Rigby *et al.* [50] reported that the level of review participation is the most influential factor in the code review efficiency. Furthermore, several studies have suggested that patches should be reviewed by at least two developers to maximize the number of defects found during the review, while minimizing the reviewing workload on the development team [47, 49, 52, 61].

Thongtanunam *et al.* [60] showed that the number of participants that are involved with a review has a large relationship with the subsequent defect proneness of files in the Qt system: A file that is examined by more reviewers is less likely to have post-release defects. Bavota *et al.* [8] also found that the patches with low number of reviewers tend to have a higher chance of inducing new bug fixes. Moreover, McIntosh *et al.* [38, 39] measured review investment (i.e., the proportion of patches that are reviewed and the amount of participation) in a module and examined the impact that review coverage has on software quality. They found that patches with low review investment are undesirable and have a negative impact on code quality. In a study of code review practices at Google, Sadowski *et al.* [51] found that Google has refined its code review process over several years into an exceptionally lightweight one, which–in part–seems to contradict the aforementioned findings. Although the majority of changes at Google are small (a practice supported by most related work [48]), these changes mostly have one reviewer and have no comments other than the authorization to commit. Ebert *et al.* [23] made the first step in identifying the factors that may confuse reviewers since confusion is likely impacts the efficiency and effectiveness of code review. In particular, they manually analyzed 800 comments of code review of Android projects to identify those where the reviewers expressed confusion. Ebert *et al.* found that humans can reasonably identify confusion in code review comments and proposed the first binary classifier able to perform the same task automatically; they also observed that identifying confusion factors in inline comments is more challenging than general comments. Finally, Spadini *et al.* [54] analyzed more than 300,000 code reviews and interviewed 12 developers about their best practices when reviewing test files. As a result, they presented an overview of current code review practices, a set of identified obstacles limiting the review of test code, and a set of issues that developers would like to see improved in code review tools. Based on their findings, the authors proposed a series of recommendations and suggestions for the design of tools and future research.

Furthermore, previous research investigated how to make a code review shorter, hence making patches be accepted at a faster rate. For example, Jiang *et al.* [31] showed that patches developed by more experienced developers are more easily accepted, reviewed faster, and integrated more quickly. Additionally, authors stated that reviewing time is mainly impacted by submission time, the number of affected subsystems by the patch and the number of requested reviewers. Baysal *et al.* [9] showed that size of the patch or the part of the code base being modified are important factors that influenced the time required to review a patch, and are likely related to the technical complexity of a given change.

Recently, Chatley and Jones have proposed an approach aimed at enhancing the performance of code review [16]. The authors built Diggit to automatically generate code review comments about potentially missing changes and worrisome trends in the growth of size and complexity of the files under review. By deploying Diggit at a company, the authors found that the developers considered Diggit's comments as actionable and fixed them with an overall rate of 51%, thus indicating the potential of this approach in supporting code review performance.

Despite many studies showing that code review participation has a positive impact on the overall software development process (*i.e.*, number of post-release defects and time spent in reviewing),

none of these studies focused on what are the developers needs when performing code review. To fill this gap, our study aims at increasing our empirical knowledge on this field by mean of quantitative and qualitative research, with the potential of reducing the cognitive load of reviewers and the time needed for the review.

3 METHODOLOGY

The *goal* of our study is to increase our empirical knowledge on the reviewers' needs when performing code review tasks, with the *purpose* of identifying promising paths for future research on code review and the next generation of software engineering tools required to improve collaboration and coordination between source code authors and reviewers. The perspective is of researchers, who are interested in understanding what are the developers' needs in code review, therefore, they can more effectively devise new methodologies and techniques helping practitioners in promoting a collaborative environment in code review and reduce discussion overheads, thus improving the overall code review process.

Starting from a set of discussion threads between authors and reviewers, we start our investigation by eliciting the actual needs that reviewers have when performing code review:

- **RQ$_1$:** *What reviewers' needs can be captured from code review discussions?*

Specifically, we analyze the types of information that reviewers may need when reviewing, we compute the frequency of each need, and we challenge our outcome with developers from the analyzed systems and from an external company. Thus, we have three sub-questions:

- **RQ$_{1.1}$:** *What are the kinds of information code reviewers require?*
- **RQ$_{1.2}$:** *How often does each category of reviewers' needs occur?*
- **RQ$_{1.3}$:** *How do developers' perceive the identified needs?*

Once investigated reviewers' needs from the reviewer perspective, we further explore the collaborative aspects of code review by asking:

- **RQ$_2$:** *What is the role of reviewers' needs in the lifecycle of a code review?*

Specifically, we first analyze how much each reviewers' need is accompanied by a reply from the author of the code change: in other words, we aim at measuring how much authors of the code under review interact with reviewers to make the applied code change more comprehensible and ease the reviewing process. To complement this analysis, we evaluate the time required by authors to address a reviewer's need; also in this case, the goal is to measure the degree of collaboration between authors and reviewers. Finally, we aim at understanding whether and how the reviewers' information needs vary at different iterations of the code review process. For instance, we want to assess whether some specific needs arise at the beginning of the process (*e.g.*, because the reviewer does not have enough initial context to understand the code change) or, similarly, if clarification questions only appear at a later stage (*e.g.*, when only the last details are missing and the context is clear). Accordingly, we structure our second research question into three sub-questions:

- **RQ$_{2.1}$:** *What are the reviewers' information needs that attract more discussion?*
- **RQ$_{2.2}$:** *How long does it take to get a response to each reviewers' information need?*
- **RQ$_{2.3}$:** *How do the reviewers' information needs change over the code review process?*

The following subsections describe the method we use to answer our research questions.

3.1 Subject Systems

The first step leading to address our research goals is the selection of a set of code reviews that might be representative for understanding the reviewers' needs when reviewing source code changes. We rely on the well-known Gerrit platform,[1] which is a code review tool used by several major software projects. Specifically, Gerrit provides a simplified web based code review interface and a repository manager for Git.[2] From the open-source software systems using Gerrit, we select three: OpenStack,[3] Android,[4] and QT.[5] The selection was driven by two criteria: (i) These systems have been extensively studied in the context of code review research and have been shown to be highly representative of the types of code review done over open-source projects *et al.* [8, 38, 39]; (ii) these systems have a large number of active authors and reviewers over a long development history.

3.2 Gathering Code Review Threads

We automatically mine Gerrit data by relying on the publicly available APIs it provides. For the considered projects, the number of code reviews is over one million: this makes the manual analysis of all of them practically impossible. Thus, as done by Breu *et al.* [12], we select a random subset composed of 300 code reviews per project, for which we identify up to 1,800 messages (*i.e.*, we extract a total of 900 code review threads). Since we are interested in discussions, we take into account only closed code reviews by considering both merged and abandoned patches, while we do not consider recently opened or pending requests.

We detect reviewers' questions (considering the presence of a '?' sign) that start a discussion thread and we extract all the subsequent comments (made by the author, the reviewer, or other developers) in the whole thread.

The considered threads refer to both patch sets and inline discussions. To better illustrate the mining process of general discussions, Figure 1 reports a code review extracted from OpenStack. As shown in the bottom of the figure (part ⑤), author and reviewers opened a discussion on the performed change. Figure 2 shows a thread of discussion started at line level. In both cases, all the comments among the participants represent the types of discussion threads that we use to detect the information needs of reviewers.

For each identified *thread*, we store the following information:

- the *Gerrit id* of the code review;
- the *revision id* that identifies the patch set of a code review;
- the opening *question*, the *answers*, and the *source code* URL identifier of the change;
- the practitioner *role e.g.*, author or reviewer;
- the code review *status*, *i.e.*, whether it is merged or abandoned;
- the *size* of the thread counting the number of comments present into discussion;
- the creation and the update *time*.

We use the aforementioned pieces of information to answer our research questions as detailed in the following.

[1] https://www.gerritcodereview.com/
[2] https://git-scm.com/
[3] https://review.openstack.org/
[4] https://android-review.googlesource.com/
[5] https://codereview.qt-project.org

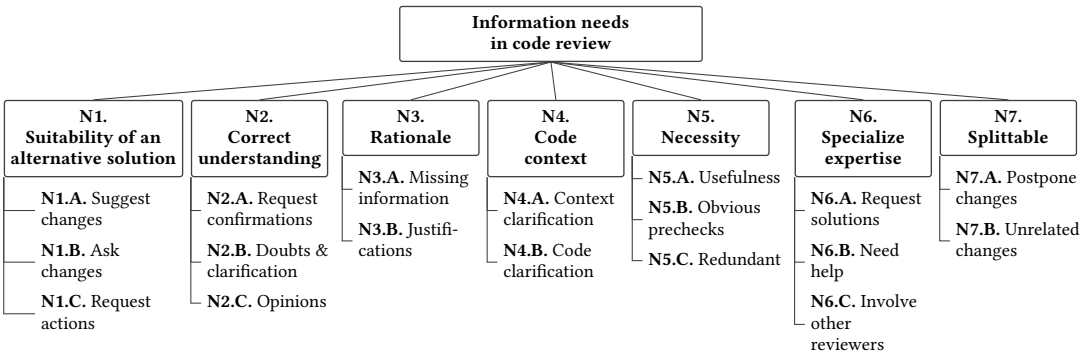

Fig. 3. The taxonomy of reviewers' information needs that emerged from our analysis

3.3 RQ_1 - Identifying the Reviewers' Needs from Code Review Discussions

To answer $\mathbf{RQ}_{1.1}$, we manually identify the reviewers' needs in code review by following a similar strategy as done in previous work on information needs [12, 14, 29, 33, 53]. Specifically, we perform a card sorting method [42] that involves all the authors of this paper (2 graduate students, 1 research associate, and 1 faculty member - who have at least seven years of programming experience). From now on, we refer to them as the *inspectors*. This method represents a well-established sorting technique that is used in information architecture with the aim of creating mental models and allowing the definition of taxonomies from input data [42]. In our case, it is used to organize code review threads into hierarchies and identify common themes. We rely on code review threads (*i.e.*, questions and answers) to better understand the meaning behind reviewers' questions that may implicitly define the reviewers' need. Finally, we apply an *open card sorting*: We have no predefined groups of reviewers' information needs, rather the needs emerge and evolve during the procedure. In our case, the process consists of the three iterative sessions described as follow.

Iteration 1: Initially, two inspectors (the first two authors of this paper) independently analyze an initial set of 100 OpenStack code review threads each. Then, they open a discussion on the reviewers' needs identified so far and try to reach a consensus on the names and types of the assigned categories. During the discussion, also the other two inspectors participate with the aim of validating the operations done in this iteration and suggesting possible improvements. As an output, this step provides a draft categorization of reviewers' needs.

Iteration 2: The first two inspectors re-categorize the 100 initial reviewers' needs according to the decisions taken during the discussion; then, they use the draft categorization as a basis for categorizing the remaining set of 200 code review threads belonging to OpenStack. This phase is used for both assessing the validity of the categories emerging from the first iteration (by confirming some of them and redefining others) and for discovering new categories. Once this iteration is completed, all the four inspectors open a new discussion aimed at refining the draft taxonomy, merging overlapping categories or better characterizing the existing ones. A second version of the taxonomy is produced.

Iteration 3: The first two inspectors re-categorize the 300 code review threads previously analyzed. Afterwards, the first inspector classifies the reviewers' needs concerning the two remaining considered systems. In doing so, the inspector tries to apply the defined categories on the set of code review threads of Android and QT. However, in cases where the inspector cannot directly apply the categories defined so far, the inspector reports such cases to the other inspectors so that a new discussion is opened. Unexpectedly this event did not eventually happen in practice;

in fact, the inspector could fit all the needs in the previously defined taxonomy, even when considering new systems. This result suggests that the categorization emerging from the first iterations reached a saturation [24], valid at least within the considered sample of threads.

Additional validation. To further check and confirm the operations performed by the first inspector, the third author of this paper—who was only involved in the discussion of the categories, but not in the assignment of the threads into categories—independently analyzed all the code review threads belonging to the three considered projects. The inspector classified all the 900 threads according to the second version of the taxonomy, as defined through iteration 2. The inspector did not need to define any further categories (thus suggesting that the taxonomy was exhaustive for the considered sample), however in six cases there were a disagreement between the category he assigned and the one assigned by the first author: as a consequence, the two authors opened a discussion in order to reach an agreement on the actual category to assign to those code review threads. Overall, the inter-rater agreement between this inspector and the first one, computed using the Krippendorff's k [35], was 98%.

Following this iterative process, we defined a hierarchical categorization composed of two layers. The top layer consists of *seven* categories, while the inner layer consists of 18 subcategories. Figure 3 depicts the identified top- and sub-categories. During the iterative sessions, ≈4% of the analyzed code review threads are discarded from our analysis since they do not contain useful information to understand the reviewers' needs. We assign these comments to four temporary sub-categories that indicate the reasons why they are discarded (e.g., they are noise or sarcastic comments), successively, we gathered these comments, in an additional top-category *Discarded*.

To answer $\mathbf{RQ}_{1.1}$, we report the reviewers' needs belonging to the categories identified in the top layer.

Subsequently, to answer $\mathbf{RQ}_{1.2}$ and understand how frequently each category of our needs appears, we verify how many information needs are assigned to each category. In this way, we can overview the most popular reviewers' needs when performing code review tasks. We answer this research question by presenting and discussing bar plots showing the frequency of each identified category.

To answer $\mathbf{RQ}_{1.3}$, we discuss the outcome of the previous sub-RQs with developers of the three considered systems and an external company. This gives us the opportunity to challenge our findings, triangulate our results, and complement our vision on the problem.

Table 1. Interviewees' experience (in years) and their working context.

ID	Years as developer	Years as reviewer	Working context
P_1	15	10	OpenStack
P_2	20	10	OpenStack
P_3	25	20	Qt
P_4	10	10	Android
FG_1	8	7	Company A
FG_2	10	10	Company A
FG_3	7	5	Company A

Interviews with reviewers from the subject systems. To organize the discussion with the developers of Android, OpenStack, and Qt, we use semi-structured interviews–a format that is often used in exploratory investigations to understand phenomena and seek new insights [68]. A crucial step in this analysis is represented by the recruitment strategy, *i.e.*, the way we select and

recruit participants for the semi-structured interviews. With the aim of gathering feedback and opinions from developers having a solid experience with the code review practices of the considered projects, we select only developers who had conducted at least 100 reviews[6] in their respective systems. Then, we randomly select 10 per system and invite them via email to participate in an online, video interview. Four experienced code reviewers accepted to be interviewed: two from OpenStack, one from Qt, and one from Android. The response rate achieved (17%) is in line with the one achieved by many previous works involving developers [43, 44, 66]. Table 1 summarizes the interviewees' demographic.

The interviews are conducted by the first two authors of this work via Skype. With the participants' consent all the interviews are recorded and transcribed for analysis. Each interview starts with general questions about programming and code reviews experience. In addition, we discuss whether the interviewees consider code reviews important, which tool they prefer, and generally how they conduct reviews. Overall, we organize the interview structure around five sections:

(1) General information regarding the developer;
(2) General perceptions on and experience with code review;
(3) Specific information needs during code review;
(4) Ranking of information needs during code review;
(5) Summary.

The main focus regarding the information needs is centered around points 3 and 4: We iteratively discuss each of the categories emerged from our analysis (also showing small examples where needed). Afterwards, we discuss the following main questions with each interviewee:

(1) What is your experience with <category>?
(2) Do you think <category> is important to successfully perform a code review? Why?
(3) Do you think current code review tools support this need?
(4) How would you improve current tools?

Our goal with these questions is to allow us to better understand the relevance of each developer's need and whether developers feel it is somehow incorporated in current code review tools or, if not, how they would envision this need incorporated. Successively, we ask developers to rank the categories according to their perceived importance. Our goal is to understand what the interviewees perceive as the most important needs and why. To conclude the interview, the first two authors of this paper summarize the interview, and before finalizing the meeting, these summaries are presented to the interviewee to validate our interpretation of their opinions.

Focus group with an external commercial company. While the original developers provide an overview of the information needs identified in the context of the systems analyzed in this study, our findings may not provide enough diversity. To improve this aspect, we complement the aforementioned semi-structured interviews with an additional analysis targeting experts in assessing the source code quality of systems. In particular, we recruited three employees from a firm in Europe specialized in software quality assessments for their customers. The mission of the firm is the definition of techniques and tools able to diagnose design problems in the clients' source code, with the purpose of providing consultancy on how to improve the productivity of their clients' industrial developers. Our decision to involve these quality experts is driven by the willingness to receive authoritative opinions from professionals who are used to perform code reviews for their customers. The three participants have more than 15 years in assess code quality and more than 10 years of experience in code review.

[6]This minimum number of reviews to ensure an appropriate experience of the interviewees is aligned with the numbers used in previous studies on code review (*e.g.*, [2]).

In this case we proceed with a *focus group* [36, 40] because it better fits our methodology. Indeed, this technique is particularly useful when a small number of people is available for discussing about a certain problem [36, 40] and consists of the organization of a meeting that involves the participants and a moderator. The moderator starts the discussion by asking general questions on the topic of interest and then leaves the participants to openly discuss about it with the aim of gathering additional qualitative data useful for the analysis of the results. In the context of this paper, the first two authors of the paper are the moderators in a meeting directly organized in the consultancy firm. The focus group is one hour long and the participants reflected on and discuss the information needs we identified and what are the factors influencing their importance. From this analysis, our aim is also to better understand the external validity of our taxonomy.

3.4 RQ_2 - On the role of reviewers' needs in the lifecycle of a code review

In the context of the second research question we perform a fine-grained investigation of the role of reviewers' needs in code review. We analyze which of them capture more replies, what is the time required for getting an answer, and whether reviewers' needs change throughout the iterations.

Specifically, we consider code review threads related to the same reviewer's need independently. Then, to answer $\mathbf{RQ}_{2.1}$ we computed the number of replies that each group received: this is a metric that represents how much in deep reviewers and authors should interact to be able to exchange the information necessary to address the code review. We do not assess the quality of the responses, since we aim at reporting quantitative observations on the number of answers provided by authors to a reviewer's need.

As for $\mathbf{RQ}_{2.2}$, this represents a follow-up of the previously considered aspect. Indeed, besides assessing the number of replies for each reviewers' need, we also measure the time (in terms of minutes) needed to get a response. This complementary analysis can possibly provide insights on whether certain needs require authors to spend more time to make their change understandable, thus providing information on the relative importance of each need which might be further exploited to prioritize software engineering research effort when devising and developing new techniques to assist code reviewers.

Finally, to answer $\mathbf{RQ}_{2.3}$ and understand how the reviewers' needs change over the code review iterations, we measure the number of times a certain need appears in each iteration of a code review. This analysis may possibly lead to observations needed by the research community to promptly provide developers with appropriate feedback during the different phases of the code review process.

As a final step of our methodology, we compute pairwise statistical tests aimed at verifying whether the observations of each sub-research question are statistically significant. We apply the Mann-Whitney test [18]. This is a non-parametric test used to evaluate the null hypothesis stating that it is equally likely that a randomly selected value from one sample will be less than or greater than a randomly selected value from a second sample. The results are intended as statistically significant at α=0.05. We also estimate the magnitude of the measured differences by using the Cliff's Delta (or d), a non-parametric effect size measure for ordinal data [27]. We follow well-established guidelines to interpret the effect size values: negligible for $|d|$<0.10, small for 0.10≤ $|d|$ < 0.33, medium for 0.33 ≤ $|d|$ < 0.474, and large for $|d|$ ≥ 0.474 [27].

4 RESULTS

In this section, we present and analyze the results of our study by research question.

4.1 RQ_1 - A Catalog of Reviewers' Information Needs

We report the results of our first research questions, which aimed at cataloging reviewers' information needs in code review and assessing their diffusion. For the sake of comprehensibility, we answer each sub-research question independently.

$RQ_{1.1}$: *What are the kinds of information code reviewers require?*

Following the methodology previously described (Section 3.3), we obtained 22 groups of reviewers' information needs. They were then clustered according to their intention into *seven* high-level categories that represent the classes of information needs associated with the discussion threads considered in our study. We describe each high-level category also including representative examples.

N1. Suitability of An Alternative Solution
This category emerged by grouping threads in which the reviewer poses a question to discuss options and alternative solutions to the implementation proposed by the author in the first place. The purpose is not only to evaluate alternatives but also to trigger a discussion on potential improvements. The example reported in the following reports a case where the reviewer starts reasoning on how much an alternative solution is suitable for the proposed code change.

> R: "... Since the change owner is always admin, this code might be able to move out of the loop? The following should be enough for this? [lines of code]"

N2. Correct understanding
In this category, we group questions in which the reviewers try to ensure to have captured the real meaning of the changes under review; in other words, this category refers to questions asked to get a consensus of reviewers' interpretation and to clarify doubts. This is more frequent when code comments or related documentation is missing. as reported in the example shown in the following.

> R: "This is now an empty heading ... Or do you feel it is important to point out that these are C++ classes?"
> A: "The entire page is split up into [more artifacts]. The following sections only refer to [one artifact]. I added a sentence introducing the section."

N3. Rationale
This category refers to questions asked to get missing information that may be relevant to justify why the project needs the submitted change set or why a specific change part was implemented/designed in a certain way. For example, a reviewer may request more details about the issue that the patch is trying to address. These details help the reviewer in better understanding whether the change fits with the project scope and style. For instance, in the example reported below the reviewer (R) asks why the author replaced a piece of code.

> R: "Can you explain why you replaced [that] with [this] and where exactly was failing?"

N4. Code Context
In this category, we grouped questions asked to retrieve information aimed at clarifying the context of a given implementation. During a code review, a reviewer has access to the

entire codebase and, in this way, may reconstruct the invocation path of a given function to understand the impact of the proposed change. However, we observed that the reviewer needs contextual information to clarify a particular choice made by authors. These questions range from very specific (*i.e.*, aimed at understanding the code behavior) to more generic (*i.e.*, aimed at clarifying the context in which such code is executed). The author replies to such questions by providing additional explanations on the code change or contextual project details. For instance, let consider the thread reported below, where the Author (A) replies to the Reviewer (R) by pointing R to the file (and the line) containing the asked clarification.

> R: "In what situations would [this condition] be false, but not undefined?"
> A: "See [file], exactly in line [number], in this case the evaluation of the expression returns false."
> R: "It may be helpful to add a comment documenting these situations to avoid future regressions."

N5. Necessity
In this category, the reviewer needs to know whether a (part of) the change is really necessary or can be simplified/removed. For example, a reviewer may spot something that seems like a duplicated code, yet is unsure if whether the existing version is a viable solution or it should be implemented as proposed by the author. In the example below, the reviewer asks whether a certain piece of code could be removed.

> R: "Is this needed?"
> A: "I believe its only required if you have methods after the last enum value, but I generally add it regardless. We have a pretty arbitrary mix."

N6. Specialized Expertise
Threads belonging to this category regard situations in which a reviewer finds or feels there is a code issue, however, the reviewer's knowledge is not appropriate to propose a solution. In these cases, typically a reviewer asks other reviewers to step in and contribute with their specialized expertise. Sometimes, reviewers may ask the author to propose informal alternatives that may better address the found issue. The examples reported below show two cases where the reviewer encourages other developers to reason on how to fix an issue.

> R: "… Lars, Simon, any ideas? We really need to fix this for [the next release] and the time draws nigh"

> R: "I need a better way to handle this …not a good idea to hard code digits in there. example also needs to be removed, its there just to make the tests pass."

N7. Splittable
For several reasons (including reducing the cognitive load of reviewers [5]), authors want to propose changes that are *atomic* and *self-contained* (*e.g.*, address a single issue or add a single feature). However, sometimes, what authors propose may be perceived by reviewers as something that can be addressed by different code changes, thus reviewed separately. For this reason, a reviewer needs to understand whether the split she has in mind can be done; based on this the reviewer asks questions aimed at finding practical evidence behind this

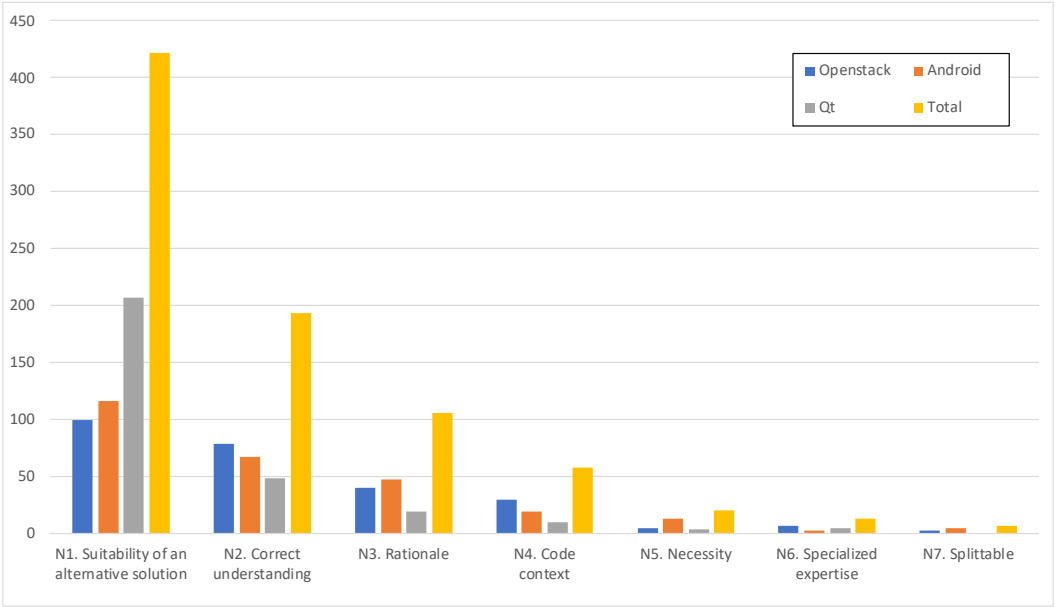

Fig. 4. Distribution of reviewers' information needs across the considered systems.

idea. In other words, this category gathers questions proposed by reviewers who need to understand whether the proposed changes can be split into multiple, separated patches. For example, the thread below reports a question where the reviewer (R) asks the author about the possibility of splitting unrelated changes, but the feasibility of this split is not confirmed.

> R: "This looks like an unrelated change. Should it be in a separate commit?"
> A: "Actually its related. The input object is needed to log the delete options."
> R: "OK, I wasn't sure because in the previous version we don't pass 'force' into the method, but now we do pass it in via the 'input'."

In addition to the aforementioned categories, we found several cases in which the presence of a question or question mark did not correspond to a real information need, similarly to the aforementioned categorized information needs, we provide an example in the following.

> R: "I hate `name` as a name. What kind of name is this?"

> R: 'If you thought it was necessary to check `exe()` for errors, then why'd you leave out [another part] here? :)"

RQ$_{1.2}$: *How often does each category of reviewers' needs occur?*

Figure 4 depicts bar plots that show the distribution of each reviewers' need over the considered set of code review threads. The results clearly reveal that not all the information needs are equally distributed and highlight the presence of a particular type (*i.e.*, N1. 'Suitability of an alternative solution'), which has a way larger number of occurrences with respect to all the others (the result is consistent over the three considered systems). Thus, we can argue that one of the most useful

tools for reviewers would definitively be one that allows them to have just-in-time feedback on the actual practicability of possible alternative solutions with respect to the one originally implemented by the authors of the committed code change.

The second most popular category is represented by 'Correct understanding' (N2), *i.e.*, questions aimed at assessing the reviewers' interpretation of the code change and to clarify doubts. This finding basically confirms one the main outputs of the work by Bacchelli and Bird [2], who found that *code review is understandability*. The popularity of this category is similar in all the considered projects, confirming that this need is independent from the type of system or the developers working on it.

Still, a pretty popular need is 'Rationale' (N3). This has also to do with the understandability of the code change, however in this case it seems that a common reviewers' need is having detailed information on the motivations leading the author to perform certain implementation choices.

Other categories are less diffused, possibly indicating that reviewers do not always need such types of information. For instance, 'Splittable' (N7) is the category having the lowest number of occurrences. This might be either because of the preventive operations that the development community adopts to limit the number of *tangled changes* [30] or because of the attention that developers put when performing code changes. In any case, this category seems to be less diffused and, as a consequence, one can claim that future research should spend more effort on different (most popular) reviewers' information needs.

RQ$_{1.3}$: *How do developers' perceive the identified needs?*

In this section, we present the results of our interviews and focus group with developers. First, we report on the participants' opinions on the taxonomy derived from the previous two sub-research questions, then we describe the most relevant themes that emerged from the analysis of the transcripts. We refer to individual interviewees using their identifiers ($P_{\#}$ and $FG_{\#}$).

Participants' opinion on the taxonomy. In general, all interviewees agreed on the information needs emerged from the code review threads: For all the categories, the developers agreed that they were asking those types of question themselves, several times and repeatedly. Furthermore, the order of importance of the categories was also generally agreed upon: According to the interviewees, the most important and discussed topic is 'suitability of an alternative solution' (N1), followed by 'understanding' (N2), 'rationale' (N3), and 'code context' (N4). Interestingly, the 'splittable' (N7) category is perceived as very important for the interviewed developers, but they confirmed that it happens rarely to receive big and long patches to review.

Although also the participants in the focus group agreed with the taxonomy of needs and their ranking, they stated that questions regarding 'correct understanding' (N2) are not common (in our taxonomy is ranked second). When discussing this difference with the focus group participants, they argued that this discrepancy was probably due to the type of projects we analyzed: Indeed, we analyzed open-source systems, while the focus group was conducted with participants working in an industrial, closed-source setting. One developer said: *"if I don't understand something of the change, I just go to my colleague that created it and ask to him. This is possible because we are all in the same office in the same working hours, while this is not the case in the projects you analyzed."*

Understanding a code change to review. An important step for all the interviewees when it comes to reviewing a patch is *to understand the rationale behind the changes* (N3). P_1 explained that to understand why the author wrote the patch, he first reads the commit message, since *"[it] should be enough to understand what's going on."* Interviewees said that it is very useful to have attached a ticket to the commit message, for example a JIRA issue, to really understand why it was necessary submitting the patch [P_{1-4}, FG_{1-3}]. However, sometime the patch is difficult to understand, and this leads to reviewers asking for more context or rationale of the change, as P_1 put it: *"Sometime*

the commit message just says "Yes, fix these things." And you say "Why? Was it broken? Is there a bug report information?" So in this case there is not enough description, and I would have to ask for it". Interestingly, P_1 reported that this issues generally happens with new contributors or with novice developers. During the focus group, FG_3 said he also uses tests to obtain more context about the change: *"In general, to get more context I read the Java docs or the tests."* Finally, all the interviewees explained that to obtain more context, or the rationale behind the change, they use external IRC channels (outside Gerrit) to get in touch with reviewers/authors, *e.g.*, by emails, or Slack.

Authors' information needs. Considering the point of view of the *author* of a change, the interviewees explained that code review is sometimes used as a way to *get information* from specialized experts, thus underling the dual nature of the knowledge exchange happening in code review [25, 26]. P_2 explained that it is sometimes difficult for an author to have all the information they need to make the change, for example if the change is in a part of the system where they are not expert. In this case, P_2 explained: *"when you make a change, you usually add the experts of the system to your review, and then you ping them on IRC, asking for a review, if they have some time."* Interestingly, this point also came up during the focus group, where a developer said *"if it's a new system [...] my knowledge lacks at one front, it may be technology, it may be knowledge of the system."* In this case, the developer would ask the help of colleagues. This is also in line with another need we discovered in the previous research question, that is the 'Specialized Expertise' (N6). Indeed, interviewees said that when they are not familiar with the change, or they do not have the full context of the change, they ask an expert to contribute: *"[in the project where I work] we have sub-system maintainers: they are persons with knowledge in that area and have more pleasure or willingness to work on those specialized areas. If the reviewers do not reach consensus during the review, we always ask to those experts."* [P_4].

Small and concise patches. When discussing with developers the 'Splittable' (N7) need, all agreed that patches should be self-contained as much as possible [P_{1-4}, FG_{1-3}]. P_3 said: *"I always ask to split it, because in the end it will be faster to get it in [the system]."* P_4 added that it is something that they do all the time, because usually people do not see this issue. P_2 said: *"It's always better to have 10 small reviews than one big review with all the changes, because no one will review your code. It's like that. So if you want to merge something big, it's always better to do it in small changes."*

Another point raised during the interviews is that large patch sets are difficult to review and require a lot of time to read [P_{1-4}, FG_{1-3}], thus this may delay the acceptance of the patches. P_4 explained: *"You can have a large patch set that is 90% okay and 10% that not okay: the 10% will generate a lot of discussion and will block the merge of 90% of the code. So yes, it's something that I do all the time. I ask people, you need to organize better the patch."*

When talking about the issue, P_1 also added that having small patches is very important for making it easier to revert them: *"yes this is something I find it to be really, really important. Bugs are everywhere – there is always another bug to fix. So the patch should be small enough that [in case of bugs] you can revert it without breaking any particular code."*

Interestingly, all the interviewees agreed that tools could help reviewers and authors in solving this need: for example, when submitting a large patch, the tool could suggest the author to split it into more parts to ease the reviewing process.

Offering a solution. All interviewees agreed that to do a proper code review, reviewers should always pinpoint the weak parts in the code and offer a solution [P_{1-4}, FG_{1-3}]. P_3 said: *"When I request the change, I usually put a link or example because I know that maybe the other guy doesn't know about the other approach. This is usually the main reason why somebody didn't do something: because he didn't know it was possible."* P_1 added: *"[...] whenever you propose a change, you should always explain why you need to change it and what. Just putting [the score to] minus two, or even*

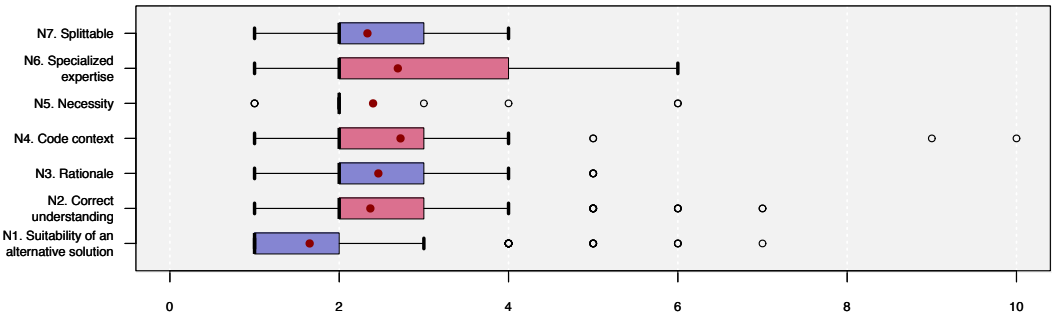

Fig. 5. Distribution of the number of replies for each reviewers' need.

minus one without explanation, is bad because then people don't know what to do. We have to try being more friendly as a community." This constructive behavior was also agreed upon during the focus group: one developer said that the worst thing that can happen in a code review is a non-constructive comment. Interestingly, this reported behavior confirms what we discovered in our previous research question: indeed, 'Suitability of an alternative solution' (N1) is the most frequent type of question when doing code review.

In addition, concerning constructive feedback, interviewees said that when they do not fully understand a change, they first ask the author explanations: *"For example, if you don't understand correctly the change this person is trying to add, you just ask him, and they are forced to answer you. And if you don't have the context information, they should be able to provide it to you."*[P_2] Interviewees said that it is better to ask explanation to the author first, and only after decide to/not merge the patch. P_4 also explained that sometime it is better to accept a patch than start a big discussion on small detail: *"Even though I understand that a better solution will be doable, I'll probably won't propose it because a lot of times people won't have time to actually rework on a new proposal, and you need to balance how you want the project to move forward: Sometimes it's better to have a code that is not the best solution, but at least does not regress and it fixes a bug."*

4.2 RQ_2 - The Role of Reviewers' Information Needs In A Code Review's Lifecycle

We present the results achieved when answering our second research question, which was focused on the understanding of the role of reviewers' information needs in the lifecycle of the code review process. We report the results by considering each sub-research question independently.

$RQ_{2.1}$: *What are the reviewers' information needs that attract more discussion?*

To answer $RQ_{2.1}$ and understand to what extent the reviewers' needs attract developers' discussions, we compute, for each discussion thread that we manually categorized, the number of iterations that involve the developers of a certain code review.

Figure 5 depicts box plots reporting the distribution of the number of answers for each reviewers' need previously identified (red dots indicate the mean). Approximately 18% of code review threads (considering both merged and abandoned patches of every projects) do not have an answer. The first observation regards the median value of each distribution: as shown, all of them are within one and three, meaning that most of the threads are concluded with a small amount of discussion. From a practical perspective, this result highlights that authors can address almost immediately the need pointed out by a reviewer; at the same time, it might highlight that tools able to address the reviewers' needs identified can be particularly useful to even avoid the discussion and lead to an important gain in terms of time spent to review source code. Among the reviewers' needs, the

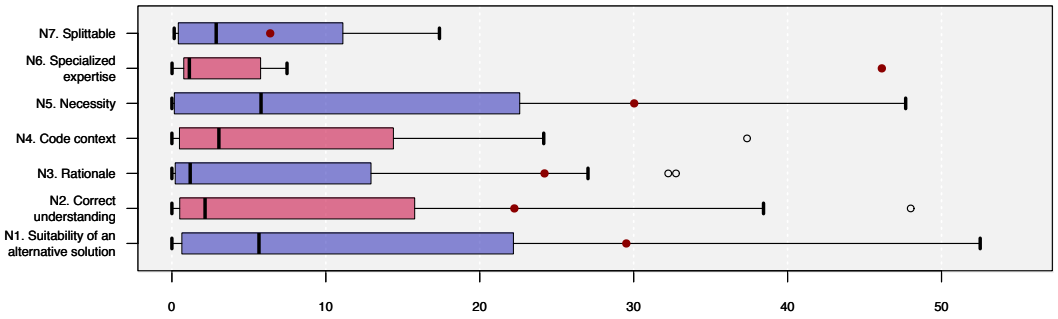

Fig. 6. Distribution of the number of hours needed to answer each reviewers' need category.

'Specialized expertise' (N6) is the one with the largest scattering of discussion rate. This result seems to indicate that the more collaboration is required due the largest number of replies a discussion receives, which possibly preclude the integration in the codebase of important changes that require the expertise of several people.

The statistical tests confirmed that there are no statistically significant differences among the investigated distributions, with the only exception of 'Suitability of an alternative solution' (N1), for which the ρ-value is lower than 0.01 and the Cliff's d is 'medium'. This category is the one having the lowest mean (1.7) and we observed that often authors of the code change tend to directly implement the alternative solution proposed by the reviewer without even answering to the original comment. This tendency possibly explain the motivation behind this statistical difference.

Overall, according to our results, most of the reviewers' information needs are satisfied with few replies–most discussions are closed shortly. The only category having more scattered results is the one where reviewers ask for the involvement of more people in the code review process.

RQ$_{2.2}$: *How long does it take to get a response to each reviewers' information need?*

Figure 6 reports the distribution of the number of hours needed to get an answer for each group of reviewers' information need. In this analysis we could only consider the questions having at least one answer; similarly, if a reviewer's comment got more than one reply, we considered only the first one to compute the number of hours needed to answer the comment.

Looking at the results, we can observe that the median is under 7 hours for almost all the categories. A possible reason for that consists of the nature of the development communities behind the subject systems. Indeed, all the projects have development teams that span across different countries and timezones: thus, we might consider as expected the fact to not have an immediate reaction to most of the comments made by reviewers. Some differences can be observed in the distributions of two reviewers' information needs such as 'Necessity' (N5) and 'Suitability of an alternative solution' (N1). In this case, the median number of hours is higher with respect to the other categories (7 vs 5), while the 3rd quartiles are around one day (meaning that 25% of the questions in this category took more than one day to have a response). Conversely, the discussion of other categories generally took less time to start. For instance, the 'Specialized expertise' (N6) need has a median of one hour and a 3rd quantile equal to four. Such differences, however, are not statistically significant.

To conclude the analysis of our findings for this research question, we can argue that developers generally tend to respond slower to questions regarding the proposal of alternatives and the evaluation of the actual necessity of a certain code change; on the other hand, questions where more reviewers are called to discuss seem to get a faster response time.

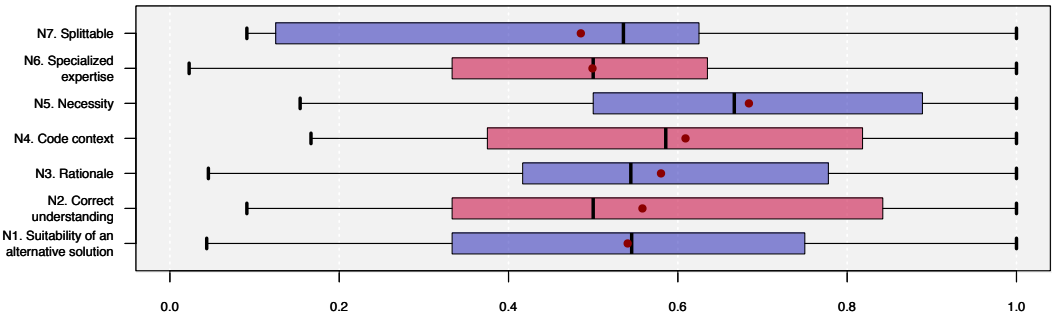

Fig. 7. Distribution of reviewers' needs over different iterations of the code review process.

RQ$_{2.3}$: *How do the reviewers' information needs change over the code review process?*

The last research question targets the understanding of whether reviewers' needs vary over the different iterations of the code review process. Figure 7 presents the result of our analysis, with box plots depicting the distribution of each reviewers' information need in the various iterations: for the sake of better comprehensibility of the results, we considered the normalized number of iterations available in each of the 900 code review threads analyzed.[7]

Almost all the categories have their median around 0.5, meaning that the majority of reviewers' information needs are raised in the first half of the review process. Moreover, we are not able to map reviewers' information needs with any specific iteration. This result might indicate that there is not a time-sensitive relationship between those needs and that they arise independently from how much discussion has already been going on in the review.

Besides this general conclusion, we also notice some differences between the category 'Necessity' (N5) and the others. In the case of the 'Necessity' (N5) category the median and mean reach both 0.67, thus indicating that most of such questions come later in the process. It is interesting to note that modifications aimed at performing perfective changes that improve the overall design/style of the source code rather than solving issues are mainly requested by reviewers in a later stage of the code review process, *i.e.*, likely after that most important fixes solving problems impacting the functioning of the system are already submitted by the author and answers about understanding the context of the change are given. Such an observation may need further investigations and validation, however it may possibly reveal the possibility to devise strategies to guide the next generation of code review tools toward a *selection* of the information that a reviewer might need in a earlier/later stage of the code review process.

5 THREATS TO VALIDITY

Our study might have been subject to a number of threats to validity that may affect our results. This section summarizes the limitations of our study and how we tried to mitigate them.

Validity of the defined reviewers' needs. Since the meaning of a question may be dependent by the context, we may lack of a full understanding of its nature and background. This type of threat may first apply to our study when we identify code review threads composed of both questions and answers: to this aim, we automatically mined the Gerrit repository that is a reliable source for the extraction of code review data [10, 28]. To extract code review threads we employed the publicly available APIs of such repository: For this reason, we are confident on the completeness of the extracted data.

[7]We also conducted an analysis using the absolute number of iterations, yet results were equivalent.

The adopted open card sorting process is also inherently subjective because different themes are likely to emerge from independent card sorts conducted by the same or different people. To ensure the correctness and completeness of the categories associated to the reviewers' needs identified with the card sorting, we iteratively conducted the process by merging and splitting reviewers' need categories if needed. As an additional step, we also took into account authors responses and discussion threads when classifying questions made by reviewers, with the aim of properly understand the context in which a certain question has been made. Moreover, all the authors of this paper, who have more than seven years of experience in software development, assessed the validity of the emerged categories, thus increasing its overall completeness. Of course, we cannot exclude the missing analysis of specific code review threads that point to categories that were not identified in our study.

We consider questions asked by reviewers through the Gerrit platform as indicators of the actual reviewers' needs. This assumption may not hold for all projects, as many active projects do not use the Gerrit platform. For example, Tsay *et al.* [64] highlighted how several developers contribute to the software development by using different platforms (*e.g.*, GitHub). However, we partly mitigated this threat to validity by carefully selecting software systems broadly studied in code review research [8, 38, 39] and having a large number of code review data (which indicates they actively use Gerrit). The study of different platforms such as GitHub, GitLab, or Collaborator is left for future work.

External validity. As for the generalizability of the results, we conducted this study on a statistically significant sample of 900 code reviews that include more than 1, 800 messages belonging to three well-known projects that use the Gerrit platform since 2011. A threat to validity in this category may arise when we consider closed-source projects. In that case, the experience of closed-source reviewers may affect the need to asks clarification questions, therefore, the findings that we found in open-source project may be not generalizable to a closed-source context. As part of our future research agenda, we plan to extend this study by including closed-source projects.

6 DISCUSSION AND IMPLICATIONS

Our quantitative and qualitative results showed that reviewers have a diversity of information needs at different conceptual levels and pertaining to different aspects of the code under review. In this section we discuss how our results lead to recommendations for practitioners and designers, as well as implications for future research.

(1) **Selection of assistant experts.** The results achieved by mining code review repositories and interviewing practitioners indicate that 'Suitability of An Alternative Solution' (N1) and 'Correct Understanding' (N2) are not only the most recurring needs, but also those perceived as the most important. When discussing these topics with developers from both open-source and industrial systems, we uncovered possible areas where current code review tools can offer better features. For example, a key need for the reviewers is being able to communicate with the experts of the sub-system under review; this underlines the importance of tools able to recognize developers' expertise and create recommendations.
Researchers have conducted the first steps into this direction. For instance, among others, Patanamon *et al.* [63] proposed RevFinder, an approach to search and recommend reviewers based on similarity of previous reviewed files, while Thongtanunam *et al.* [59] validated the performance of a reviewer recommendation model based on file paths similarity. An interesting novelty that emerged from our analysis, with respect to existing previous work on reviewers recommendation, is the *target* of the recommendation. In fact, existing reviewer recommendation mechanisms target the author of the change who has to select the reviewer

and propose reviewers for full changes or files. Instead, we found that also the reviewers have the need to consult an external expert, maybe for a more specific part of the entire change under review. For instance P_3 explained that *"reviewers sometimes ask for other reviewers that may be more expert"*, thus having an assistant that can help the selection of an expert reviewer may increase her productivity. Targeting reviewers instead of change authors and having a finer grained focus for the recommendation mechanism can lead to interesting changes in both the model (which may use different features to compute expertise and the *difference* of expertise among reviewers) and the evaluation approach (which may no longer be based on just matching actually selected reviewers). Further studies can be designed and conducted to better understand this novel angle.

(2) **Early detection of splittable changes.** Even if the 'Splittable' (N7) category is the less frequently occurring, interviewees argued that it is really useful to automatically detect splittable code changes before a submission. For example, in the focus group all the participants (FG_{1-3}) suggested: *"if it's an unrelated change [...], pull it out of this ticket and put it on another issue."* In fact, this would (1) decrease the time spent in detecting this issue and asking the author to re-work the change, as well as (2) reduce the risks of introducing defects in the source code [30].
Researchers have already underlined the risks of tangled code changes (*i.e.*, non-cohesive code changes that may be divided in atomic commits) for mining software repositories approaches [30] and have proposed mechanisms for automatically splitting them. For instance, Herzig and Zeller [30] proposed an automated approach relying on static and dynamic analysis to identify which code changes should be separated; Yamauchi suggested a clustering algorithm tuned to identify unrelated changes in a commit message [69]; and Dias *et al.* [22] proposed a methodology to untangle code changes at a finer-granularity, *i.e.*, by selecting the single statement of a code review that should be placed in other commits. More recently researchers also proposed untangling techniques tailored explicitly to code review [5, 58] and conducted the first experiments to measure the effects of tangled code changes on code review [21, 58] substantiating the value of separating unrelated changes.
Despite these advances in splitting algorithms and their immediate practical value, no commercial code review tool offers this feature. Our analysis underlines even more the relevance of having such a feature integrated as early as possible in the development process, possibly in the development environment, so that authors send already self-contained patches for review. Moreover, despite the notable research advances in the field, we believe that there is still room for improvement, *e.g.*, by complementing state-of-the-art methods with conceptual-related information aimed at capturing the semantic relationships between different code changes.
Also, early improvements of code changes before review are in line with the work by Balachandran [4]. He reported that the time to market can be reduced also creating automatic bots able to conduct preliminary reviews [4]. In this regard, there are still plenty of opportunities and challenges on how and when bots can automatically help reviewers during their activities and whether they may be employed to assist some of the developers' needs in code review.

(3) **Automatically detecting alternative solutions.** In connection with the most frequent need (*i.e.*, 'Suitability of an alternative solution' (N1)), an interviewee from one of the open-source projects explained to prefer to propose an alternative solution before rejecting a patch: *"[...] usually I put a link or an example."* In this light, a promising avenue for an impactful improvement in code review is to integrate a toll that automatically mines alternative solutions. Accordingly, a first interesting step would be to investigate how to

integrate at code review time an approach such as the one proposed by Ponzanelli *et al.*, which systematically mines community based resources such as forums or Stack Overflow to propose related changes [46]. Another promising starting point in this direction is the concept of programming with "Big Code," as proposed by Vechev and Yahav [67], to automatically learn alternative solutions from the large amount of code available in public code repositories such as GitHub.

(4) **Synchronous communication support.** The absence of a proper real-time communication channel within code review tools was a common issue that emerged from both the interviews with the open-source developers and the focus group. In fact, two interviewees ([FG_1] and [FG_3]) explained: *"you can just go to the author and ask to him in person, and maybe it would be a long discussion [...]"*. This is in line with the experience reported by developers at Microsoft in a previous study by Bacchelli and Bird [2]. Nevertheless, in-person discussions can happen only if both author and reviewer are co-located, otherwise logistic barriers could impose serious constraints [2]. Yet, open-source developers are able to fulfill this real-time communication need using alternative channels; P_2 stated: *"we usually have an IRC channel [...]"*. The two observations suggest that, when it is possible, developers prefer to rely on direct communication to discuss feedback; this may be to avoid discussing difficult criticism online in a public forum and to have a higher communication bandwidth than small online thread comments. In both scenarios, our results show that current code review tools are clearly not able to fully satisfy the communication need of the involved people. Future work should be conducted to understand how communication can be facilitated within the code review tool itself (thus improving traceability of discussions, which is relevant for future developers' information needs [55]); in principle, this future analysis should take into account not only technical aspects to increase the communication bandwidth, but also the social aspects that could currently hinder developers from discussing certain arguments with the current tools.

(5) **Automatic change summarization.** 'Correct understanding' (N2) and 'Rationale' (N3) are also key information needs for reviewers. Normally this is achieved by perusing the code change description or additional comments. Nevertheless, our interviewees reported cases in which these sources of information were insufficient to fulfill this need; on this P_3 reported: *"I even had cases where the description didn't have anything in common with the code"*. Indeed this shows that another significant source of delay in a code review process is when patches contain unaligned or missing information (*i.e.*, the commit message is not clear enough or it does not match with the actual patch). Code summarization techniques appear to be a good fit for this task: Indeed, past literature presented different summarization techniques that can be used to both produce or check the current documentation. For example, Buse and Weimer proposed a technique to synthesize human-readable documentation starting from code changes [13], but also several other researchers have been contributing with more approachess: Canfora*et al.* experimented LDiff [15], Parnin and Görg developed CILDiff [45], and Cortés-Coy *et al.* designed ChangeScribe [19]. Our analysis suggests that supporting code review is a ripe opportunity for research on code summarization techniques to have another angle of impact on a real-world application.

7 CONCLUSIONS

Modern code review is an important technique used to improve software quality and promote collaboration and knowledge sharing within a development community. In a typical code review process, authors and reviewers interact with each other to exchange ideas, find bugs, and discuss

alternative solutions to better design the structure of a submitted code change. Often reviewers are required to inspect author patches without knowing the rationale or without being aware of the context in which a code change is supposed to be plugged-in. Therefore, they must ask questions aimed at addressing their doubts, possibly waiting for a long time before getting the expected clarifications. This might potentially result in causing delays in the integration of important changes into production.

In this work we investigate the reviewers' information needs by analyzing 900 code review threads of three popular open-source software systems (OpenStack, Android, and QT). Moreover, we conduct four semi-structured interviews with developers from the considered projects and one focus group with developers from a software quality consultancy company, with the aim of challenging and discussing our outcome.

We discovered the existence of seven high-level reviewers' information needs, which are differently distributed and have, therefore, different relevance for reviewers. Furthermore, we analyzed the role played by each category of reviewers' information needs across the lifecycle of a code review, and in particular what are the reviewers' information needs that attract more discussion, for how long a reviewer should wait to get a response, and how the information needs change over the code review lifecycle.

Based on our findings, we provide recommendations for practitioners and researchers, as well as viable directions for impactful tools and future research. We hope that the insights we have discovered will lead to improved tools and validated practices which in turn may lead to higher code quality overall.

ACKNOWLEDGMENTS

This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Sklodowska-Curie grant agreement No 642954. Bacchelli and Palomba gratefully acknowledge the support of the Swiss National Science Foundation through the SNF Project No. PP00P2_170529.

REFERENCES

[1] Ulrike Abelein and Barbara Paech. 2015. Understanding the Influence of User Participation and Involvement on System Success: a Systematic Mapping Study. *Empirical Software Engineering* 20, 1 (2015), 28–81. https://doi.org/10.1007/s10664-013-9278-4

[2] Alberto Bacchelli and Christian Bird. 2013. Expectations, Outcomes, and Challenges of Modern Code Review. In *35th International Conference on Software Engineering (ICSE 2013a)*. 710–719.

[3] Richard A Baker Jr. 1997. Code reviews enhance software quality. In *Proceedings of the 19th International Conference on Software Engineering*. ACM, 570–571.

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

[5] Mike Barnett, Christian Bird, Joao Brunet, and Shuvendu K Lahiri. 2015. Helping developers help themselves: Automatic decomposition of code review changesets. In *Proceedings of the 2015 International Conference on Software Engineering*. *IEEE Press*.

[6] Tobias Baum, Olga Liskin, Kai Niklas, and Kurt Schneider. 2016. A Faceted Classification Scheme for Change-Based Industrial Code Review Processes. In *Software Quality, Reliability and Security (QRS), 2016 IEEE International Conference on*. IEEE, Vienna, Austria. https://doi.org/10.1109/QRS.2016.19

[7] Tobias Baum, Kurt Schneider, and Alberto Bacchelli. 2017. On the Optimal Order of Reading Source Code Changes for Review. In *33rd IEEE International Conference on Software Maintenance and Evolution (ICSME), Proceedings*.

[8] Gabriele Bavota and Barbara Russo. 2015. Four eyes are better than two: On the impact of code reviews on software quality. In *2015 IEEE 31st International Conference on Software Maintenance and Evolution, ICSME 2015 - Proceedings*. 81–90. https://doi.org/10.1109/ICSM.2015.7332454

[9] Olga Baysal, Oleksii Kononenko, Reid Holmes, and Michael W Godfrey. 2013. The influence of non-technical factors on code review. In *Reverse Engineering (WCRE), 2013 20th Working Conference on*. IEEE, 122–131.

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

[11] Fevzi Belli and Radu Crisan. 1996. Towards automation of checklist-based code-reviews. In *Software Reliability Engineering, 1996. Proceedings., Seventh International Symposium on*. IEEE, 24–33.

[12] Silvia Breu, Rahul Premraj, Jonathan Sillito, and Thomas Zimmermann. 2010. Information Needs in Bug Reports : Improving Cooperation Between Developers and Users. *Proceedings of the 2010 Computer Supported Cooperative Work Conference* (2010), 301–310. https://doi.org/10.1145/1718918.1718973

[13] Raymond PL Buse and Westley R Weimer. 2010. Automatically documenting program changes. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*. ACM, 33–42.

[14] Raymond P.L. Buse and Thomas Zimmermann. 2012. Information needs for software development analytics. In *Proceedings - International Conference on Software Engineering*. 987–996. https://doi.org/10.1109/ICSE.2012.6227122

[15] Gerardo Canfora, Luigi Cerulo, and Massimiliano Di Penta. 2009. Ldiff: An enhanced line differencing tool. In *Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, 595–598.

[16] Robert Chatley and Lawrence Jones. 2018. Diggit: Automated code review via software repository mining. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 567–571.

[17] Jason Cohen. 2010. Modern Code Review. In *Making Software*, Andy Oram and Greg Wilson (Eds.). O'Reilly, Chapter 18, 329–338.

[18] W. J. Conover. 1999. *Practical Nonparametric Statistics* (3rd ed.). John Wiley & Sons, Inc.

[19] Luis Fernando Cortés-Coy, Mario Linares-Vásquez, Jairo Aponte, and Denys Poshyvanyk. 2014. On automatically generating commit messages via summarization of source code changes. In *Source Code Analysis and Manipulation (SCAM), 2014 IEEE 14th International Working Conference on*. IEEE, 275–284.

[20] Jacek Czerwonka, Michaela Greiler, and Jack Tilford. 2015. Code Reviews Do Not Find Bugs. How the Current Code Review Best Practice Slows Us Down. In *Proceedings of the 2015 International Conference on Software Engineering*. IEEE – Institute of Electrical and Electronics Engineers. http://research.microsoft.com/apps/pubs/default.aspx?id=242201

[21] Marco di Biase, Magiel Bruntink, Arie van Deursen, and Alberto Bacchelli. 2018. The effects of change-decomposition on code review-A Controlled Experiment. *arXiv preprint arXiv:1805.10978* (2018).

[22] Martín Dias, Alberto Bacchelli, Georgios Gousios, Damien Cassou, and Stéphane Ducasse. 2015. Untangling fine-grained code changes. In *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*. IEEE, 341–350.

[23] Felipe Ebert, Fernando Castor, Nicole Novielli, and Alexander Serebrenik. 2017. Confusion Detection in Code Reviews. In *33rd International Conference on Software Maintenance and Evolution (ICSME), Proceedings*. ICSME.

[24] Deborah Finfgeld-Connett. 2014. Use of content analysis to conduct knowledge-building and theory-generating qualitative systematic reviews. *Qualitative Research* 14, 3 (2014), 341–352.

[25] Georgios Gousios, Margaret-Anne Storey, and Alberto Bacchelli. 2016. Work Practices and Challenges in Pull-Based Development: The Contributor's Perspective. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, 285–296.

[26] Georgios Gousios, Andy Zaidman, Margaret-Anne Storey, and Arie Van Deursen. 2015. Work practices and challenges in pull-based development: the integrator's perspective. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 358–368.

[27] Robert J Grissom and John J Kim. 2005. *Effect sizes for research: A broad practical approach*. Lawrence Erlbaum Associates Publishers.

[28] Kazuki Hamasaki, Raula Gaikovina Kula, Norihiro Yoshida, AE Cruz, Kenji Fujiwara, and Hajimu Iida. 2013. Who does what during a code review? datasets of oss peer review repositories. In *Proceedings of the 10th Working Conference on Mining Software Repositories*. IEEE Press, 49–52.

[29] JD Herbsleb and E Kuwana. 1993. Preserving knowledge in design projects: What designers need to know. *Chi '93 & Interact '93* (1993), 7–14. https://doi.org/10.1145/169059.169061

[30] Kim Herzig and Andreas Zeller. 2013. The impact of tangled code changes. In *Proceedings of the 10th Working Conference on Mining Software Repositories*. IEEE Press, 121–130.

[31] Yujuan Jiang, Bram Adams, and Daniel M. German. 2013. Will my patch make it? And how fast?: Case study on the linux kernel. In *IEEE International Working Conference on Mining Software Repositories*. 101–110. https://doi.org/10.1109/MSR.2013.6624016

[32] Norihito Kitagawa, Hideaki Hata, Akinori Ihara, Kiminao Kogiso, and Kenichi Matsumoto. 2016. Code review participation: game theoretical modeling of reviewers in gerrit datasets. In *Proceedings of the 9th International Workshop on Cooperative and Human Aspects of Software Engineering*. ACM, 64–67.

[33] Andrew J. Ko, Robert DeLine, and Gina Venolia. 2007. Information Needs in Collocated Software Development Teams. In *29th International Conference on Software Engineering (ICSE'07)*. 344–353. https://doi.org/10.1109/ICSE.2007.45

[34] Oleksii Kononenko, Olga Baysal, Latifa Guerrouj, Yaxin Cao, and Michael W. Godfrey. 2015. Investigating code review quality: Do people and participation matter?. In *2015 IEEE 31st International Conference on Software Maintenance and Evolution, ICSME 2015 - Proceedings*. 111–120. https://doi.org/10.1109/ICSM.2015.7332457

[35] Klaus Krippendorff. 2011. Agreement and information in the reliability of coding. *Communication Methods and Measures* 5, 2 (2011), 93–112.

[36] Mike Kuniavsky. 2003. *Observing the user experience: a practitioner's guide to user research*. Elsevier.

[37] Mika V Mantyla and Casper Lassenius. 2009. What types of defects are really discovered in code reviews? *Software Engineering, IEEE Transactions on* 35, 3 (2009), 430–448.

[38] Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E Hassan. 2014. The impact of code review coverage and code review participation on software quality: A case study of the qt, vtk, and itk projects. In *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM, 192–201.

[39] Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E. Hassan. 2016. An empirical study of the impact of modern code review practices on software quality. 21, 5 (2016), 2146–2189. https://doi.org/10.1007/s10664-015-9381-9

[40] Robert K Merton and Patricia L Kendall. 1946. The focused interview. *American journal of Sociology* 51, 6 (1946), 541–557.

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

[42] Hazel E Nelson. 1976. A modified card sorting test sensitive to frontal lobe defects. *Cortex* 12, 4 (1976), 313–324.

[43] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Denys Poshyvanyk, and Andrea De Lucia. 2015. Mining version histories for detecting code smells. *IEEE Transactions on Software Engineering* 41, 5 (2015), 462–489.

[44] Fabio Palomba, Annibale Panichella, Andy Zaidman, Rocco Oliveto, and Andrea De Lucia. 2017. The scent of a smell: An extensive comparison between textual and structural smells. *IEEE Transactions on Software Engineering* (2017).

[45] Chris Parnin and Carsten G'org. 2008. Improving change descriptions with change contexts. In *Proceedings of the 2008 international working conference on Mining software repositories*. ACM, 51–60.

[46] Luca Ponzanelli, Simone Scalabrino, Gabriele Bavota, Andrea Mocci, Massimiliano Di Penta, Rocco Oliveto, and Michele Lanza. 2017. Supporting Software Developers with a Holistic Recommender System. In *Proceedings of ICSE 2017 (39th ACM/IEEE International Conference on Software Engineering)*. to be published.

[47] Adam Porter, Harvey Siy, Audris Mockus, and Lawrence Votta. 1998. Understanding the sources of variation in software inspections. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 7, 1 (1998), 41–79.

[48] Achyudh Ram, Anand Ashok Sawant, Marco Castelluccio, and Alberto Bacchelli. 2018. What Makes A Code Change Easier To Review? An Empirical Investigation On Code Change Reviewability. In *26th Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018)*. forthcoming.

[49] Peter C Rigby and Christian Bird. 2013. Convergent contemporary software peer review practices. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, Saint Petersburg, Russia, 202–212.

[50] Peter C Rigby, Daniel M German, Laura Cowen, and Margaret-Anne Storey. 2014. Peer Review on Open Source Software Projects: Parameters, Statistical Models, and Theory. *ACM Transactions on Software Engineering and Methodology* (2014), 34.

[51] Caitlin Sadowski, Emma Söderberg, Luke Church, Michal Sipko, and Alberto Bacchelli. 2018. Modern code review: a case study at google. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*. ACM, 181–190.

[52] Chris Sauer, D Ross Jeffery, Lesley Land, and Philip Yetton. 2000. The effectiveness of software development technical reviews: A behaviorally motivated program of research. *Software Engineering, IEEE Transactions on* 26, 1 (2000), 1–14.

[53] Jonathan Sillito, Gail C Murphy, and Kris De Volder. 2006. Questions programmers ask during software evolution tasks. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*. ACM, 23–34.

[54] Davide Spadini, Maurício Aniche, Margaret-Anne Storey, Magiel Bruntink, and Alberto Bacchelli. 2018. When Testing Meets Code Review: Why and How Developers Review Tests. In *Software Engineering (ICSE), 2018 IEEE/ACM 40th International Conference on*. to appear.

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

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

[57] Yida Tao, Yingnong Dang, Tao Xie, Dongmei Zhang, and Sunghun Kim. 2012. How do software engineers understand code changes?: an exploratory study in industry. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM.

[58] Yida Tao and Sunghun Kim. 2015. Partitioning composite code changes to facilitate code review. In *Proceedings of the 12th Working Conference on Mining Software Repositories*. IEEE Press, 180–190.
[59] Patanamon Thongtanunam, Raula Gaikovina Kula, Ana Erika Camargo Cruz, Norihiro Yoshida, and Hajimu Iida. 2014. Improving code review effectiveness through reviewer recommendations. In *Proceedings of the 7th International Workshop on Cooperative and Human Aspects of Software Engineering*. ACM, 119–122.
[60] Patanamon Thongtanunam, Shane McIntosh, Ahmed E Hassan, and Hajimu Iida. 2015. Investigating Code Review Practices in Defective Files: An Empirical Study of the Qt System. In *MSR '15 Proceedings of the 12th Working Conference on Mining Software Repositories*.
[61] Patanamon Thongtanunam, Shane Mcintosh, Ahmed E. Hassan, and Hajimu Iida. 2016. Review Participation in Modern Code Review. *Empirical Software Engineering (EMSE)* (2016), to appear. https://doi.org/10.1007/s10664-016-9452-6
[62] Patanamon Thongtanunam, Shane McIntosh, Ahmed E Hassan, and Hajimu Iida. 2016. Revisiting code ownership and its relationship with software quality in the scope of modern code review. In *Proceedings of the 38th international conference on software engineering*. ACM, 1039–1050.
[63] Patanamon Thongtanunam, Chakkrit Tantithamthavorn, Raula Gaikovina Kula, Norihiro Yoshida, Hajimu Iida, and Ken-ichi Matsumoto. 2015. Who should review my code? A file location-based code-reviewer recommendation approach for modern code review. In *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*. IEEE, 141–150.
[64] Jason Tsay, Laura Dabbish, and James Herbsleb. 2014. Let's talk about it: evaluating contributions through discussion in GitHub. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 144–154.
[65] Yuriy Tymchuk, Andrea Mocci, and Michele Lanza. 2015. Code Review: Veni, ViDI, Vici. In *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*. IEEE, 151–160.
[66] Bogdan Vasilescu, Daryl Posnett, Baishakhi Ray, Mark GJ van den Brand, Alexander Serebrenik, Premkumar Devanbu, and Vladimir Filkov. 2015. Gender and tenure diversity in GitHub teams. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*. ACM, 3789–3798.
[67] Martin Vechev, Eran Yahav, et al. 2016. Programming with "Big Code". *Foundations and Trends® in Programming Languages* 3, 4 (2016), 231–284.
[68] Robert S Weiss. 1995. *Learning from strangers: The art and method of qualitative interview studies*. Simon and Schuster.
[69] Kenji Yamauchi, Jiachen Yang, Keisuke Hotta, Yoshiki Higo, and Shinji Kusumoto. 2014. Clustering commits for understanding the intents of implementation. In *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*. IEEE, 406–410.

Received April 2018; revised July 2018; accepted September 2018