**Investigating the effectiveness of peer code review in distributed software development based on objective and subjective data**

**Abstract** Code review is a potential means of improving software quality. To be effective, it depends on different factors, and many have been investigated in the literature to identify the scenarios in which it adds quality to the final code. However, factors associated with distributed software development, which is becoming increasingly common, have been little explored. Geographic distance can impose additional challenges to the reviewing process. We thus in this paper present the results of a mixed-method study of the effectiveness of code review in distributed software development. We investigate factors that can potentially influence the outcomes of peer code review. The study involved an analysis of objective data collected from a software project involving 201 members and a survey with 50 practitioners with experience in code review. Our analysis of objective data led to the conclusion that a high number of changed lines of code tends to increase the review duration with a reduced number of messages, while the number of involved teams, locations, and participant reviewers generally improve reviewer contributions, but with a severe penalty to the duration. These results are consistent with those obtained in the survey regarding the influence of factors over duration and participation. However, participants’ opinion about the impact on contributions diverges from results obtained from historical data, mainly with respect to distribution.

**Keywords:** Code review, Distributed software development, Empirical study, Survey

**1 Background**

Code review is a common practice adopted in software development to improve software quality based on static code analysis by peers. There are studies that provide evidence that it reduces the number of defects detected after release, mainly when it has adequate code coverage as well as engagement and participation of reviewers (McIntosh et al. 2014). Moreover, code review is a recognized way to foster knowledge sharing that benefits authors and reviewers (Hundhausen et al. 2013). It also improves team collaboration because it creates collective ownership of the source code, which results from collaborative work rather than individual work (Bacchelli and Bird 2013; Thongtanunam et al. 2016b). Nowadays, code reviews are less formal than in earlier decades of software development. In the past, it was typically in the form of code inspections (Fagan 1986), which required formal meetings and checklists (Kollanus and Koskinen 2009). Today, such a practice is more informal, being referred to as Modern Code Review (MCR) (Bacchelli and Bird 2013). It is often assisted and enforced by tools, such as Gerrit (Google 2017a).

The effectiveness of code review depends on different factors and, when it cannot provide expected benefits, it becomes a costly and time-consuming task (Czerwonka et al. 2015; Thongtanunam et al. 2016a). For example, if there is a time gap between the completion of a change and its review by a peer, the author may have its work partially blocked, possibly affecting the whole software release (Thongtanunam et al. 2015b). This lack of dynamism in the code review activity increases the work in progress of teams, as new tasks are started while waiting for the pending reviews. Furthermore, the context switching between coding tasks and reviews may also have a negative impact on developers’ work.

To understand the factors that positively and negatively affect the effectiveness of

code review, previous studies were performed, e.g. (Thongtanunam et al. 2015a; Baysal et al. 2016; Yang 2014; Bosu et al. 2015). Examples of investigated factors are the patch size, the nature of the change, or author’s company—that is, both technical and nontechnical factors have been investigated. Moreover, to evaluate effectiveness, different criteria have been adopted, such as the review duration and the number of defects found after code review. As a result, relevant conclusions regarding code review have been reached. For instance, developers from other teams provide fewer but more useful feedback than those from the same team (Bosu et al. 2015). Despite all the significant results obtained so far, code review has been investigated only to a limited extent in the context of geographically distributed software development (Sengupta et al. 2006), which is becoming increasingly common over the last decades. In the late 90s, researchers focused on enabling formal code inspections, which involve meetings, in distributed scenarios (Perpich et al. 1997; Stein et al. 1997). In modern code review, in contrast, tool support and asynchronous communication help deal with geographic distribution. However, the effects of geographic distribution on the outcomes of code review (such as duration or reviewer engagement) have not been explored. Recent studies of code review in distributed software development are limited to experience reports on code inspection (Meyer 2008).

We thus in this paper focus on exploring how both technical and non-technical factors influence a set of metrics that are indicators of the effectiveness of code review in the context of Distributed Software Development (DSD). We present the results of a mixed-method study in which we investigated the relationship between four influence factors—namely number of changed lines of code, involved teams, involved locations and active reviewers—and the effectiveness of code review. As there is no single objective metric that captures whether a review is effective, we measured and analyzed different review outcomes that can be seen as an indication of the review effectiveness, such as reviewer participation and number of comments. The study involved (1) an analysis of objective data collected from a software project; and (2) a survey with 50 practitioners with experience in code review. This study is an extension of our previously presented work (Witter dos Santos and Nunes 2017), which was complemented by the survey that allows us to compare the results obtained with both research methods.

The first part of our study, referred to as *repository mining*, is based on a large amount of data (8329 commits and 39,237 comments) extracted from the code review database of a project with 201 members during 72 weeks. The analysis of our results allowed us to conclude that a high number of changed lines of code tends to increase the duration of the review process with a reduced number of messages, while the number of involved teams, locations and participant reviewers generally improve the contributions from reviewers, but with a severe penalty to the duration. These results are consistent with those obtained in the survey regarding the influence of factors over duration and participation. However, participants’ opinion about the impact on contributions diverges from results obtained from historical data, mainly with respect to distribution.

The remainder of this paper is organized as follows. We first discuss related work in Section 2. We then provide details of our target project in Section 3, describing the code review process of our target project. Next, we describe our study settings in Section 4. The results of the first and second parts of our study are presented and analyzed in Section 5. A discussion regarding obtained results is presented in Section 6, followed by our conclusions, which are presented in Section 7.

**2 Related work**

Since the pioneering work of Fagan (1976) on formal code inspections, many researchers proposed approaches to improve this well-structured and phased form of code review (Parnas and Weiss 1985; Bisant and Lyle 1989; Martin and Tsai 1990). With the popularity of DSD, other researchers investigated how to make code inspections feasible when the involved people cannot physically meet in a particular location (Perpich et al. 1997; Stein et al. 1997). Despite its popularity among researchers and practitioners, formal code inspection and its variations have received less attention since the early 2000s (Kollanus and Koskinen 2009).

More recently, much work focusing on modern code review has been done, ranging

from studies that investigate what leads to successful code review to approaches that

recommend suitable reviewers. For example, in Balachandran (2013)’s approach, recommended reviewers are those that made the most recent changes in the portion of code to be reviewed. His approach was improved by Thongtanunam et al. (2014), for projects with specific characteristics, using the File Path Similarity (FPS), which takes into account previous changes with similar paths or file names. These approaches were extended by also considering similarity among past commit messages (Xia et al. 2015) and recent activity of the possible reviewers (Zanjani et al. 2016). Viviani and Murphy (2016) took another direction by prioritizing pending reviews for each reviewer instead of finding the best candidate reviewers for a given change. This is motivated by the fact that several projects have a high concentration of review requests in a small group of contributors (Yang 2014).

Despite all these significant contributions to the field of code review, it is crucial to understand the factors that influence the effectiveness of code review to, for example, provide foundations to improvements while making reviewer recommendations. Therefore, many studies focus on providing a deeper understanding of code review, and its influence factors (e.g. number of changed lines of code and experience of individuals) and outcomes (e.g. duration and discussion among reviewers). Although such studies are similar to ours, they do not focus on DSD. We next discuss technical and non-technical influence factors investigated in existing studies.

**Investigation of technical factors** Different correlations were studied involving technical factors. Thongtanunam et al. (2015a)’s study provided evidence that reviewers are less rigorous and find fewer defects on files with a high incidence of defects in the past, focusing on superficial aspects, such as coding standards rather than on functional aspects. In a more recent study (Thongtanunam et al. 2016a), the same authors identified that bug fixes typically receive the first feedback faster than implementations of new features. Moreover, they reported that changes with detailed and explanatory commit messages have lower stale rates, while those that are poorly described receive less attention of reviewers.

Focusing on the code review duration (in working days), a few influence factors were investigated. Bosu et al. (2015) concluded that the patch size affects the duration in most of the analyzed cases, while task priority in the release plan and the affected software components have only occasionally influenced some of the projects analyzed by Baysal et al. (2016).

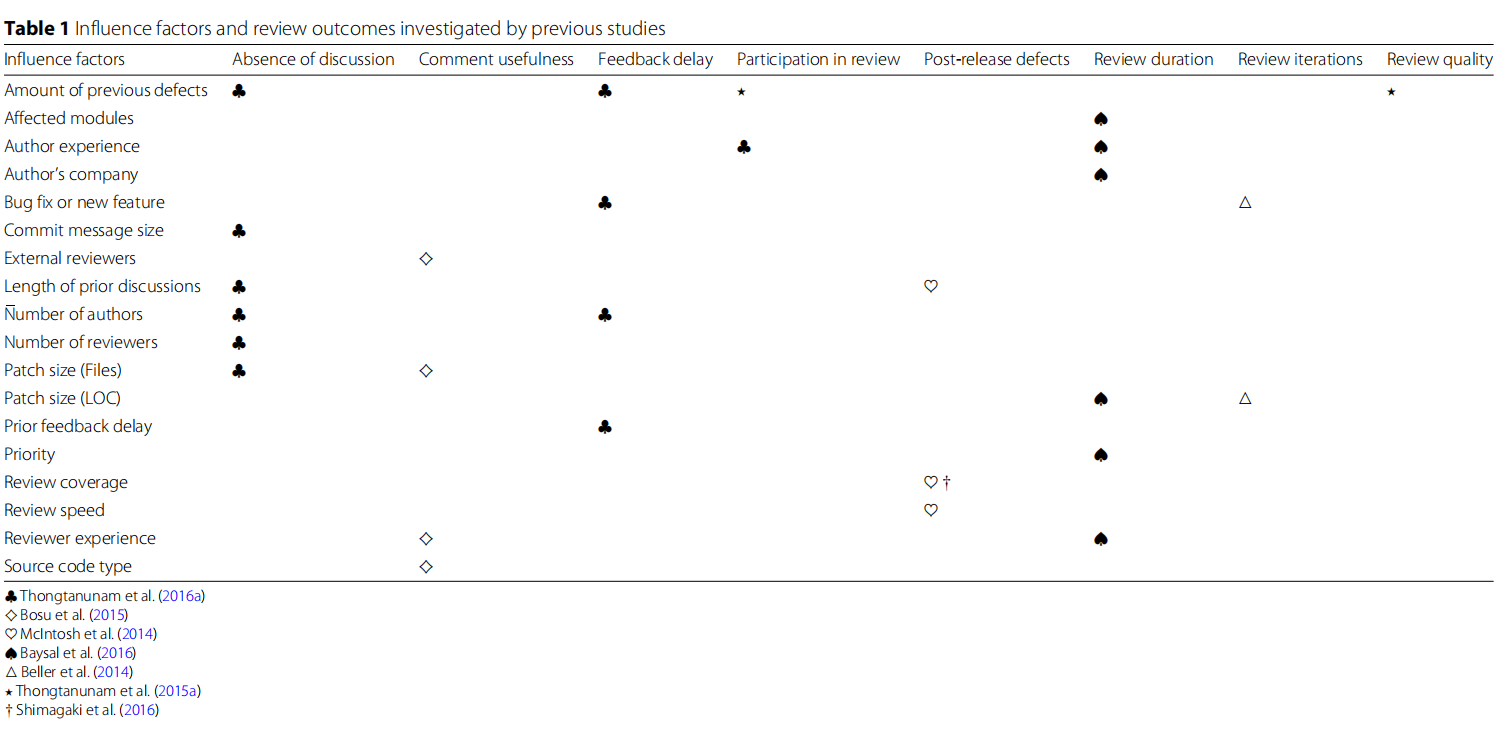
**Investigation of non-technical factors** Non-technical factors also received attention

recently. As stated by Czerwonka et al. (2015), the social network that naturally emerges inside the companies or projects should be considered as well as the specific reviewers’ skills and their availability and willingness to review. An analysis of the social network of three open source projects (Yang 2014) revealed that the most active reviewers have central roles in the social network of those projects and are frequently some of the most important contributors. Bosu et al. (2015) observed, in a particular organization, that 75% of the code review feedbacks come from members of the author’s team, but are slightly less useful than those from other teams. Baysal et al. (2016), in turn, pointed out that when multiple organizations contribute to the same project, the code review can take more time to be completed and have higher rejection rates depending on which organization is authoring or reviewing a patch, based on the analysis of several case studies.

The experience of authors of the code under review has also been pointed out as relevant in code review. Senior members of the company and those with recognized expertise usually receive more priority, faster and more detailed feedback, enabling a faster code review with better results for the quality (Baysal et al. 2016; Rahman et al. 2016). The experience of the reviewers is relevant as well, based on results of the investigation of a large company (Bosu et al. 2015)—the quality of provided feedback increased during the first year in the company and then stabilized in a *plateau*.

In a study involving three large open source projects, Thongtanunam et al. (2016a) also investigated non-technical factors, focusing on how the code review was affected by prior events on the files under review. Their conclusions are (1) files that received a slow initial feedback in the past will also likely receive slow feedback in the future; (2) files with more authors and reviewers in the past receive more attention; and (3) the number of changed files, directories and the length of the commit message are also important.

**Summary** Given that many factors that influence code review have been investigated, we summarize what each previous study analyzed in Table 1. Rows in this table consist of the examined influence factors, while columns represent the analyzed outcomes associated with code review. In cells, we list the studies that focused on the relationship between a given influence factor and outcome.



Some of these studies analyzed MCR targeting FLOSS (Free, Libre and Open Source Software) projects, such as OpenStack, Qt, and LibreOffice, which present DSD characteristics. However, we emphasize that most of these studies did not investigate the impact of distribution: factors associated with distribution were random variables rather than independent variables. For instance, Baysal et al. (2016) reported that some analyzed companies had co-located groups, while others used DSD, without treating this issue as a dependent variable. Similarly, Bosu et al. (2015) found that comments from other teams are slightly more useful, but without considering co-location or the number of involved teams.

As can be seen, different combinations of influence factor and outcome have been

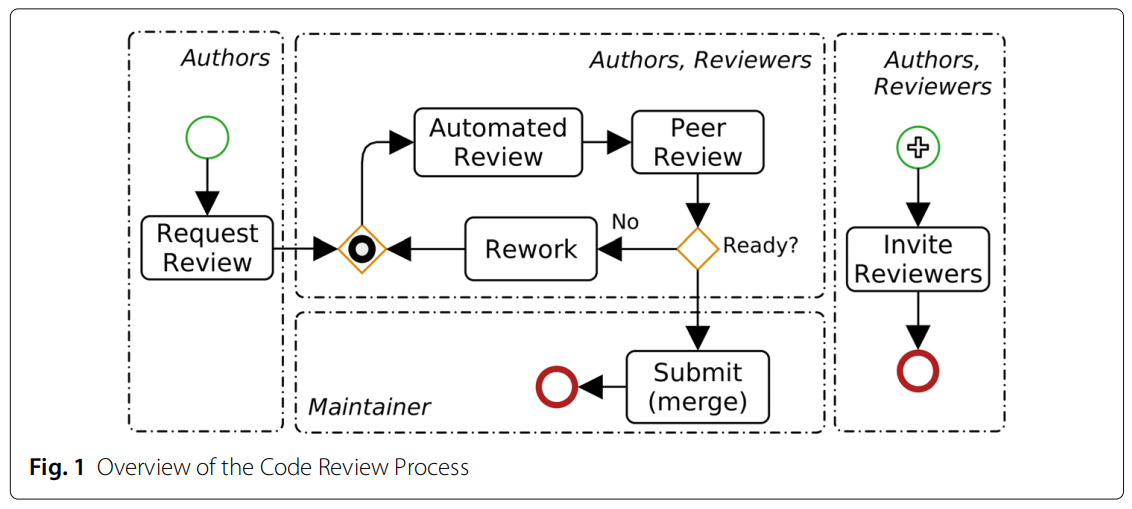
analyzed. Differently from previous work, our study focuses on DSD and, therefore, we focus on other influence factors, such as the number of involved cities and teams. Some of our investigated factors, e.g. patch size (LOC), have already been studied, but not in a DSD scenario. Moreover, we analyze four different outcomes of code review, which are described in next section together with other details of our study settings.

**3 Study subject**

Our study is based on the analysis of data collected from a (commercial) software project and developers from a single software development company. Due to the project size, we were able to collect a large amount of information regarding its code review. We next describe the code review process of the project, provide details about the collected data, and characterize the participants of our survey. No further information can be given due to a confidentiality agreement.

**3.1 Code review process**

We overview the code review process followed in the target project in Fig. 1. First, authors send a piece of code to be reviewed. Anyone can, at any point in time, invite reviewers or add itself as a reviewer, what would allow any (interested) developer to contribute. Moreover, in our target project, Gerrit is configured with the reviewers-by-blame plugin (Google 2017c), which automatically adds reviewers based on the last changes made on the files to be reviewed, as proposed by Balachandran (2013). The code is then analyzed by automated reviewers that check several quality criteria, such as compilation, cyclomatic complexity, lack of documentation, failed unit tests, among other static analysis and runtime verifications. This automated verification usually takes less than 15 min to execute and rejects the change if any critical test fails, so that the author can fix the reported issues. Human reviewers and authors can discuss, ask and provide suggestions for each line of code. Moreover, each reviewer can vote to summarize its feedback using one of the following values.



**Veto** The reviewer considers that the change cannot be integrated without fixing the

reported issues or answering questions made. This prevents the commit to be merged.

**Rejection** The reviewer recommends fixes before the change is merged.

**Neutral** The reviewer typically asks easy questions to be answered.

**Acceptance** The reviewer considers that, though the change it adequate, it needs more

reviews from other developers.

**Approval** Only maintainers of the module associated with the commit have this kind

of vote, as they are responsible for the module quality. Maintainers can perform

technical reviews, but must also verify that relevant developers are not missing in

the list of invited reviewers and that the overall state of the code review is adequate.

It is important to note that all invited reviewers, but the maintainer, are not obliged to provide feedback. Before approving the change, the maintainer of each module should consider if the most important reviewers already reviewed the code. In the end, the piece of reviewed code is submittable if all the following conditions are satisfied: (i) there is no rejection from automated reviewers; (ii) there is no veto; and (iii) the maintainer has approved the change. If all these conditions hold, the maintainer is able to merge the change into the destination branch.

**3.2 Analyzed data**

The target project of this study involves the development of an operating system for embedded systems of routers and switches, using the C, C++, and Yang (Internet Engineering Task Force (IETF) 2017) languages. This project has a total of 269 repositories, from which 63 are dedicated to test automation, using Python,Vagrant, and Ansible. We consider that the operating system code and its tests are part of the same project, as the developers implement both firmware and tests for each task. All repositories are configured to reject the merges without code review.

The mined data refers to a period of 72 weeks, starting in October 2014. In the collected data, we had a total of 11,109 code reviews. After filtering these data (see next section), we obtained 8329 code reviews associated with 39,237 comments (an average of 4.7 comments by review). Such code reviews are associated with: (i) 201 experienced developers; (ii) 4 development locations in 4 different cities in the same

country and time zone; and (iii) 21 different teams. Members of a given team work on

the same location, i.e. there are no distributed teams. All teams are organized as feature teams and use Scrum with three-week sprints to release new software versions

every three months. A continuous integration pipeline is used to run functional tests on several test environments that contain network topologies with real products and

emulators.

**3.3 Survey participants**

Our goal by conducting a survey with software developers of the same company is to be able to compare the perceptions of developers with concrete objective data. The survey was conducted in January 2018, when we randomly invited 80 developers to participate. From them, 50 voluntarily provided a response within a week (response rate of 62.5%). However, 5 participants reported (very) low experience in code review (as author or reviewer) and these were discarded because they could provide unreliable answers. Because our survey involved other projects and a different time period of the first part of the study, many participants were not developers of our target project during the period in which we collected data. However, the projects of which participants are members use Gerrit to implement and reinforce modern code review practices. Although there may be divergences in the software development process as a whole, all these projects adopt the code review workflow described in Fig. 1.

Table 2 provides detailed information about 45 participants, including age, education and experience. Answers that indicate the participant experience—used solely for the purpose of characterizing our sample—were self-reported based on the participant’s subjective view. We can see that 97.8% of the participants reported medium to very high experience with projects with multiple teams, whereas 93.3% reported medium to very high experience with projects with multiple locations, suggesting that they have experience in DSD.

**4 Conclusion**

Code review is an important static verification technique for improving software quality as well as promotes knowledge sharing within a software project. To identify the scenarios in which code review in fact succeeds, many studies investigated the relationship between different factors and the review outcomes. However, there is limited investigation of the situations in which modern code review is effective in the context of distributed software development when developers and reviewers are spread into geographically distant development locations.

In this paper, we presented the results of a mixed-method study, composed of two parts. In the first part, repository mining, we extracted a large amount of code review information from a software project whose aim is to develop an operating system for embedded systems. This project involves 201 developers, spread into 21 teams located in 4 different cities. We investigated how the patch size (in terms of lines of code), the number of teams, the number of locations and the number of active reviewers influence the duration, reviewer participation and comment density (general and by reviewer) of the review. We found evidence that the duration of the code review is highly affected by all investigated factors—the higher they are, the longer the review process. Similarly, the participation of reviewers is negatively affected in all cases, but mainly by the number of lines of code to be reviewed. The density of review comments is higher when a relatively small patch size is reviewed by other reviewers of teams or locations other than that of the author. The density of review comments per reviewer is positively affected by the number of involved locations and negatively affected by the other factors.

In the second part of the study, we conducted a survey to collect data about the perceived effects of the four investigated influence factors over code review outcomes

(duration, participation and total number of comments). We obtained 50 responses from software developers with relevant professional experience in DSD projects with modern code review practices. We found evidence that higher values of the influence factors have similar effects on the analyzed code review outcomes. Duration and participation are negatively affected; the total number of comments is negatively affected by patch size, teams and locations, but is positively affected by the number of active reviewers.

Due to the large amount of data investigated in our study, we could not identify particular occurrences of code review that could help us to make other analyses and further explain our data. Even if this was possible, given that we used data from the past to have complete reviews, developers would potentially not remember specific cases. Our study had, however, gave us insights for future investigations. First, we aim to perform an observational study involving developers and managers that will allow us to verify if our conclusions based on the present study hold. Second, further analyses can be made using code review data. For example, the proportion of votes (vetoes, rejections, approvals and neutral feedback), the influence of the number of contributions as author or reviewer (overall and in the same module or file) and other reviewers’ characteristics are interesting issues to be investigated.

As the patch size demonstrated to be a prominent influence factor, we also plan to analyze other forms of complexity and effort during code review, assuming that reviewing ten lines added to a complex module requires more effort than reviewing ten lines added to a simple module. It is possible to analyze the influence of other indications of complexity, such as the total number of classes, files and LOC as well as the total cyclomatic complexity.

For modular systems, some influence factors arise from the relations among the modules and from the role of each module. For instance, the number of dependent modules could influence the participation in or the duration of the code review, and critical infrastructure modules might have different code review dynamics when compared to modules that implement user interfaces. Therefore, we plan to analyze the influence of architectural aspects of the modules in the code review.

Finally, we considered many metrics to indicate the effectiveness of the review and

aim to investigate whether it is possible to derive a single metric that captures review

effectiveness by combining different review outcomes.