NAIST-IS-DD1761014

Doctoral Dissertation

Understanding Code Review Resolutions at a Fine-Grained Level

Toshiki Hirao

January 28, 2020

Graduate School of Information Science
Nara Institute of Science and Technology

A Doctoral Dissertation
submitted to Graduate School of Information Science,
Nara Institute of Science and Technology
in partial fulfillment of the requirements for the degree of
Doctor of ENGINEERING

Toshiki Hirao

Thesis Committee:

Professor Kenichi Matsumoto	(Supervisor)
Professor Hajimu Iida	(Co-supervisor)
Associate Professor Takashi Ishio	(Co-supervisor)
Assistant Professor Raula Gaikovina Kula	(Co-supervisor)
Assistant Professor Shane McIntosh	(Co-supervisor, McGill University)

Understanding Code Review Resolutions at a Fine-Grained Level*

Toshiki Hirao

Abstract

Code review is a software quality assurance approach to ensure that a software system is safe and reliable enough. Recent software developments employ the Modern Code Review (MCR) approach, where developers informally discuss the quality of software systems via online web tools. The MCR has mainly been driven by five motivations i.e., defect finding, code improvement, knowledge transfer, context understanding, and team transparency. Especially, in terms of resolving code reviews, the concept of divergence (i.e., different thoughts are provided asynchronously) drives developers to transfer their technical knowledge. Moreover, the concept of openness (i.e., any developer can participate in code reviews) allows developers to join and leave discussions.

The divergence and openness may cause negative impacts on code review resolutions i.e., tense discussions and the lack of participation. Developers may leave communities due to tense discussions, while the lack of participation is associated with the poor software quality. Prior work discovered those negative impacts on code review quality; however, little is known about the impacts of divergence and openness on code review resolutions (i.e., from an initial submission to final integration decision). We presume that our fine-grained studies, where the thesis focuses divergence and openness on reviewer commenting, can bring practical implications to deal with the diversity and openness on code review resolutions.

The thesis set out to perform three empirical studies at a comment-level. We first explore how reviewer discussions become divergent. The study shows that

*Doctoral Dissertation, Graduate School of Information Science, Nara Institute of Science and Technology, NAIST-IS-DD1761014, January 28, 2020.

external dependencies across multiple review artifacts are highly likely to be a root cause of divergent discussion, suggesting that developers should be aware of both internal and external issues to resolve divergent discussions. Second, we study what factors play an important role in keeping participation until the end of reviews. The study shows that experienced reviewers should be invited to have much participation. Third, we investigate the impact of review linkage on code review analytics. The study shows that taking review linkage into account improves code review analytics, suggesting that practitioners should be aware of the potential that their reviews may be linked to other reviews. Overall, the thesis shows practical implications to have a deep understanding of the mechanism of code review resolutions at a comment-level. Moreover, the thesis also provides generalized suggestions that can be applicable to other software projects.

Keywords:

Software Engineering, Modern Code Review, Mining Software Repository, Natural Language Processing, Data Mining, Big Data Analytics, Machine Learning

List of Major Publications

Journal paper

1. Toshiki Hirao, Raula Gaikovina Kula, Akinori Ihara, and Kenichi Matsumoto, "Understanding Developer Commenting Trends in Code Reviews," The IEICE Transactions on Information and Systems, Vol.E102-D, No.12, pages 2423-2432, Dec. 2019. (Section 4)

International Conference

1. Toshiki Hirao, Shane McIntosh, Akinori Ihara, and Kenichi Matsumoto, "The Review Linkage Graph for Code Review Analytics: A Recovery Approach and Empirical Study," In Proceedings of the 27th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, pages 578-589, August 2019. (Section 5)

Domestic Conference

1. Toshiki Hirao, Akinori Ihara, Yuki Ueda, Daiki Katsuragawa, Dong Wang, Kenichi Matsumoto, "Towards the Feasibility of Chat-Bot That Provides Feedback to Software Developers (In Japanese)," Software Engineering Symposium Workshop, 2017. (Section 3)

Award

1. **SIGSE Distinguished Paper Award:** Toshiki Hirao, Shane McIntosh, Akinori Ihara, and Kenichi Matsumoto, "The Review Linkage Graph for Code Review Analytics: A Recovery Approach and Empirical Study," In Proceedings of the 27th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, pages 578-589, August 2019. (Section 5)

Contents

1. Introduction **1**
1.1 Problem Statement 2
1.2 Thesis Overview 4
1.3 Thesis Contributions 5
1.4 Thesis Organization 6

2. Motivation **7**
2.1 Modern Code Review Process 7
2.2 Literature Review 8
2.3 Results 9
2.4 Summary 17

3. The Mechanism of Divergent Discussion at a Fine-Grained Level **18**
3.1 Introduction 18
3.2 Case Study Design 21
3.3 Quantitative Results 25
3.4 Qualitative Results 29
3.5 Practical Suggestions 36
3.6 Threats to Validity 37
3.7 Conclusions 39

4. The Important Factors on Keeping Participation at a Fine-Grained Level **41**
4.1 Introduction 41
4.2 Methodology 43
4.3 Results 49
4.4 Practical Suggestions 56
4.5 Threats to Validity 58
4.6 Conclusions 59

5. The impact of Review Linkage on Code Review Analytics **60**
5.1 Introduction 60
5.2 Study Preparation 62

5.3 Review Graph Extraction (RQ1) 64
5.4 Qualitative Analysis of Review Links (RQ2) 68
5.5 Automated Link Classification (RQ3) 77
5.6 Linkage Impact Analysis (RQ4) 80
5.7 Practical Suggestions 83
5.8 Threats To Validity 84
5.9 Conclusion 85

6. Conclusions & Future Work **86**
6.1 Contributions 86
6.2 Opportunities for Future Work 87

Acknowledgements **90**

References **92**

Appendix **105**

A. The Mechanism of Divergent Discussion at a Fine-Grained Level **105**

List of Figures

1 The overview of the thesis sections 3
2 An overview of the Modern Code Review Process 7
3 An overview of our approach to classify patches with divergent scores. 23
4 Examples of review score patterns that we consider divergent and non-divergent. 24
5 The revision number that divergent scores appear on and revision number that the final decision is made in. 30
6 The distributions of revision number that divergent scores appear on and revision number that the final decision is made in. 31
7 Overview of our methodology. 43
8 The distributions of number of comments per a review. 47
9 The distributions of number of words per a general or inline comment. 48
10 The evolution of the (i) number of total comments per a developer and (ii) number of total words per a developer throughout a studied timeframe. 52
11 Monthly linkage rate in the studied communities. 68
12 An example of a review link from Review #126831 in Nova. . . . 70
13 The Patch Dependency subgraphs. 72
14 The Broader Context subgraph. 74
15 The Alternative Solution subgraph. 74
16 The Version Control Issues subgraph. 75
17 The Feedback Related subgraph. 76

List of Tables

1 An overview of the subject systems. The white and gray rows show the overview of each subject system and each community, respectively. . . . 22
2 The number and proportion of each potentially divergent pattern in the four subject systems. . . . 25
3 The integration rate of each divergent score pattern in the four subject systems. . . . 26
4 The number of reviewers in reviews that receive divergent scores and reviews that do not in the four subject systems. . . . 28
5 The N→SP and P→SN rates of the five most active reviewers in the four subject systems. . . . 28
6 The reasons for abandoned patches with and without divergence. . . . 34
7 The rate of outsider-dominated discussions. . . . 34
8 An overview of our five studied systems. . . . 45
9 The description of our selected features in Patch Property, Human Experience and Project Management segments. . . . 46
10 The statistics of numbers of total comments, general comments and inline comments for each system. . . . 50
11 The correlations between the number of general and inline comments and the number of words in general and inline comments. . . . 51
12 The standardized coefficient of each feature for general comments and words in the general comments. The blue colour cell depicts the most impactful metric. . . . 54
13 The standardized coefficient of each feature for inline comments and words in the inline comments. The blue colour cell depicts the most impactful metric. Note that the standardized coefficient of Patch Churn (0.145) is larger than of Reviewers Experience (0.138) in Eclipse. . . . 55
14 An overview of our subject communities. . . . 63
15 Review graph characteristics in the subject communities. . . . 65
16 The frequency of the discovered types of review linkage in OpenStack Nova and Neutron. . . . 71

17 The performance of our five link category classifiers, all of which outperform the ZeroR and random guessing baselines in all cases. 79
18 The mean performance scores of cHRev [100] and our proposed link-aware reviewer recommenders at different prediction delays. The bulk of the performance improvement is achieved in the no delay (0 hour) setting. 80
19 The taxonomies of abandoned divergent and non-divergent patches. 105

1. Introduction

Code review is a well-known practice to assess the quality of software systems where developers discuss the quality of changed code. Many software communities have benefited from the code review activities. For example, code reviews can reduce bugs from software products [23]. High quality of code reviews can eliminate the risk of anti-patterns in software systems [58]. Moreover, code reviews affect the effectiveness of defect removing process [45].

The code review practice has evolved from traditional code inspection to modern code review. Traditionally, Fagan-style inspection—a formal software inspection, had been performed in a variety of software developments [23]. Fagan inspection is a process in which developers participate in an in-person meeting to evaluate the quality of source codes. This inspection comprises of six operations (i.e., Planning, Overview, Preparation, Inspection meeting, Rework, and Follow-up). Despite the highly structured inspection, the process is known as time-consuming and expensive. Moreover, it is hard to apply the inspection in globally distributed developments because of the physical constrains of in-person meetings. In contrast, the Modern Code Review (MCR) has been recently practiced in place as online communication platforms are developed. The MCR is a lightweight process where its discussions are informal, tool-based, and asynchronous [4]. The MCR process comprises of patch upload, reviewer assignment, discussion, and integration. Companies such as Microsoft, Google and several OSS projects have successfully adopted the MCR process [76].

The MCR has mainly been driven by five primary motivations i.e., defect finding, code improvement, knowledge transfer, context understanding, and team transparency [4, 71]. Those motivations are associated with code review quality and resolution. For example, patches that attract many comments are less likely to be integrated into codebase [89]. In code review resolutions, the concept of divergence (i.e., different comments are posted online) drives developers to transfer their technical knowledge among them. Moreover, the concept of openness (i.e., any developer can flexibly participate in code reviews) allows developers to join and leave code review discussions [84].

Although the diversity and openness play an important role in the MCR, there have been still negative impacts on code review resolutions. For example, a

review can collect diversified opinions from multiple developers. However, such a review discussion may result in tense environment [76] and cause developers leave their projects. For instance, in review #12807 from the QtBase project,[1] two reviewers who provided opposed opinions argued that the scope of the patch must be expanded before integration could be permitted. Since the agreement between those opposed reviewers has never been reached for one month, the patch author abandoned the patch without participating in the discussion. Despite making several prior contributions, this is the last patch that the author submitted to the the project. In openness aspect, since anyone can join and leave the community, the lax code reviewing practices may happen [84]. Indeed, patches have the risk of poor reviewer participation due to openness [87]. Moreover, human factors (e.g., developer experience) are associated with the likelihood that a developer who is invited to review a changed code will actually join the review [75].

1.1 Problem Statement

The diversity and openness have been little studied at a fine-grained level (i.e., a comment-level). For example, disagreements in reviews increase a developer' s likelihood of leaving the project [37]. Filippova and Cho [24] reported that those tense environments negatively affects the performance of the project teams. However, little is known about how those tense environments occurred and were resolved as developers provide their comments.

When taking a look at review participation, code property and human factors play an important role in determining whether or not a review participate in reviews [75, 87]. However, important factors that can keep developers participating until the end of reviews are unclear. If reviewers leave projects half way through completing their reviews, this leaves review activities uncompleted state. Such a little participation is estimated to lead to components with additional post-release defects [54]. Hence, a fine-grained study at a comment level is needed to explore important factors that let participants stay until the end of reviews.

To eliminate the drawbacks of diversity and openness, prior work proposed code review analytics that can support code review resolutions [65, 66, 88]. For example, reviewer recommendations have been proposed to automatically select

[1] https://codereview.qt-project.org/#/c/12807/

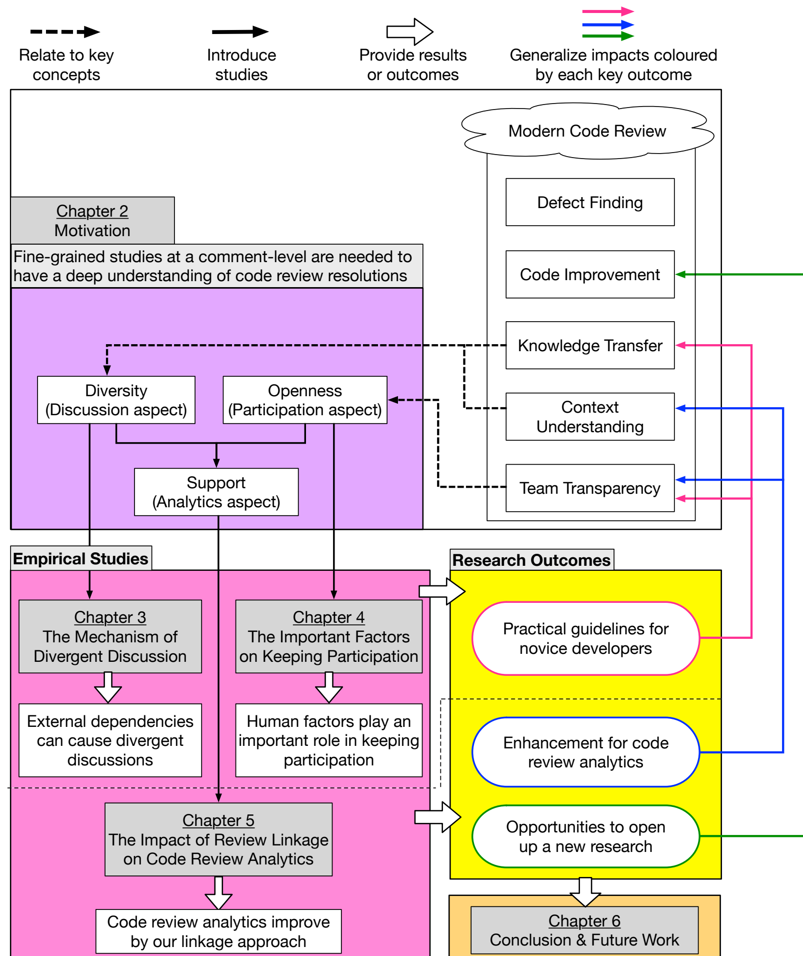

Figure 1: The overview of the thesis sections

appropriate reviewers. Review outcome predictions are also developed to project the likelihood of integration based on the past review history information. However, little studies have taken into consideration the impact of comment-level analyses that we introduced above. Hence, we presume that there is a room for improvement on those prior code review analytics.

A fine-grained study that explores the impacts of diversity and openness at a comment-level is necessary to have a deep understanding of the MRC. Our empirical studies at a comment-level can provide practical implications and show the potential to enhance code review analytics.

1.2 Thesis Overview

We provide an overview of this thesis. Figure 1 outlines the overview of this thesis flow. This thesis focuses its empirical studies on code review resolutions. Therefore, We first motivate our thesis as following:

- **Section 2**— The section introduces the overall process of the Modern Code Reviews. Then, prior studies that are related to code review resolutions are shown. Those prior work mainly comprised of three dimensions (i) Reviewer Discussion, (ii) Review Participation, and (iii) Code Review Analytics.

We then present the main empirical studies that are split into three sections. Each section shows its corresponding empirical study ("Empirical Studies" box) that has the compelling potential outcome ("Research Outcomes" box).

- **Section 3**— We perform empirical studies for analyzing the mechanism of divergent discussions. Prior work has studied the impact of divergent discussions [24, 37]; yet, practical implications for understanding how those divergent discussions occurred at a comment-level have been unclear. Our thesis quantitatively and qualitatively analyzed how divergent discussions occurred and were addressed by studying two popular communities.

- **Section 4**— The section studies factors that impact keeping participation until the end of reviews. In the MCR process, any participant can provide general comments and inline comments. A general comment is mostly placed on a general discussion thread, while an inline comment is provided on a specific line of code. We would like to have a deep understanding of the important factors across three dimensions (i.e., patch property, human experience, and management) on code review resolutions.

- **Section 5**— We study the impacts of the practical implications that we gained (from Section 3 and Section 4) on code review analytics. Prior work showed that selecting appropriate reviewers is an important procedure to successfully resolve code reviews. Hence, a variety of reviewer recommenders [65, 66, 88, 95] and outcome predictoin models [28, 39, 40] have been proposed. Our fine-grained studies at a comment-level suggested that

reviews may be linked to other reviews, which can impact code review resolutions (see Section 3 and Section 4). However, those previous recommenders and predictors have not treated review artifacts by taking the linkage aspect into consideration. Hence, we set out to explore what types of links exist, and analyze the impact of those links on code review analytics.

Finally, Section 6 provides our conclusion that includes the main research results and contributions of this thesis. We also provide some opportunities for future code review research.

1.3 Thesis Contributions

This thesis performed empirical studies at a comment-level. In a nutshell, we provided practical implications for code review resolutions and showed the impact of those implications on code review analytics.

- Software organizations should be aware of the potential for divergent discussion, since patches with divergent scores are not rare and tend to require additional personnel to be resolved. (Section 3)
- Automation could relieve the burden of reviewing external concerns, such as "Integration Planning" and "Unnecessary Fix" issues. (Section 3)
- The numbers of comments vary among reviews and across our studied systems. Reviewers change their behaviours in commenting as a system evolves. (Section 4)
- The review participation tends to be influenced by reviewer experience and patch property size. (Section 4)
- Linkage is not uncommon in six studied software communities. (Section 5)
- Adding linkage awareness to code review analytics yields considerable performance improvements, suggesting that review linkage should be taken into consideration in the future MCR studies and tools. (Section 5)

1.4 Thesis Organization

The remainder of this paper is organized as follows. Section 2 describes motivation and related work of this thesis. Section 3 presents empirical studies that investigate divergent discussion. Section 4 presents empirical studies that analyze the important factors on review participation. Section 5 presents empirical studies that demonstrate the impact of review linkage on code review resolution. Finally, Section 6 draws conclusions and contributions of our thesis.

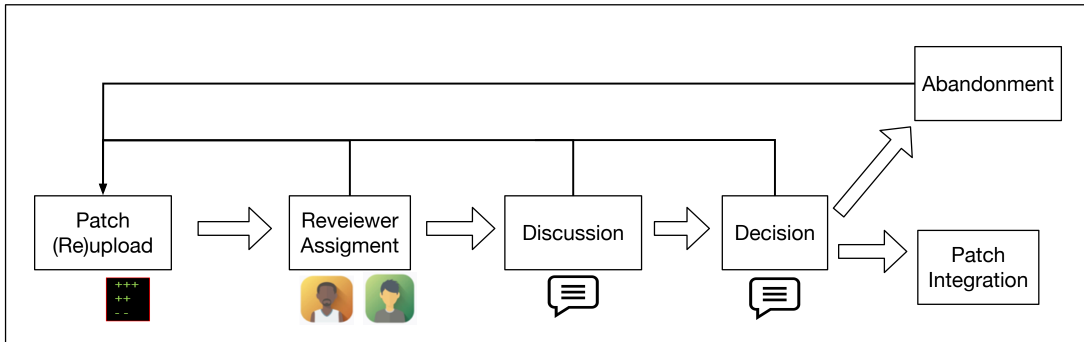

Figure 2: An overview of the Modern Code Review Process

2. Motivation

The section introduces the overall process of the MCR. Then, prior work that are related to code review resolutions are shown.

2.1 Modern Code Review Process

The MCR manages code contributions (i.e., patches) using a rigourous lightweight process that is enabled by Gerrit. Figure 2 provides an overview of that code review process, which is made up of the three segments.

- **Patch (Re)upload.** A patch author uploads a new patch or revision to Gerrit and invites a set of reviewers to critique it by leaving comments for the author to address or for review participants to discuss. Reviewers are not notified yet about the patch's details. Before reviewers examine the submitted patches, sanity tests verify that the patch is compliant with the coding and documentation style guides, and does not introduce obvious regression in system behaviour (e.g., compilation error). If the sanity tests report issues, the patch is blocked from integration until a revision of the patch that addresses the issues is uploaded.

- **Reviewer Assignment.** When submitting reviews onto the system, developers also select other developers who are suitable for reviewing the change of code. Reviewers (i.e., other team members) are either: (1) appointed

automatically based on their expertise [66, 88, 100]; or (2) invited by the author [36, 54, 88];

- **Discussion.** After a submitted patch passes sanity testing, reviewers examine the patch. The reviewer is asked to provide feedback and a review score, which may be: `+2` indicating approval and clearance for integration, `+1` indicating approval without clearance for integration, `0` indicating abstention, `-2` indicating blockage of integration, or `-1` indicating disagreement without blocking integration. Only those reviewers on the core team have access to `+2` or `-2` scores.

- **Decision.** Through commenting step, if the change needs to be revised, the patch author will revise his/her change and re-upload the revision. If the revision requires a new reviewer, the reviewer will be assigned. Then, the commenting step starts for this revision.

- **Patch integration or Abandonment.** Once the patch passes integration testing, the patch is integrated into the upstream (official) project repositories. This step still may fail due to conflicts that may arise due to integration timing. If the patch is eventually considered as not sufficient to be integrated, the patch will be abandoned.

2.2 Literature Review

The MCR concept has been defined in [4]. Since then, a lot of studies have analyzed the mechanism of the modern code reviews. To survey those previous studies, we performed literature reviews. The approach consists of three processes: (1) including recent papers that have studied the MCR process, (2) excluding papers that are out of this thesis scope, and (3) snowballing supplemental papers. We explain those steps each by each as following:

- **Including recent papers that have studied the MCR process.** We try to collect papers that are related to the MCR. To do so, we surveyed papers that have been published since the concept of the MCR was established by [4] until 2019. Those papers were collected from the top international conferences (i.e., International Conference on Software Engineering, The

ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, The IEEE/ACM International Conference on Automated Software Engineering, The International Conference on Software Maintenance and Evolution, and IEEE/ACM International Conference on Mining Software Repositories) and academic journals (i.e., IEEE Transactions on Software Engineering, Empirical Software Engineering, ACM Transactions on Software Engineering and Methodology, and Information and Software Technology) in software engineering field.

- **Excluding papers that are out of this thesis scope.** We surveyed papers that were collected by the inclusion step to retrieve papers that fall into the scope that the thesis focuses on. In details, papers that have focused on none of review discussion, reviewer participation, and code review analytics aspects are excluded from this thesis.

- **Snowballing for supplementary papers.** The results of those papers that were selected in the exclusion step may have been complemented by their related work. Therefore, we explored those related papers and add them into the list of our related work.

2.3 Results

We first describe general motivations for studying the MCR processes based on prior related studies. Then, we break those studies into three segments that are reviewer discussion, review participation, and code review analytics. Finally, we summarize our key motivations for this thesis.

General motivations. The MCR is known as an effective process and an easy-to-follow environment where a software community can easily create their code review project in a code review tool such as Gerrit. Despite the MCR brings developers five primary motivations (i.e., defect finding, code improvement, knowledge transfer, context understanding, and team transparency) [4], it has been considered having a room for improvement. For example, developers can have benefits beyond finding defects. The MCR provides both authors and reviewers with opportunities to improve code style and the quality of problem solution. This is

because code review tools assist developers in joining code review activities and communicating with other developers easily through online communication tools.

Resolving code reviews through those current code review tools still have some difficulty to manage the flexible review processes of the MCR. Although those code review tools have grown over recent time, a richer communication is still needed because the current review tools are performed in an asynchronous environment. Such an insufficient communication may trigger misunderstanding of context and increase the risk of excessively diversified discussions. Indeed, an insufficient communication is associated with poor software quality [55]. In addition, an unsolved disagreement due to excessively diversified discussion can increase a developer's likelihood of leaving their project [37] and negatively affect software community performance [24]. Aside from the divergence, the code review tools are mostly usable in public so that anyone can participate in their favorite review discussion. From the alternative perspective, the openness can provide with the lax participation practice, where developers can contribute to or leave projects as desired because they do not have the responsibility for their contributions. This is considered a significant problem that might induce poor software quality and affect code review performance [87]. Those related studies show that studying the divergence and openness of review discussions plays an role in successfully managing software communities in the MCR environment.

Prior studies have shown various supportive analytics for code review resolutions. A popular research topic in those support tools is a reviewer recommendation [88, 100]. Selecting appropriate reviewers is important to evaluate the change of codes with high quality. However, traditional code review analytics have treated review artifacts without considering the impacts of divergent and openness aspects. The impacts of divergent and openness on code review analytics could be significant so that it is implied that code review analytics may have the necessity to take those impacts into consideration.

Below, we introduce related work that follows general motivations above. We break it down into three key segments i.e., reviewer discussion, review participation, and code review analytics. For each segment, we state our key motivation that supports our empirical studies.

Reviewer Discussion

Recent work has analyzed the discussion process in modern code review. Jiang et al. [40] showed that the amount of discussion is an important indicator of whether patch will be integrated into the Linux kernel. Tsay et al. [89] showed that patches that attract many comments are less likely to be integrated into codebase. Moreover, Thongtanunam et al. [85] have argued that the amount of review discussion that was generated during review should be considered when making integration decisions. Zanaty et al. [99] showed that patches that generate code design-related discussion are less likely to be integrated into codebase. They argued that it is because that 67% and 65% of the design-related discussions observed in their study are due to unawareness of the broad context of the codebase in OpenStack projects. Indeed, Tao et al. [83] found that patch design issues like suboptimal solutions increase the patch rejection rate in the Eclipse and Mozilla projects. Moreover, Tao et al. reported that patches are rejected when they are (1) problematic in implementation, (2) with low readability and maintainability, (3) deviating from the scope, (4) affecting development schedule, and (5) suffering with the lack of communication.

Deeper analysis has found that there are surprisingly few defects caught during code reviews. Tsay and Dabbish [90] showed that reviewers are often concerned with the appropriateness of a code solution, and often provide alternative solutions during discussion. Czerwonka et al. [19] found that only 15% of code reviews at Microsoft discuss and address defects. Rigby and Storey [74] showed that review discussions often tackle broader technical issues, project scope, and political issues. Mäntylä and Lassenius [53] found that 75 maintainability issues are raised for every 25 functional defects raised during code review discussions of student and industrial projects. Beller et al. [11] found a similar 75:25 ratio in the review comments that are addressed in industrial and open source projects. Indeed, McIntosh et al. [54, 55] reported that lacking review discussion (or participation) is negatively associated with software quality with higher post-release defect counts.

During review discussions, reviewers may argue patches in their various aspects. Hirao et al. showed that the final decisions of patches with divergent scores do not always follow a simple majority rule [35] and that patches with divergent

scores take a longer time to review [36]. Wang et al. [91] found that opposing views provides benefits (e.g., knowledge sharing) to the OSS projects, whereas it also has potential to lead to negative consequences. Indeed, Huang et al. [37] showed that disagreements in review increase a developer's likelihood of leaving the project. Filippova and Cho [24] showed that review disagreements negatively affects the performance of the project teams. Moreover, Thongtanunam et al. [85] showed that review discussions with large reviewer score discrepancies share a link with files that are defective in the future. Wen [93] found that as communities mature, the number of comments that a reviewer provides tends to decrease.

Prior work reported that divergent discussion may occur in reality; however, little is known about how divergent discussion occurred and was resolved during developer commenting. Pascarella et al. [61] showed that information such as an alternative solution, rational, and correct understanding is required to perform code reviews. Ebert et al. [21] also showed that when developers get confused in reviews, most of those confusions are caused by the missing information about non-functional requirements, and the lack of familiarity with code. Ebert et al. also reported that those confusions can be the cause of delaying integration time and decreasing review quality. However, the mechanism of divergent discussion at a comment-level is still unclear. We contribute to this growing body of knowledge by quantitatively and qualitatively analyzing code reviews in two large open source communities with a focus on how often reviews with divergent scores occurred and were treated.

Practical implications for divergent discussions at a comment-level are unclear. A fine-grained analysis that investigates the mechanism of divergent discussions and how developers can deal with those discussions is necessary.

Reviewer Participation

Code property plays an important role in patch integration. Mishra and Sureka [57] found that patch churn (i.e., the sum of the number of lines added and deleted in a patch) is the most important variable that impacts the code review effort towards the patch acceptance. Tao et al. [83] found that patches that include incomplete fixes are often linked with patch rejection in the ECLIPSE and MOZILLA projects.

Weißgerber et al. [92] reported that small patches (i.e., at most 4 changed lines) are likely to be integrated at the higher rate than average, while huge patches are significantly at the lower rate of acceptance than average. Baysal et al. [9] showed that patches that are (i) labeled as high priority in their reviews and (ii) placed at the front of the review queue, tend to be integrated into codebase. Jiang et al. [40] reported that the more popular a subsystem that a patch modifies is, the less likely that the patch will be integrated. They mentioned that since a popular subsystem relates to a lot of codes, the popular subsystem always has a new requirement to be satisfied, making its maintenance more difficult.

Code property can impact software quality after patches are integrated. Porter et al. [62] showed that patch size is correlated with number of defects. Similarly, Hassan [32] showed that the entropy of the source code changes impacts defect-proneness of code files. Panichella et al. [60] showed that when code property had warnings that had been raised by static analysis tools, 6%–22% of those warnings were removed during code reviews. Thongtanunam et al. [87] also showed that patch churn and the length of a patch description influence reviewer involvement. Moreover, software quality is affected by not only property metrics, but also process metrics. Rahman and Devanbu [64] and Kamei et al. [44] showed that process metrics are more useful in assessing the quality of source code. Rahman and Devanbu [63] also reported that the lines of changed code that are actually modified are strongly associated with the single developer contribution in code.

Prior work has found important factors that impact code review participant. Weißgerber et al. [92] shows that small patches are likely to receive faster responses. Indeed, an interviewee in Rigby et al. [73] mentioned that the size of code change should be short, making review processes easier. Morales et al. [58] shows that components that contain a high proportion of changes with hastily-reviewed may exhibit anti-design patterns (e.g., God Classes and Code duplication). Bosu et al. [14] shows that the likelihood of being vulnerable in code increases with number of lines that are changed because large changes tend to make review processes more difficult. Gousios et al. [28] empirically reported that number of files that are changed per a review is generally less than 20, while number of total lines that are changed is mostly less than 500. Jiang et al. [40] found that reviewing time is impacted by technical and non-technical factors

(e.g., submission time, number of modified subsystems, developer experience). Kononenko et al. [47] qualitatively analyzed how developers feel that size-related factors (e.g., patch size and number of modified files) are the most influential factors for reviewing time. Thongtanunam et al. [87] found that feedback delay of prior patches has a relationship with the likelihood that a patch will receive slow initial responses.

Prior work have studied various factors that impact review participation. Indeed, Thongtanunam et al. [87] showed that code property and human factors are associated with the likelihood of attracting at least one reviewer involvement. However, little is known about whether those important factors also impact participation from initial submission to the end of the review.

Little is known about how code review-related factors impact review participation until the end of reviews. A fine-grained study of review participation is also needed to have a deep understanding of code review resolutions.

Code Review Analytics

A patch author needs to select reviewers who are responsible for his/her change of code to have the change integrated into codebase. Aurum et al. [3] have argued that reviewers should have a deep understanding of the related source code towards patch integration. Jiang et al. [40] reported that developer experience is associated with the integration rate. Indeed, they showed that developers who have experienced a commit that was integrated into Linux Torvalds' codebase in their past development tend to have their patches integrated more than those who do not. In addition, they suggested that having experiences in active participation can accelerate the integration time. McIntosh et al. [54] hypothesized that a large size of code change could be integrated, but due to either (1) being omitted from review activity or (2) the lack of review involvement. Indeed, McIntosh et al. [55] found that low review coverage, participation, and expertise can cause the permeation of defect-prone code.

Thongtanunam et al. [86] showed that modules with a larger proportion of developers without code authorship or reviewing expertise are more likely to be defect-prone. Kononenko et al. [47] reported that Mozilla developers believe that

reviewer experience is associated with review quality. Reviewer involvement has also been linked with post-release defect proneness [85], as well as the incidence of software design anti-patterns [58], and security vulnerabilities [56]. For instance, Kemerer and Paulk [45] showed that adding design and code reviews to student projects at the SEI led to software artifacts that were of higher quality. Bosu et al. [15] found that the reviewer expertise is an important factor in determining whether Microsoft developers consider code review feedback useful towards a good quality of source code. Kononenko et al. [48] found that review participation metrics (e.g., the number of participants in a review) are associated with the quality of the code review process. Shimagaki et al. [79] reported that code review participation is significantly associated with software quality.

Baysal et al. [9] found that more experienced patch authors receive faster responses. Gousios et al. [27] showed that less experienced developers often exhibit the lack of git skills, leading to unnecessary communication effort. Porter et al. [62] showed that selecting appropriate reviewers improves the defect prediction effectiveness. Prior work has proposed reviewer recommenders based on the past history of review processes. For example, Balachandran [6] proposed ReviewBot, which recommends reviewers based on past contributions to the lines that were modified by the patch. Thongtanunam et al. [88] proposed RevFinder, which recommends reviewers based on past contributions to the modules that have been modified by the patch. More recent work has improved reviewer recommendations by leveraging past review contributions [100], technological experience and experience with other related projects [65, 66], and the content of the patch itself [95]. Kovalenko et al. [49] question the value of recommending reviewers in cases where the best reviewers are already known.

Not all changes that are submitted for code review end up being integrated. Indeed, Weißgerber et al. [92] found that 39% and 42% of OpenAFS and FLAC code changes were eventually integrated. Jiang et al. [40] found that 33% of Linux code reviews were eventually integrated. Baysal et al. [8] found that 36%–39% of code changes in Mozilla Firefox were rejected and resubmitted at least once. To help developers to understand (and improve) the chances of their submissions being integrated, researchers have studied the characteristics of code changes (and their reviews) that were eventually integrated. For example, Weißgerber et

al. [92] found that small code changes are integrated more frequently than large ones. Jiang et al. [40] showed that prior experience, patch maturity, and code churn significantly impact the likelihood of integration. Baysal et al. [9,10] found that non-technical issues are a common reason for abandonment in WebKit and Blink projects. Moreover, Tao et al. [83] found that patch design issues like suboptimal solutions and incomplete fixes are often raised during the reviews of the abandoned code changes of the Eclipse and Mozilla projects.

Review links may affect or imply review outcomes. For example, since reviews that supersede prior reviews have had the opportunity to improve the design of the code change, they may have a higher likelihood of being integrated than initial submissions. Moreover, large tasks that have been divided into a series of reviews (e.g., Reviews #102532[2] and #102543[3] of OpenStack) will have review outcomes that are inherently linked.

Code review analytics have traditionally treated reviews as individual; In reality, review artifact may have more complex relationship with other reviews due to a variety of reasons (e.g., external dependency, superseding). For example, Review #109178 is another attempt at tackling the underlying task of (the abandoned) Review #105238.[4] To preserve continuity of the review discussion, the reviewers of Review #105238 should also be recommended for Review #109178. Since links may indicate a (strong) relationship between linked reviews, reviewers who are recommended for one of the linked reviews may also need to be recommended for the others. Based on the context above, review processes should be analyzed at a fine-grained level.

Traditional code review analytics have typically treated review artifacts as individual; yet, reviews may be interdependent. A fine-grained study for the impact of review linkage on code review resolutions is needed to demonstrate that linkage could improve the performance of code review analytics.

[2] `https://review.openstack.org/#/c/102532/`

[3] `https://review.openstack.org/#/c/102543/`

[4] `https://review.openstack.org/#/c/105238/`

2.4 Summary

Despite the growth of code review tools, there have been a lot of room for improvement in code review resolution processes. Based on related work, we summarize that a fine-grained research that deepens both quantitative and qualitative analyses at a comment-level is needed. Our related studies have mostly analyzed code review resolutions at a process-level (i.e., exploring the overall outcomes of code review artifacts that are impacted by divergence and openness). However, it is unclear that how the impacts of divergence and openness occurred and were resolved as developers communicate with other developers. Having practical implications at a comment-level to improve code review resolutions would provide developers with dynamic resolution approaches in which developers can resolve problems that are brought by divergent discussion and open participation when commenting. Hence, this thesis is motivated to discover the resolution mechanism for the impacts of divergence and openness at a comment-level.

3. The Mechanism of Divergent Discussion at a Fine-Grained Level

3.1 Introduction

Code review is widely considered a best practice for software quality assurance [94]. The Modern Code Review (MCR) process—a lightweight variant of the traditional code inspection process [23]—allows developers to post patches, i.e., sets of changes to the software system, for review. Reviewers (i.e., other team members) are either: (1) appointed automatically based on their expertise [66, 88, 100]; (2) invited by the author [36, 54, 88]; or (3) self-selected by broadcasting a review request to a mailing list [30, 73, 74].

Reviewer opinions about a patch may differ. Therefore, patches that are critiqued by more than one reviewer may receive scores both in favour of and in opposition to integration. In this paper, we focus on these patches as their discussions have the potential to be divergent. In order to arrive at a final decision about these patches, the divergent positions of reviewers will likely need to be resolved.

Broadly speaking, divergent opinions have been studied for a long time in academic settings [38]. Recently, divergence in reviews has been studied within open source communities [91], and has been observed to correlate with negative development outcomes [91]. For example, divergence has been associated with increased rates of developer abandonment [37] and poorer team performance [24].

Specifically, in the context of code review, divergent review scores may forebode disagreement among reviewers. Divergent reviews can slow integration processes down [36, 74] and can create a tense environment for contributors [76]. For instance, consider review #12807 from the QtBase project.[5] The first reviewer approves the patch for integration (+2). Afterwards, another reviewer blocks the patch from being integrated with a strong disapproval (-2), arguing that the scope of the patch must be expanded before integration could be permitted. Those reviewers who provided divergent scores discussed whether the scope of the patch was sufficient for five days, but an agreement was never reached. One month

[5]https://codereview.qt-project.org/#/c/12807/

later, the patch author abandoned the patch without participating in the discussion. Despite making several prior contributions, this is the last patch that the author submitted to the QtBase project.

In this paper, we set out to better understand patches with divergent review scores and the process by which an integration decision is made. To achieve our goal, we analyze the large and thriving OpenStack and Qt communities. Through quantitative analysis of 49,694 reviews, we address the following two research questions:

(RQ1) How often do patches receive divergent scores?

Motivation: Review discussions may diverge among reviewers. We first set out to investigate how often patches with divergent review scores occur.

Results: Divergent review scores are not rare. Indeed, 15%–37% of the studied patch revisions that receive review scores of opposing polarity.

(RQ2) How often are patches with divergent scores eventually integrated?

Motivation: Given that patches with divergent scores receive both positive and negative scores, making an integration decision is not straightforward. Indeed, integration decisions do not always follow a simple majority rule [35]. We want to know how often these patches are eventually integrated.

Results: Patches are integrated more often than they are abandoned. For example, patches that elicit positive and negative scores of equal strength are eventually integrated on average 71% of the time. The order in which review scores appear correlates with the integration rate, which tends to increase if negative scores precede positive ones.

(RQ3) How are reviewers involved in patches with divergent scores?

Motivation: Patches may require scores from additional reviewers to arrive at a final decision, imposing an overhead on development. We set out to study in reviews with divergent scores (a) if additional reviewers are involved; (b) when reviewers join the reviews; and (c) when divergence tends to occur.

Results: Patches that are eventually integrated involve one or two more re-

viewers than patches without divergent scores on average. Moreover, positive scores appear before negative scores in 70% of patches with divergent scores. Reviewers may feel pressured to critique such patches before integration (e.g., due to lazy consensus).[6] Finally, divergence tends to arise early, with 75% of them occurring by the third (Qt) or fourth (OpenStack) revision.

To better understand divergent review discussions, we qualitatively analyze: (a) all 131 of the patches that elicit strongly divergent scores from members of the core development teams; (b) a random sample of 329 patches that elicit weakly divergent scores from contributors; and (c) a random sample of 131 patches without divergent scores. In doing so, we address the following research questions:

(RQ4) What drives patches with divergent scores to be abandoned?
Motivation: In RQ2, we observe that 29% of the studied patches with divergent scores are eventually abandoned. Since each patch requires effort to produce, we want to understand how the decision to abandon patches with divergent scores is reached.
Results: Abandoned patches with strong divergent scores more often suffer from external issues than patches with weakly divergent scores and without divergent scores do. These external issues most often relate to release planning and the concurrent development of solutions to the same problem.

Our results suggest that: (a) software organizations should be aware of the potential for divergence, since patches with divergent scores are not rare and tend to involve additional personnel; and (b) automation could relieve the burden of reviewing for external concerns, such as release scheduling and duplicate solutions to similar problems.

Enabling replication. To enable future studies, we have made a replication package available online,[7] which includes the collected data and analysis scripts,

[6] `https://community.apache.org/committers/lazyConsensus.html`
[7] `https://figshare.com/s/4d3e096f4339f6d4b3da`

as well as more detailed hierarchies of the reasons for abandonment and integration of patches with divergent scores.

Paper organization. The remainder of this section is organized as follows. Section 3.2 describes the design of our case study, while Sections 3.3 and 3.4 present the quantitative and qualitative results, respectively. Section 3.5 discusses the broader implications of our results. Section 3.6 discloses threats to the validity of our study. Finally, Section 3.7 draws conclusions.

3.2 Case Study Design

We describe the subject systems, their Gerrit-based review processes, and our data preparation approach.

3.2.1 Studied Projects

To address our research questions, similar to prior work [55, 86], we perform an empirical study on large, thriving, and rapidly evolving open source systems with globally distributed teams. Due to the manually intensive nature of our qualitative analysis, we choose to select a sample of four software projects from two communities. We select the OpenStack and Qt communities for analysis because they have made a serious investment in code review for several years, having adopted the Gerrit tool for managing the code review process.[8] These communities are theoretically sampled [97] to represent different sub-populations. OpenStack is a free and open source software platform for cloud computing that is primarily implemented in Python and is developed by many well-known software companies, e.g., IBM, VMware, and NEC [88]. Qt is a cross-platform application and UI framework that is primarily implemented in C++ and is developed by the Digia corporation, but welcomes contributions from the community at large [87].

The OpenStack and Qt communities are composed of several projects. We study the two most active projects in each community (based on the number on patch submissions). Table 1, which provides an overview of the studied projects, shows that 18% (Nova), 20% (Neutron), 67% (QtBase), and 84% (Qt-creator) of patches involve only one reviewer. The discrepancy between

[8] `https://code.google.com/p/gerrit/`

Table 1: An overview of the subject systems. The white and gray rows show the overview of each subject system and each community, respectively.

Product	Scope	Studied Period	#Patches	#Multi-Rev Patches	#Revs
OpenStack		07/2011 to 01/2018	444,582	297,961	9,108
Nova	Provisioning management	09/2011 to 01/2018	25,901	21,224	1,602
Neutron	Networking abstraction	07/2013 to 01/2018	12,350	9,936	1,063
Qt		05/2011 to 01/2018	168,066	36,752	1,785
QtBase	Core UI functionality	05/2011 to 01/2018	37,974	12,459	627
Qt-creator	Qt IDE	05/2011 to 01/2018	37,587	6,075	278

the communities (OpenStack vs. Qt) is likely reflective of differences in integration policies—while OpenStack changes normally require two +2 scores for integration approval, only one +2 score is required for Qt changes.

3.2.2 Data Preparation

Using the Gerrit API, we extracted review records for all of the available OpenStack and Qt patches in January 2018. Then, as described below, we prepared the data for analysis.

Divergent score identification. We begin by identifying patches with divergent scores, i.e., patches elicit at least one positive and one negative score. Since these patches receive opposing scores, they indicate the potential for divergence. However, patches with divergent scores may include false positive cases where there is no actual divergence between positive and negative scores. In this study, we further identify patches that have actual divergent scores due to a variety of reviewer concerns.

First, to identify whether or not a patch has divergent review scores, we select the patches that have at least one positive and one negative review score in the same revision. In cases where a reviewer updates their score, we select their first score because we want to know their initial opinion before changing their mind. We then use four patterns of divergence based on the Gerrit review scores (+2,

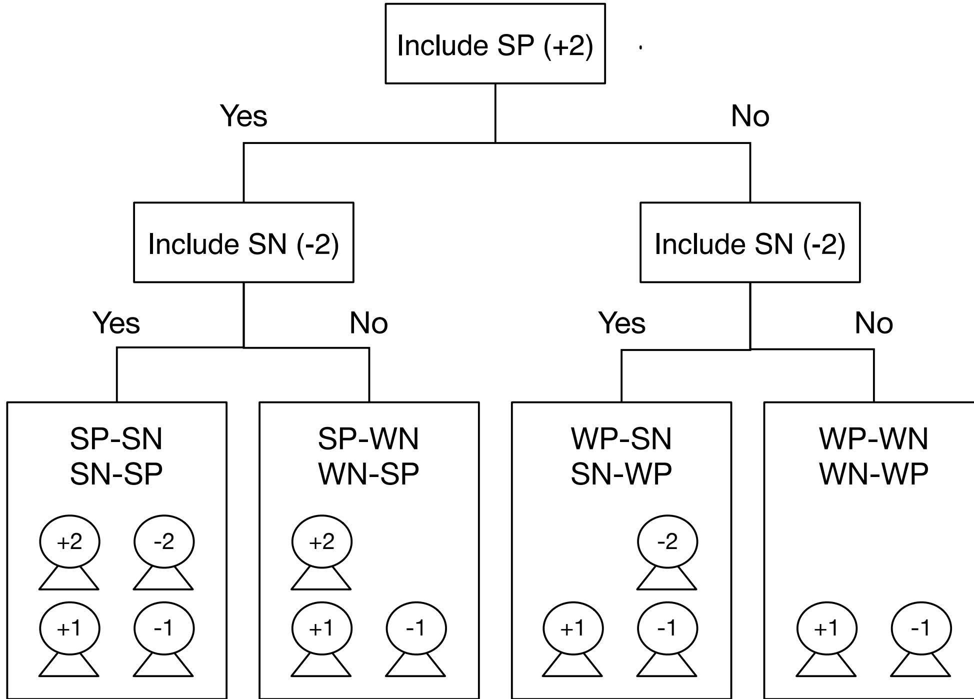

Figure 3: An overview of our approach to classify patches with divergent scores.

+1, -1, -2). We do not include 0 scores because they indicate no position. In addition, we consider the order in which the scores were submitted, and classify patches with divergent votes into eight patterns:

- *Strong Positive, Strong Negative (SP-SN, SN-SP).* Patches with at least one +2 and at least one -2. SP-SN if +2 appears before -2, SN-SP otherwise.

- *Strong Positive, Weak Negative (SP-WN, WN-SP).* Patches with at least one +2 and at least one -1, but no -2. SP-WN if +2 appears before -1, WN-SP otherwise.

- *Weak Positive, Strong Negative (WP-SN, SN-WP).* Patches with at least one -2 and at least one +1, but no +2. WP-SN if +1 appears before -2, SN-WP otherwise.

- *Weak Positive, Weak Negative (WP-WN, WN-WP).* Patches with at least one -1 and at least one +1, but no +2 or -2. WP-WN if +1 appears before -1, WN-WP otherwise.

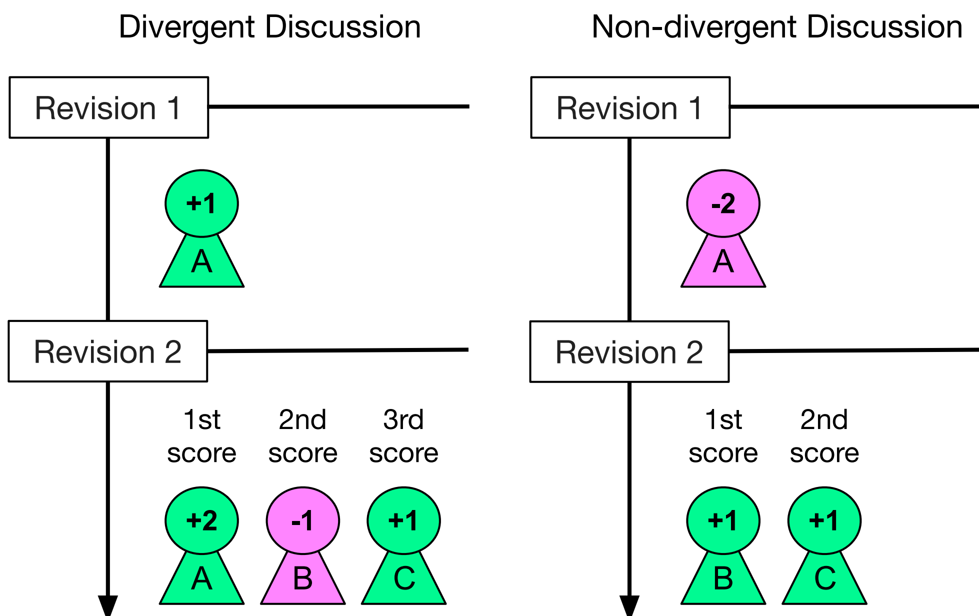

Figure 4: Examples of review score patterns that we consider divergent and non-divergent.

Figure 3 provides an overview of our classification process. For example, when classifying patch with `+2`, `-1`, and `+1` scores on the same revision, we follow the path for patches that "Include SP" on the left, and then the path for patches that do not "Include SN" on the right to arrive at the "strong positive and weak negative" category. Since the strong positive is provided earlier than weak negative, the patch is classified as SP-WN.

Note that the patch revision process complicates our classification approach, since review scores may be recorded with respect to different patch revisions. We focus on review scores that were applied to the same revision when identifying patches with divergent scores. For example, the scenario on the left of Figure 4 shows a patch with divergent scores, since it has at least one positive score (Rev A or Rev C) and one negative score (Rev B) on the same revision (Revision 2). Although the scenario on the right of Figure 4 also receives positive and negative scores, they do not appear in the same revision, and thus, it is not labelled as divergent.

Table 2: The number and proportion of each potentially divergent pattern in the four subject systems.

System	SP-SN	SN-SP	SP-WN	WN-SP	WP-SN	SN-WP	WP-WN	WN-WP	Total
Nova	192 (1%)	52 (< 1%)	2051 (10%)	785 (4%)	299 (1%)	30 (< 1%)	2945 (14%)	349 (2%)	6703 (32%)
Neutron	112 (1%)	24 (< 1%)	918 (9%)	414 (4%)	124 (1%)	21 (< 1%)	1763 (18%)	320 (3%)	3696 (37%)
QtBase	50 (< 1%)	11 (< 1%)	499 (4%)	302 (2%)	86 (1%)	10 (< 1%)	1163 (9%)	208 (2%)	2329 (19%)
Qt-creator	21 (< 1%)	4 (< 1%)	180 (3%)	81 (1%)	32 (1%)	9 (< 1%)	497 (8%)	81 (1%)	905 (15%)

3.3 Quantitative Results

We present the results of our quantitative analysis with respect to RQ1–RQ3. For each question, we describe our approach for addressing it followed by the results that we observe.

(RQ1) How often do patches receive divergent scores?

Approach. To address RQ1, we examine how often the reviews of patches in Nova, Neutron, QtBase, and Qt-creator receive divergent scores. To do so, we compute the rate of multi-reviewer patches that receive positive and negative scores on the same revision (Table 2).

Results. Observation 1—Patches with divergent scores are not rare in OpenStack and Qt. Table 2 shows that 15%–37% of the patches that have multiple review scores receive both positive and negative scores. While the majority of patches with divergent scores have only a weak negative score (SP-WN, WN-SP, WP-WN, and WN-WP), 7%–9% have at least one strong negative scores (SP-SN, SN-SP, WP-SN, and SN-WP) in the four subject systems (e.g., $\frac{192+52+299+30}{6{,}703} \simeq 9\%$ in Nova).

From the alternative perspective, 63%–85% of reviews do not have divergent scores. On the surface, this may seem like reviewers often agree about patches, calling into question the necessity of having more than one reviewer evaluate patches. We caution against this interpretation because more reviewers bring additional perspectives, commenting on a variety of concerns during the review process. Moreover, reviewer scores may change across revisions—a case that we conservatively classify as non-divergent.

Table 3: The integration rate of each divergent score pattern in the four subject systems.

System	SP-SN	SN-SP	SP-WN	WN-SP	WP-SN	SN-WP	WP-WN	WN-WP
Nova	46%	94%	83%	93%	34%	50%	69%	66%
Neutron	58%	92%	88%	96%	19%	33%	71%	65%
QtBase	56%	82%	81%	90%	28%	50%	73%	70%
Qt-creator	43%	75%	81%	96%	59%	56%	81%	90%

15%–37% of patches that receive multiple review scores have divergent scores.

(RQ2) How often are patches with divergent scores eventually integrated?

Approach. To address RQ2, we examine the integration rates of patches with divergent scores. For each divergence pattern, we compute the proportion of those patches that are eventually integrated (Table 3).

Results. Observation 2—Patches that receive stronger positive scores than negative ones tend to be integrated. In patches that receive stronger positive review scores than negative ones, intuition suggests that most will be integrated. Table 3 shows that this is indeed the case, with the SP-WN integration rate of 81%–88% and WN-SP integration rate of 90%–96% across the four subject systems. Interestingly, even patches with weakly scored criticism are not integrated on occasion, suggesting that authors are attuned to that criticism despite securing a positive score that enables integration.

Similarly, in patches that receive stronger negative scores than positive scores, intuition suggests that most will be abandoned. Again, Table 3 shows that in the Nova, Neutron, and QtBase systems, this is indeed the case in the WP-SN pattern, with the integration rates below 50%. The WP-SN integration rate is only above 50% for Qt-creator (59%); however, we note a greater tendency towards acceptance in all of the patterns for the Qt-creator system.

The results imply that a -2 score is a large obstacle to integration. Resolving criticism is a common research topic not only in code review [37], but also academic paper reviewing [38]. The prior work on code reviews suggests

that when developers include constructive suggestions (e.g., alternative solutions) when critiquing patches, authors feel encouraged to continue contributing. The recommendation indicates that such a critique should be resolved through patient contributions from both reviewers and authors.

Unlike in academic peer review settings, the order in which the code review scores are recorded shares a strong relationship with the integration rate. Table 3 shows that if the negative score is submitted before the positive score (i.e., SN-WP), the integration rate grows by 14–22 percentage points in the NOVA, NEUTRON, and QTBASE systems. On the other hand, we observe a drop of three percentage points in the QT-CREATOR system; however, the integration rate remains above 50%.

Observation 3—Patches that receive positive and negative scores of equal strength tend to be integrated. In patches that receive positive and negative scores of equal strength, intuition suggests that half of such patches will be integrated. Indeed, Table 3 shows that the SP-SN pattern has integration rates of 43%–58%, which is roughly in line with our intuition. Again, the order in which scores are recorded plays a key role—when a strong positive score is provided after a strong negative one (SN-SP), the integration rates increase to 75%–94%, suggesting that the late arrival of a proponent can encourage contributors to revise and rescue the patch from abandonment.

Likewise, the WP-WN and WN-WP patterns have the integration rates of 69%–81% and 65%–90% respectively, which is higher than we had anticipated. The results suggest that patches with weak divergent scores are often redeemed, even if there is only weak support for them initially.

Patches with divergent scores are often eventually integrated. The timing of the arrival of scores plays a role, with later positive scores being correlated with considerable increases in integration rates.

(RQ3) How are reviewers involved in patches with divergent scores?

Approach. To address RQ3, we examine the review data from three perspectives. First, we report the difference in the number of reviewers in patches with divergent review scores and those without (Table 4). To analyze reviewer ten-

Table 4: The number of reviewers in reviews that receive divergent scores and reviews that do not in the four subject systems.

System	SP-SN	SN-SP	SP-WN	WN-SP	WP-SN	SN-WP	WP-WN	WN-WP	Non
Nova	4.28	4.96	3.71	4.44	3.89	3.40	3.41	3.32	3.65
	(0.63)	(1.31)	(0.06)	(0.79)	(0.24)	(-0.25)	(-0.24)	(-0.33)	–
Neutron	5.34	5.00	4.69	5.19	3.74	4.71	4.01	4.07	4.07
	(1.27)	(0.93)	(0.63)	(1.12)	(-0.33)	(0.65)	(-0.05)	(< 0.01)	–
QtBase	2.57	2.56	2.49	2.64	2.46	2.20	2.46	2.39	2.30
	(0.27)	(0.26)	(0.19)	(0.34)	(0.16)	(-0.10)	(0.17)	(0.10)	–
Qt-creator	2.44	2.00	2.24	2.42	2.21	2.00	2.26	2.22	2.17
	(0.27)	(-0.17)	(0.07)	(0.25)	(0.04)	(-0.17)	(0.09)	(0.05)	–

Table 5: The N→SP and P→SN rates of the five most active reviewers in the four subject systems.

Rank	Nova		Neutron		QtBase		Qt-creator	
	N→SP	P→SN	N→SP	P→SN	N→SP	P→SN	N→SP	P→SN
1st	17%	50%	31%	0%	50%	75%	18%	90%
2nd	58%	90%	43%	88%	36%	100%	46%	93%
3rd	49%	97%	53%	100%	35%	75%	22%	0%
4th	36%	88%	52%	90%	46%	91%	28%	0%
5rd	29%	100%	44%	84%	30%	0%	31%	83%

dencies, we then compute the rate at which the five most active reviewers in each subject system submit scores of a different polarity than the other reviewers (Table 5). Finally, we plot the revision when divergence occurs against the last revision of each review with divergent scores (Figure 5).

Results. Observation 4—Patches with divergent scores tend to require one to two more reviewers than patches without divergent scores. Table 4 shows that overall, patches that receive strong positive scores (SP-SN, SN-SP, SP-WN, and WN-SP) have a greater difference in the number of reviewers than the other patterns. Table 4 shows that specifically in SP-SN and SN-SP patterns of Nova and Neutron, the differences range from 0.63 to 1.31. Unlike these patterns, the difference in patches without `-2` scores (WP-SN, SN-WP, WP-WN, and WN-WP) tends to remain below one. Rigby and Bird [72] observed that two reviewers were often enough, except in cases where reviews became a group problem solving activity. Our results complement theirs, suggesting that patches with divergent scores also require additional review(er) resources to reach a final decision.

Observation 5—The most active reviewers tend to provide negative scores after positive ones. Table 5 shows that the most active reviewers provide strong negative scores after positive ones (P→SN) more often than strong positive scores after negative ones (N→SP). Indeed, 16 of the 20 analyzed reviewers stop a positive trend in review scores more often than they stop a negative trend, with personal differences of 25–72 percentage points. Active reviewers may feel pressured to critique patches with a positive trend before integration. The practice of lazy consensus, where a reviewer avoids joining a discussion when other reviewers raise concerns, may also help to explain this tendency.

In contrast, there are four reviewers who tend to stop negative trends (1st in Neutron, 5th in QtBase, and 3rd and 4th in Qt-creator). All four of these reviewers never stop a positive trend with a strong negative score. Upon closer inspection, we find that these reviewers are senior community members who take on a managerial role. These reviewers appear to only join reviews late in order to provide a strong vote in favour for high priority content.

Observation 6—Divergent scores have a tendency to arise before the final revision. Figure 5 shows that the majority of divergent review scores occur well before the final revision where the integration decision is made. Moreover, Figure 6 shows that 75% of patches with divergent scores occur by the third and fourth revisions in Qt and OpenStack, respectively. Intuitively, this may be occurring because early revisions are rougher drafts of a patch.

Patches with divergent scores tend to require one to two more reviewers than ones without divergent scores. Moreover, the most active reviewers tend to submit negative scores after positive ones more often than vice versa. Finally, divergent scores tend to arise in the early revisions of a patch.

3.4 Qualitative Results

Our quantitative results in Section 3.3 (RQ1–RQ3) indicate that patches with divergent scores are not rare (Observation 1), and have a tendency to be integrated (Observations 2–4). Furthermore, on average, patches with divergent scores tend to involve one or two more reviewers than patches without divergent scores, and receive negative scores after positive ones in 70% of cases (Observations 5, 6).

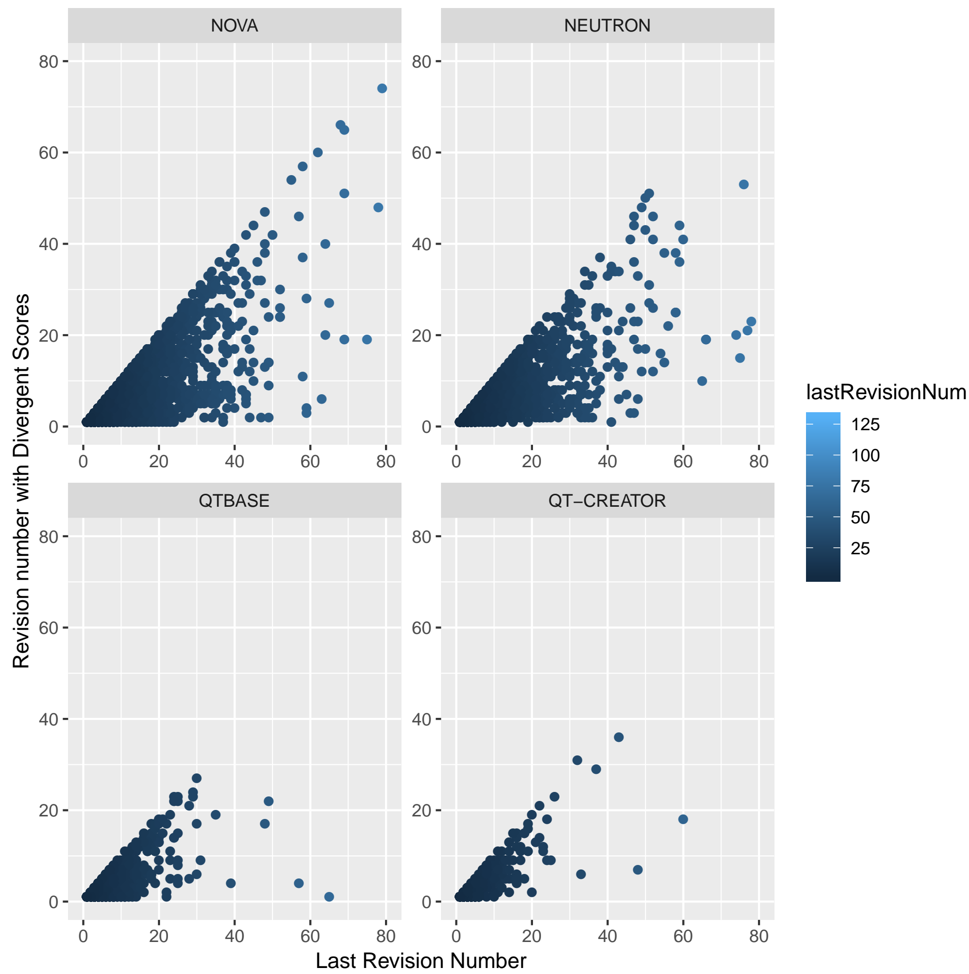

Figure 5: The revision number that divergent scores appear on and revision number that the final decision is made in.

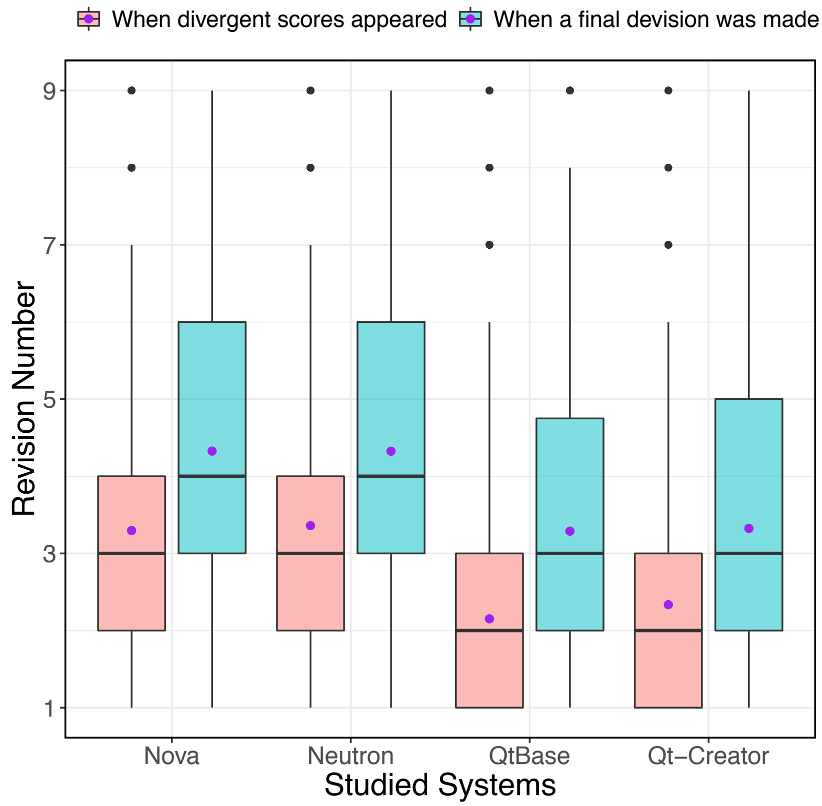

Figure 6: The distributions of revision number that divergent scores appear on and revision number that the final decision is made in.

We set out to better understand what concerns are raised in reviews with divergent scores through qualitative analysis. More specifically, we present the results of our qualitative analysis with respect to RQ4. Similar to Section 3.3, we describe our approach for addressing it followed by the results.

To perform our qualitative analysis, we first analyze patches that have divergent scores in SP-SN and SN-SP patterns of Nova and QtBase systems. We select SP-SN and SN-SP patterns for qualitative analysis because their patches elicit strong and divergent opinions from core members of the reviewing team. Moreover, to understand the difference between patches that have divergent scores and patches that do not, we analyze a sample of patches without divergent scores that is of equal size to the number of SP-SN and SN-SP patches. The sample of those patches is selected randomly, and covers the overall studied period (i.e., September 2011-January 2018). Moreover, half of the reviewers who have participated in the SP-SN and SN-SP patches are also observed in the sample of patches without divergent scores. Finally, we analyze WP-WN and WN-WP patterns to understand disagreements that do not have integration implications (i.e., `+1` and `-1`). Since manual analysis of all WP-WN and WN-WP patterns is impractical, similar to prior work [74], we aim to achieve saturation. Like prior work [99], we continue to label randomly selected WP-WN and WN-WP patches until no new labels are discovered for 50 consecutive patches.

(RQ4) What drives patches with divergent scores to be abandoned?

Approach. In addressing RQ4, we want to know why authors and reviewers eventually agree to abandon patches that have divergent scores. To do so, we code the review discussions of sampled patches with and without divergent scores that are eventually abandoned. Our codes focus on the key concerns that reviewers with negative scores raise. Each review may be labelled with multiple codes.

In this qualitative analysis, similar to prior studies [4, 30, 46, 81], we apply open card sorting to construct a taxonomy from our coded data. This taxonomy helps us to extrapolate general themes from our detailed codes. The card sorting process is comprised of four steps. First, we perform an initial round of coding. The first three authors were involved in the coding process such that each review was coded by at least two authors. Second, if the authors' codes disagree, the

authors discussed the review in question until a consensus was reached. Third, the coded patches are merged into cohesive groups, which can be summarized with a short title. Finally, the related groups are linked together to form a taxonomy of reasons for abandonment.

Results. Table 6 shows an overview of the categories of key concerns that were raised in the analyzed reviews of abandoned patches with and without divergent scores. A more detailed figure is available online.[7] In Table 6, the white cells show the number of occurrences of each label, while the gray cells show the number of patches with and without divergent scores that belong to the internal or external concerns categories.

Observation 7—Patches with weakly divergent scores have a higher rate of false positives than patches with strongly divergent scores. We find that patches with weakly divergent scores are less likely to actually have a divergence in Nova and QtBase. More specifically, we detect 142 and 73 false positive patches among the 192 and 137 potentially WP-WN and WN-WP patches in the Nova and QtBase systems, respectively. In most WP-WN false positives, reviewers with `-1` scores raise concerns that reviewers with `+1` scores have not found, which makes reviewers wait for the next revision due to lazy consensus. In contrast, we detect only four and one false positive patches among the 107 and 24 potentially SP-SN and SN-SP patches in the Nova and QtBase systems, respectively.

We designate another six patches as works in progress, i.e., patches for which the author intends to collect early feedback. These patches are not intended for integration into the codebase. For example, in review #336921,[9] the commit message states that the patch is "Internal review purpose only." The remaining patches include divergence among reviewers through divergent scores and contain internal concerns about the content of the patch and external concerns about factors that are beyond the scope of the patch content.

Observation 8—External concerns are often raised in abandoned patches that have strong divergent scores. Surprisingly, Table 6 shows that external concerns in abandoned SP-SN and SN-SP patches appear more often than internal concerns in both of Nova and QtBase. Moreover, when compared

[9] `https://review.openstack.org/#/c/336921/`

Table 6: The reasons for abandoned patches with and without divergence.

Label	Nova					QtBase				
	SP-SN	SN-SP	WP-WN	WN-WP	Non	SP-SN	SN-SP	WP-WN	WN-WP	Non
External Concern	56 (54%)		17 (32%)		45 (41%)	14 (61%)		17 (27%)		13 (54%)
Unnecessary Fix	25	-	6	7	27	12	2	2	9	7
Integration Planning	28	2	3	3	12	1	-	2	4	1
Policy Compliance	3	-	-	-	3	-	-	1	-	3
Lack of Interest	-	-	-	-	5	-	-	-	-	3
Internal Concern	48 (46%)		36 (68%)		61 (56%)	9 (39%)		47 (73%)		8 (33%)
Design	44	1	10	19	45	6	-	18	17	6
Implementation	2	-	1	2	2	2	-	7	8	-
Testing	2	-	4	1	15	1	-	1	2	2
Works in Progress	-		-		3 (3%)	-		-		3 (13%)
Early Feedback	-	-	-	-	3	-	-	-	-	3

Table 7: The rate of outsider-dominated discussions.

In patches whose scores are	Nova	Neutron	QtBase	Qt-creator
Strongly divergent	25%	28%	$< 1\%$	8%
Weakly divergent	28%	29%	2%	2%
not divergent	11%	14%	1%	$< 1\%$

with abandoned patches with weakly divergent scores and patches without divergent scores, the rates of external concerns increase by 13–22 percentage points in Nova and 7–34 percentage points in QtBase. Similar to prior work [74], we find that reviews within our studied context are mostly topical and relevant; however, they occasionally digress into "bike shed" discussions, i.e., strongly opinionated discussions about (irrelevant) details. For example, in review #40317,[10] a reviewer explicitly mentioned that the divergent discussion is "bike-shedding." To better understand how often review discussions in our context show signs of "bike shedding," we replicate the detection approach of Rigby and Storey [74], which focuses on review discussions that are dominated by outsiders (non-core members). Table 7 shows the rate of outsider-dominated discussions in patches with and without divergent scores. The results suggest that patches with divergent scores may be more susceptible to "bike shed" discussions in Nova, but less so in QtBase. Indeed, the OpenStack community is larger (see Table 1) and involves more organizations than Qt, which may explain why more reviews are dominated by outsiders. A closer look at the relationship between "bike shed-

[10] https://codereview.qt-project.org/#/c/40317/

ding" and divergence would be an interesting topic for future work.

External concerns are further decomposed into "Unnecessary Fix," "Integration Planning," "Policy Compliance," and "Lack of Interest" categories, which we describe below.

Unnecessary Fix accounts for the majority of external concerns. This category consists of patches whose changes are not necessary or have already been addressed in another patch. For example, in review #16715,[11] a reviewer pointed out that the issue has been already fixed by review #15726.[12]

Integration Planning occurs often in the Nova project. The category consists of patches where the integration depends on the release schedule or the integration of another patch. For example, the author of review #262938[13] was told to "resubmit this for Newton" (a future release at the time).

Policy Compliance consists of patches that violate project change management policies. For example, in review #33157,[14] reviewers argued that the patch has an incorrect change ID, which would hinder its traceability in the future.

Lack of Interest consists of patches that the author or reviewers did not pursue until integration. For example, in review #76783,[15] (a patch without divergent scores) the author did not address the reviewer feedback. The patch was automatically abandoned after one week of inactivity.

Observation 9—Internal concerns are raised at greater or equal rates in patches with strong divergent scores and those without divergent scores. Table 6 shows that in Nova, internal concerns are raised in patches without divergent scores ten percentage points more often than in SP-SN and SN-SP patches. In QtBase, internal concerns are raised in roughly equal proportions in SP-SN and SN-SP patches and patches without divergent scores. Internal concerns include "Design," "Implementation," and "Testing."

Design accounts for the majority of internal concerns. This concern consists of patches that suffer from patch design issues. For example, in review #87595,[16]

[11] https://codereview.qt-project.org/#/c/16715/
[12] https://codereview.qt-project.org/#/c/15726/
[13] https://review.openstack.org/#/c/262938/
[14] https://review.openstack.org/#/c/33157/
[15] https://review.openstack.org/#/c/76783/
[16] https://codereview.qt-project.org/#/c/87595/

a reviewer suggests an alternative solution for the underlying issue.

Implementation concerns consist of patches that break backward compatibility, introduce readability issues, and/or introduce bugs. For example, in review #83667,[17] reviewers pointed out that the removal of an API was not possible, since it would break backwards compatibility.

Testing concerns consist of patches that suffer from test design and coverage issues. In situations where a change required the addition of use cases like review #21250,[18] reviewers argue that the patch should not be integrated until the requisite tests are added.

Abandoned patches that elicit strong divergent scores tend to suffer more from external concerns than patches with weak divergent and without divergent scores.

3.5 Practical Suggestions

We discuss broader implications of our observations for organizations, tool developers, and authors.

3.5.1 Software Organizations

Software organizations should be aware of the potential for divergent review scores. This recommendation is especially pertinent for organizations that, like OpenStack, set integration criteria to require more than one approval (+2 score), since this creates more opportunities for divergent scores to arise (cf. Qt in Table 1). Patches with divergent scores account for 15%–37% of multi-reviewer patches in the four subject systems (Observation 1).

Patches with divergent scores tend to require additional personnel. Patches with positive and negative scores of equal magnitude have a tendency to be integrated (Observations 2 and 3). However, those patches that are eventually integrated tend to involve one to two more reviewers than patches without divergent scores (Observation 4).

[17] https://review.openstack.org/#/c/83667/

[18] https://codereview.qt-project.org/#/c/21250/

3.5.2 Tool Developers

Automatic detection of similar changes would reduce waste in the reviewing process. "Unnecessary Fix" (e.g., Already Fixed) accounts for the majority of external concerns in patches with strong divergent scores of Nova and QtBase that are eventually abandoned (Observation 8). Indeed, Sedano et al. [77] also found that duplicated work is a frequently occurring type of software development waste. This duplicated effort is not only a waste of author time, but given that this is a common pattern in patches with divergent scores, reviewers are also wasting time to suggest improvements for patches that will not be integrated. Ideally, patch authors should check for existing solutions or ones being actively developed before developing a solution of their own. However, as Zhou et al. [101] point out, in large, decentralized projects (like OpenStack and Qt), it is difficult for developers to keep track of concurrent development activity. Tool support for detection of similar changes in the code reviewing interface would likely save this reviewing effort. Future research may make improvements to team synchronization tools that notify team members of issues that have already been addressed by others.

3.5.3 Patch Authors

A perfect technical solution may still receive divergent scores. For a patch to be integrated, concerns that are related to the broader context of the entire project need to be satisfied. For example, the majority of abandoned patches with strong divergent scores in Nova suffer from non-technical issues (Observation 8). The results suggest that raising the author awareness of external concerns would help to avoid such divergence. Automating the detection of similar changes and release schedule (see suggestions for tool developers above) would likely help in this regard.

3.6 Threats to Validity

We now discuss the threats to the validity of our analyses.

3.6.1 Construct Validity

Construct threats to validity are concerned with the degree to which our analyses are measuring what we aim to study. Our qualitative analyses are focused on review discussions that are recorded in the Gerrit code review system. However, there are likely cases where reviewers discuss submitted patches in-person [11], on IRC [78], or in mailing list discussions [30,74]. Unfortunately, there are no explicit links that connect these communication threads to patches, and recovering these links is a non-trivial research problem [5, 12, 41]. On the other hand, virtually every patch in the studied OpenStack and Qt systems could be linked to review discussions on Gerrit.

We focus our detection on patches with opposing reviewer scores that appear in the same revision. However, there may be cases where the discussion on an early revision is related with divergence that happens in a later revision. As such, our results should be interpreted as a lower bound on the rate of patches with divergent scores. Moreover, some reviewing systems do not provide a mechanism for reviewers to vote in favour or against integration of a patch. Note that this does not mean that divergence is not possible in such systems (i.e., reviewers may still disagree with each other). However, to analyze patches with divergent scores in such a setting, our detection model would need to be updated (e.g., using NLP or sentiment analysis techniques).

3.6.2 Internal Validity

Internal threats to validity are concerned with our ability to draw conclusions from the relationship between study variables. We analyze patches with divergent scores that solicit at least one positive and one negative review score from different reviewers. However, there might be cases where reviewers disagree with other reviewers in the discussion without submitting opposing review scores. Natural language processing techniques (e.g., sentiment analysis) could be used to approximate the opinions of the reviewers. However, such techniques are based on heuristics and may introduce noise. Nonetheless, we plan to explore the applicability of sentiment analysis in future work.

The open card sorting approach that we adopt is subject to opinions of the coders. To mitigate the risk, similar to prior work [4, 30, 46, 81], two authors

discussed and agreed about all of the labels. Moreover, we make our coding results available online[7] for others to scrutinize. In addition, to support the statistical aspect of our qualitative analysis, we applied saturation technique based on prior work [99]. To strengthen this aspect more, one future approach would be to perform post-survey that asks developers how much our results are valid in reality.

3.6.3 External Validity

External threats are concerned with our ability to generalize our results. We focus our study on the OpenStack and Qt communities, since those communities have made a serious investment in code review for several years (see Section 3.2.1). Although we conduct a multiple case study on four subject systems from those communities, we may not be able to generalize our conclusions to other systems. Replication studies are needed to reach more general conclusions.

3.7 Conclusions

Code review is a common software quality assurance practice. During the review process, reviewers critique patches to provide authors with feedback. In theory, this provides a simple feedback loop, where reviewers provide criticism and authors update patches. In practice, this loop is complex because reviewers may not agree about a patch, with some voting in favour and others voting against integration.

In this paper, we set out to better understand patches with divergent scores. To that end, we study patches with divergent scores in the OpenStack and Qt communities, making the following observations:

- Patches with divergent scores are not rare in OpenStack and Qt, with 15%–37% of patches that receive multiple review scores having divergent scores.

- Patches with divergent scores are integrated more often than they are abandoned.

- Patches with divergent scores that are eventually integrated tend to involve one or two more reviewers than patches without divergent scores. Moreover,

core reviewers provide negative scores after positive ones 70% of the time on average.

- Divergent discussion tends to arise early, with 75% of divergencies occurring by the third (Qt) or fourth (OpenStack) revision.
- Patches that are eventually abandoned with strong divergent scores more often suffer from external issues than patches with weak divergent and without divergent scores. Moreover, internal concerns are raised at greater or equal rates in patches with strong divergent scores and those without divergent scores.

Based on our results, we suggest that: (a) software organizations should be aware of the potential for divergent discussion, since patches with divergent scores are not rare and tend to require additional personnel to be resolved; and (b) automation could relieve the burden of reviewing external concerns.

4. The Important Factors on Keeping Participation at a Fine-Grained Level

4.1 Introduction

Code review is widely recognized as best practice for Software Quality Assurance [94]. The highly structured processes of Fagan-style reviews [23] are renown for being time-consuming in nature. Modern code review (MCR) is a lightweight process—developers informally interact with other developers and discuss patches in a web-based review tool such as Gerrit Code Review. Companies such as Microsoft, Facebook and several OSS projects have successfully adopted the MCR process [4].

However, code reviews today still have the potential for being expensive and slow [72], especially in terms of the discussion size before a final decision is made. For example, Microsoft engineers raised concerns over modern code review workflows, stating that *'current code review best practice slows us down'* [20]. Moreover, it is generally undesirable to have bloated reviewer comments in a review system [7]. In fact, the design of most modern code review systems promotes the practice of *lazy consensus* concept where reviewers reply only to the necessary content to avoid excessive comments.

Our main objective in this paper is to understand the most important perspectives on reviewer commenting. To fulfill this, we analyze the phenomena; namely, how many comments and words are involved to complete reviews, how reviewer commenting evolves over time, and what features drive reviewer comments. The goal of the work is to understand how excessive or underwhelming comments can be identified and managed.

To this end, we conduct the large-scale exploratory and modeling studies of over 1.1 million reviews across five open source projects. We form three research questions to guide those studies:

- *(RQ_1) Is there a typical number of reviewer comments before the final decisions?*

  Motivation: Towards understanding the most impactful features on reviewer commenting, we first would like to explore the phenomenon of reviewer com-

menting. Hence, the study investigates the extent to which the number of reviewer comments and words in the comments range across studied systems. The analysis provides with the evidence for understanding how many comments and words normally suffice to complete or manage reviews.

Results: There is no typical number of review comments across five studied systems. Reviewers reach a decision on a review ranging up to 13 comments, with 22 words per a comment on average. Moreover, the number of comments is mostly correlated with the number of words in the comments.

- *(RQ_2) Does reviewer commenting change over the evolution of a project?*
  Motivation: The evolution of reviewer commenting is also an essential aspect to thoroughly understand how reviewers have changed their behaviours in commenting. This analysis enriches our observations of RQ1.

  Results: The number of comments tends to steadily increase over time in the largest studied system. Furthermore, the number of comments tends to mostly stabilize in other studied systems.

- *(RQ_3) What (a) patch, (b) human and (c) management features drive reviewer comments?*
  Motivation: Our main objective of this paper is to discover the most impactful features in reviewer commenting through a modeling study. The results (i) allow us to have a deep understanding of the key features needed to manage reviewer commenting; and (ii) provide us with considerable findings towards future code review studies one of which is a review cost estimation approach.

  Results: Human experience (e.g., reviewer experience) and patch property (e.g., patch churn) features drive reviewer comments and words in the comments. Moreover, both novice authors and experienced reviewers tend to induce more comments and words, while the large and widespread modifications also have the tendency to raise more comments and words.

Our contributions are two-fold, with key implications listed in this paper. The first contribution is an exploratory study for the phenomenon of reviewer commenting. The second is the confirmation of review-related features that drive

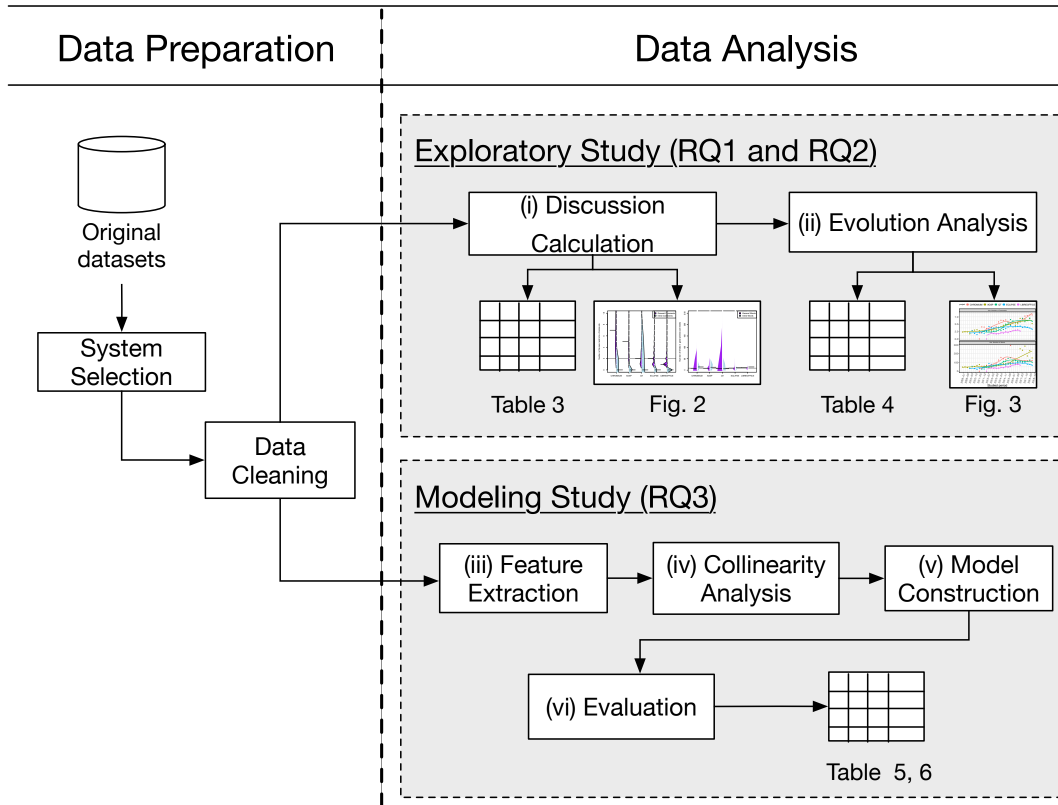

Figure 7: Overview of our methodology.

reviewer comments. We envision the work as a study towards a review cost estimation approach. Our replication package that includes our dataset and scripts is available online.[19]

Paper Organization. The rest of this section is organized as follows. Section 4.2 describes our methodology. Section 4.3 describes our results. Section 4.4 presents our practical suggestions. Section 4.5 presents threats to validity. Finally, Section 4.6 draws our conclusion and future work.

4.2 Methodology

Figure 7 depicts the overall methodology used in the study. It is broken into two parts, data preparation and data analysis. We describe in detail each part.

[19]`https://bitbucket.org/toshiki-h/commentanalysis`

4.2.1 Data Preparation

Figure 7 shows how Data Preparation involves two processes of (1) system selection and (2) data cleaning.

System Selection. Table 8 shows our studied systems, which are Chromium, AOSP, Qt, Eclipse, and Libreoffice. These five systems use the Gerrit Code Review. In addition, our selected systems have been used broadly in code review studies [88] [25] [96] and span over nine years. We first collect the review history (e.g., code details) through Gerrit REST API.[20] The API allows users to send queries to obtain the detailed information of patch change, reviewer and author statuses, and general and inline comments. For example, the query `https://codereview.qt-project.org/changes/102938` provides users with the basic information of Qt review #102938.[21] We collect multiple types of information that we need to perform our three research questions. After collecting this information, we store those types of collected data into Mongo DB.

Data Cleaning. We clean data by removing noise in review discussion history. We are only concerned with comments that were provided by human developers. Our data clearing process comprises three steps.

Step 1. We remove comments that were generated by review bot systems. To exclude those comments, we first identify comments that include auto-generated messages (e.g, "Major sanity problems found") by review bot systems. Moreover, since review bots in our studied systems include the keyword "Bot" in their account names (e.g., Sanity Bot), we also identify comments that are given by accounts whose names include the keyword. The auto-generated messages and bot accounts that are detected by our approach are shown in the scripts of our replication package.

Step 2. We also remove build log comments that were generated by continuous integration systems. A build log comment can be identified by searching formatted messages that are generated at the beginning of its comment (e.g., "Build succeed"). Our replication package includes a list of those formatted build log messages.

Step 3. After removing those comments, we exclude reviews in which there

[20] `https://gerrit-review.googlesource.com/Documentation/rest-api.html`

[21] `https://codereview.qt-project.org/c/qt-creator/qt-creator/+/102938`

is neither a reviewer comment nor an author comment. More specifically, after excluding comments that were generated by review bots and continuous integration systems, we count the number of general and inline comments. When there is then at least one comment, we use the review in our study. We repeat the same procedure for each studied system. Finally, to study reviews whose processes have been completed, we select reviews that were labeled "MERGED" or "ABANDONED" in our database.

4.2.2 Data Analysis

As shown in Figure 7, we describe in detail the approaches we use to answer all three research questions. Those approaches are broken into six stages.

Approach for RQ$_1$ The approach is the *(i) Discussion Calculation* stage. For this research question, we set out to understand the extent to which reviewers provide comments to complete reviews. To do so, we compute how many comments each review involves during its reviewing. In addition, to consider comment size aspect, we compute how many words are included in a reviewer comment. Moreover, in Gerrit, any developer is able to provide a general comment (i.e., placed on a reviewing board) and an inline comment (i.e., placed on a specific line of modified code). Thus, we investigate the number of general and inline comments and the number of words in a general and an inline comment separately, and analyze their distributions in our studied systems by using the `beanplot` function

Table 8: An overview of our five studied systems.

Product	#Studied Reviews	Studied Periods	#Developers
Chromium	498,821	04/2011–10/2018	7,187
AOSP	282,203	10/2008–10/2018	6,847
Qt	207,217	07/2011–10/2018	2,446
Eclipse	116,353	04/2012–10/2018	2,143
Libreoffice	50,750	04/2012–10/2018	846
Total	1,155,344	32 years	19,469

Table 9: The description of our selected features in Patch Property, Human Experience and Project Management segments.

Feature	Description
Patch Property	
Patch Churn	Number of lines added to and deleted from changed files.
#Subsystems	Number of subsystems that are changed.
#Directories	Number of directories that are changed under the subsystem(s). For example, in QT review #265820, since the modification appears in three different directories (coreplugin, projectexplorer, vcsbase) under one subsystem (qt-creator), #subsystems and #directories are counted as one and three, respectively.
Description Length	The length of a commit message i.e., Number of words.
Purpose (Doc)	Whether the purpose of a patch is documentation.
Purpose (Feature)	Whether the purpose of a patch is feature introduction.
Human Experience	
Author Experience	Number of reviews that an author has submitted in past development. For example, suppose an author has made three submissions under subsystem A and two submissions under subsystem B in a past development, we count the author's experience as five (submissions).
Reviewers Experience	Number of reviews that reviewers have reviewed in past development. For example, suppose a review involves two reviewers one of which has reviewed five past submissions and another has reviewed ten past submissions, we count the reviewers' experience as 15 (reviews).
Project Management	
Overall Workload	Number of reviews that have been submitted in a certain period (i.e., within 7 days).
Directory Workload	Number of reviews that have been submitted under the same directory in a certain period (i.e., within 7 days).
#Days since Last Modification	Number of days since the last time when a file in a patch was modified
#Prior Defects	Number of prior defects that have appeared in a file of a patch under review.

in the `beanplot` R package.

Approach for $\mathbf{RQ_2}$ The approach is the *(ii) Evolution Analysis* stage. In this analysis, we study the extent to which the number of comments and the number of words in the comments have changed over time. To conduct the analysis, we compute the average number of comments per a developer and the average number of words per a developer throughout a studied timeframe. Those two metrics can show how reviewers have changed their behaviours in commenting as studied systems evolved.

Approach for $\mathbf{RQ_3}$ In Figure 7, our $\mathrm{RQ_3}$ approach is split into three parts. We first describe a feature extraction to prepare studied features. Then, we describe collinearity analysis and model construction. Finally, we explain how we evaluate our models to find features that impact the number of comments and number of words in the comments per a review. We explain in detail each stage.

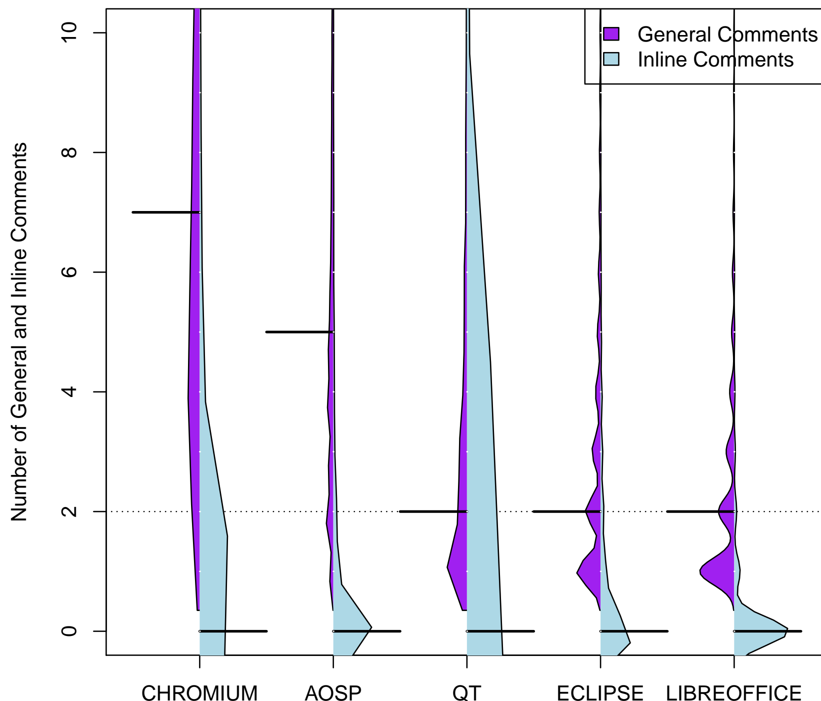

Figure 8: The distributions of number of comments per a review.

The *(iii) Feature Extraction* stage studies the overall perspectives in code review activities. We choose features based on prior work [87]. Table 9 describes each feature with its definition. In summary, our selected features are divided into three segments. The first segment is *Patch Property* in which its features characterize patch information. The second segment is *Human Experience* that provides features related to the activity of developers who have submitted or reviewed patches in the past. The third segment is *Project Management* where features quantify the extent to which the workload is intense in a project.

The *(iv) Collinearity Analysis* stage identifies and removes highly correlated features and redundant features due to the risk for disturbing results of our modeling analysis. Highly correlated features are features that have a high correlation with other ones. However, after removing highly correlated features, some

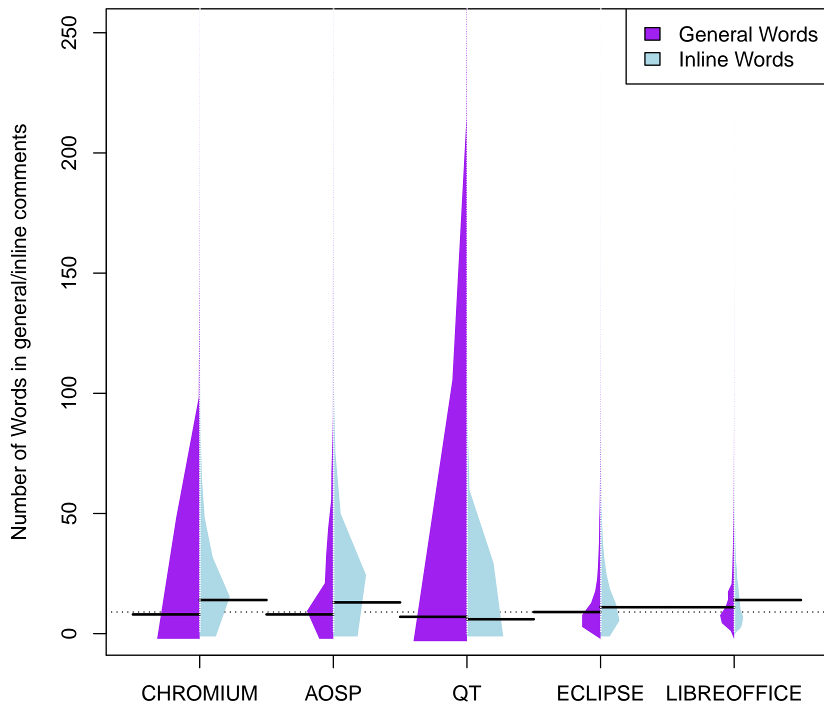

Figure 9: The distributions of number of words per a general or inline comment.

features might still quantify the same phenomenon and show homogeneous outcomes. Those features are considered as the redundant features. To counteract those two types of features, we conduct two methods to eliminate high correlation and redundancy between our features. We measure the possible correlation among our features using Spearman's rank correlation coefficients (ρ) to remove highly correlated features. Similar to prior work [87], we use Spearman's rank correlation coefficient $|\rho| = 0.7$ as the threshold to eliminate highly correlated features. Furthermore, features that do not have a high correlation might still be redundant and result in misleading conclusions [42]. To remove them, we use the `redun` function in the `rms` R package. The `redun` function uses a flexible additive model to predict each feature from remaining features to find the redundant

feature(s).[22]

The *(v) Model Construction* stage is when we train regression models using our features. A regression model is a commonly used type that describes the relationship between a dependent variable and independent variables. From our preliminary investigation, we decided to use a logarithmic scale in independent variables for the best performance. To construct models, we use the `lm` function in the R package. Moreover, since our features are computed in their different approaches, we standardize the coefficients that our regression models generate to measure the degree of impact for each feature. To compute standardized coefficients, we use the `lm.beta` function in the `lm.beta` R package.

Finally, the *(vi) Evaluation* stage analyzes the importance of each feature for general and inline comments. The evaluation is done by computation of the standardized coefficient of each feature that explains the degree to which the feature impacts the number of comments and number of words in the comments. We then analyze each high value of the coefficients and use them as discussions for rationale.

4.3 Results

We provide the results to each of the research questions. First, we highlight our findings and then answer each question.

4.3.1 Answering RQ_1

Table 10 and Figure 9 show the results of RQ_1. We were able to make two observations as outlined below.

The number of comments and number of words in a comment vary among reviews and across systems. Table 10 shows that the number of total comments approximate 9–13 on average in the two largest studied systems (i.e., Chromium and AOSP). In contrast, Qt, Eclipse, and Libreoffice reviews comprise 3–7 comments on average. We assume that the results are reflective of a system size (i.e., the number of reviews). Indeed, we observe that the larger a system size is, the more comments the system's reviews garner (see Table 8

[22]`https://www.rdocumentation.org/packages/Hmisc/versions/4.2-0/topics/redun`

Table 10: The statistics of numbers of total comments, general comments and inline comments for each system.

Project	Min.	1st Qu.	Med.	Mean	3rd Qu.	Max.
Total Comments (i.e., the sum of general and inline comments)						
Chromium	1	4	8	12.59	14	1,372
AOSP	1	3	6	9.27	11	503
Qt	1	1	3	6.68	6	2,632
Eclipse	1	1	3	5.23	5	338
Libreoffice	1	1	2	2.99	3	81
General Comments						
Chromium	1	4	7	10.49	13	905
AOSP	1	3	5	7.86	10	247
Qt	1	1	2	4.06	5	367
Eclipse	1	1	2	3.31	4	106
Libreoffice	1	1	2	2.37	3	34
Inline Comments						
Chromium	0	0	0	2.10	1	1,144
AOSP	0	0	0	1.41	1	365
Qt	0	0	0	2.62	1	2,630
Eclipse	0	0	0	1.92	1	232
Libreoffice	0	0	0	0.62	0	70

and Table 10). Furthermore, inline comments have the tendency to include more words than general comments. Figure 8 and Figure 9 show that despite general comments are provided more frequently than inline comments, inline comments tend to involve more words than general comments. For example, Figure 9 shows that the median of the number of words in a inline comment is larger than the number of words in a general comment in Chromium, AOSP, Eclipse, and Libreoffice.

Prior studies found that useful comments are considered as comments that trigger code changes [15], and are different from non-useful comments with regard to several textual properties [68]. Basically, inline comments are provided to trigger specific code changes, requiring much context to clarify such needs. Our results complement prior studies, suggesting that inline comments have the potential to include more insights than general comments in terms of comment size.

Table 11: The correlations between the number of general and inline comments and the number of words in general and inline comments.

Comment Type	Chromium	AOSP	Qt	Eclipse	Libreoffice
General	0.23	0.65	0.30	0.74	0.76
Inline	0.89	0.83	0.67	0.85	0.77

Reviewers treat general and inline comments differently. Table 10 and Figure 8 show that general comments often appear during reviewing at 3–13 comments on average, whereas reviewers rarely provide inline comments. We suspect that developers tend to discuss the abstract impact of the patch rather than looking into specific modification or enhancement of the submitted patch. Interestingly, the distributions in Figure 8 visually show that developers tend to write more general comments than inline comments. Indeed, this figure shows that a majority of the Libreoffice reviews contained no inline comments (i.e., the inline count is zero). Table 10 shows that 25% of reviews (above 3rd Qu.) in the two most studied and largest systems (i.e., Chromium and AOSP) used more than ten general comments to make their final decisions. Although MCR is known as lightweight [4], those systems have the potential for requiring more comments.

Table 11 shows that the number of inline comments per a review have a substantial correlation (0.67–0.89) with the number of words in the inline comments per a review across our studied projects. This result indicates that the number of words in inline comments is mostly consistent in each studied system, proportionally increasing with the number of inline comments. Indeed, we find that AOSP, Eclipse, and Libreoffice systems have a substantial correlation (0.65–0.74) between the number of general comments and the number of words in the general comments, while Chromium and Qt systems do not. General comments are provided not only to discuss the abstract impacts, but also to simply show their agreements, which may cause the inconsistent degree of correlation in general comments among studied systems.

We return to answer *(RQ_1) Is there a typical number of reviewer comments before the final decisions?*:

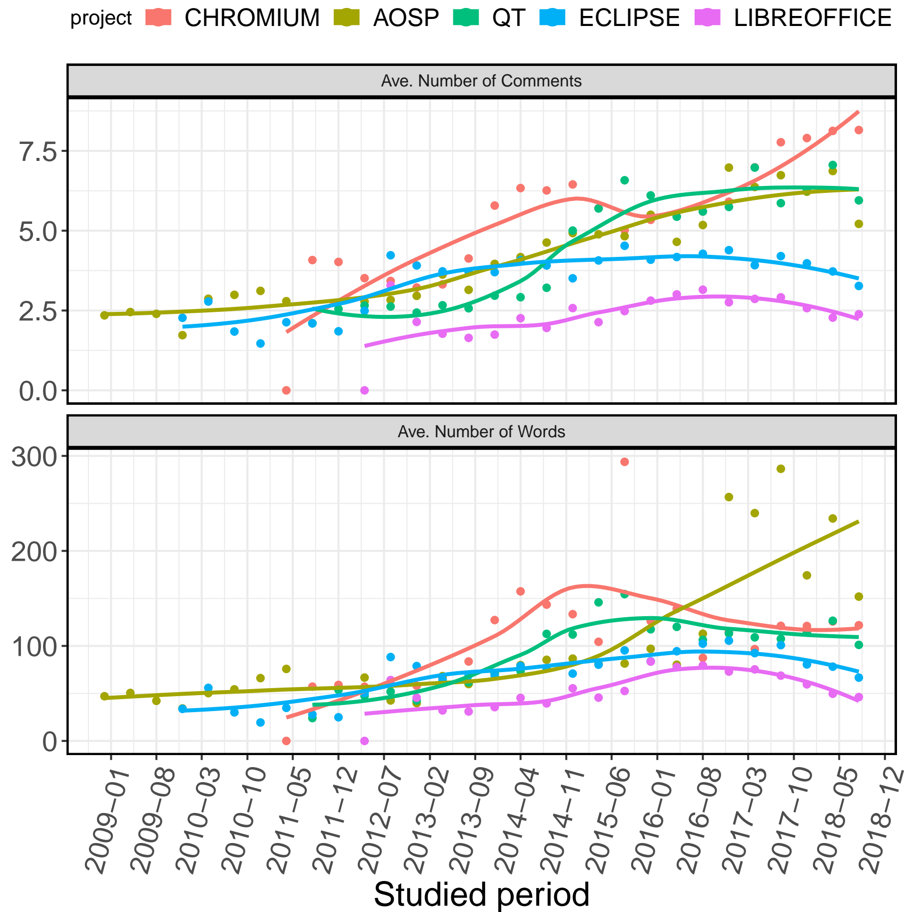

Figure 10: The evolution of the (i) number of total comments per a developer and (ii) number of total words per a developer throughout a studied timeframe.

There is no typical number of review comments across five studied systems. Reviewers reach a decision on a review ranging up to 13 comments, with 22 words per a comment on average. Moreover, the number of comments is mostly correlated with the number of words in the comments.

4.3.2 Answering RQ_2

Figure 10 shows the results of RQ_2. From this figure, we make two following observations:

Reviewer comments have increased over time for Chromium. Figure 10 shows that the average number of total comments that a reviewer provides in a review has increased over time in the largest system CHROMIUM. For example, the system reached the first peak in late 2014. After this peak, its average number of total comments per a developer grew rapidly again in recent months. Conversely, the number of words per a developer has decreased after the peak in CHROMIUM, implying that reviewers are likely to participate more actively rather than just commenting. Our results complement prior work [72], suggesting that the number of comments in modern code review environments has the potential to increase as a system evolves.

In contrast, reviewer comments tend to stabilize or stall in other studied systems. Figure 10 shows that the average number of total comments per a developer in AOSP and QT systems has increased until early 2016, while the average number roughly tended to stabilize in recent months. Although the number of total comments per a developer in ECLIPSE and LIBREOFFICE has the slight increase from their initial start until mid 2016, those numbers stalled in recent months. We assume that the project which has a frequent release schedule may induce more comments because developers feel pressure to catch up with every release. Indeed, the CHROMIUM system has a major release every month, while the rest of our studied systems release a main version once every six months or twelve months. Moreover, Figure 10 also shows that the number of words per a developer has a stable tendency to the number of comments per a developer in AOSP, ECLIPSE, and LIBREOFFICE over time. For example, the number of comments per a developer and number of words per a developer in LIBREOFFICE gradually increased until late 2016. After that time, the decrease tendency has been found in recent periods. We observe that an increase in correlations between comment and word aspects (see Table 11) across AOSP, ECLIPSE, and LIBREOFFICE had led to the similar tendencies of the evolution between the number of comments per a developer and the number of words per a developer. However, Figure 10 shows that the number of words per a developer in CHROMIUM and QT has decreased, while the number of comments per a developer in those systems has increased rapidly or gradually over time. This inverse result might be related to the weak correlations of those two systems that we showed in the

Table 12: The standardized coefficient of each feature for general comments and words in the general comments. The blue colour cell depicts the most impactful metric.

Feature	General Comments					General Words				
	Chromium	AOSP	Qt	Eclipse	LibreOffice	Chromium	AOSP	Qt	Eclipse	LibreOffice
Patch Property										
Patch Churn	0.16	0.13	0.15	0.12	0.05	0.10	0.06	0.11	0.12	0.01
#Subsystems	0.04	–	0.14	0.01	0.04	0.03	–	0.18	0.01	0.04
#Directories	0.11	–	-0.04	–	0.01	0.02	–	-0.05	-	0.01
Description Length	0.02	0.07	0.10	0.12	0.03	0.06	0.08	0.08	0.14	0.05
Purpose (Doc)	0.02	0.01	-0.01	< 0.01	0.02	< 0.01	< 0.01	-0.02	< 0.01	0.02
Purpose (Feature)	0.03	0.03	0.02	0.01	< 0.01	0.02	0.01	< 0.01	-0.01	-0.01
Human Experience										
Author Experience	-0.10	-0.04	-0.13	-0.10	-0.21	-0.08	0.01	-0.23	-0.14	-0.23
Reviewers Experience	0.34	0.41	0.19	0.28	0.23	0.68	0.75	0.24	0.36	0.27
Project Management										
Overall Workload	0.10	0.16	0.02	< 0.01	-0.02	0.02	0.04	0.05	-0.07	-0.03
Directory Workload	0.04	< 0.01	0.01	< 0.01	0.01	0.07	-0.01	0.04	-0.01	< 0.01
#Days since Last Modification	0.03	0.02	0.01	< 0.01	0.03	0.04	0.03	< 0.01	< 0.01	0.01
#Prior Defects	-0.03	0.03	0.05	< 0.01	0.02	-0.07	0.01	0.02	-0.01	0.01

second observation of the previous research question.

We now return to answer *(RQ_2) Does reviewer commenting change over the evolution of a project?*:

> The number of comments tends to steadily increase over time in the largest studied system. Furthermore, the number of comments tends to mostly stabilize in other studied systems.

4.3.3 Answering RQ_3

Table 12 and Table 13 show our twelve features that remain after collinearity and redundancy analyses. To answer RQ_3, we make two observations.

Human experience features can drive general comments across studied systems. Table 12 shows that for both the number of general comments and number of words in the general comments, reviewer experience is the most impactful feature in all studied systems. Specifically, reviewer experience has an increase impact across our studied systems, indicating that experienced reviewers tend to provide more general comments and include more words. This is because experienced reviewers are capable of pointing out broader issues than inexperienced ones. In contrast, author experience shows a decrease impact across our studied systems, suggesting that since experienced authors are more knowledgeable than novice ones, they can address issues that may induce discussion before

Table 13: The standardized coefficient of each feature for inline comments and words in the inline comments. The blue colour cell depicts the most impactful metric. Note that the standardized coefficient of Patch Churn (0.145) is larger than of Reviewers Experience (0.138) in Eclipse.

Feature	Inline Comments					Inline Words				
	Chromium	AOSP	Qt	Eclipse	Libreoffice	Chromium	AOSP	Qt	Eclipse	Libreoffice
Patch Property										
Patch Churn	0.22	0.19	0.10	0.14	0.10	0.30	0.22	0.29	0.16	0.10
#Subsystems	0.04	–	< 0.01	-0.01	0.02	0.05	–	0.02	-0.02	0.03
#Directories	-0.02	–	0.03	–	-0.01	-0.04	–	-0.04	–	-0.01
Description Length	0.03	0.04	-0.01	0.05	-0.02	0.03	0.07	0.11	0.08	-0.03
Purpose (Doc)	0.03	0.01	< 0.01	0.01	< 0.01	0.02	0.01	0.01	< 0.01	< 0.01
Purpose (Feature)	0.03	0.02	< 0.01	0.02	0.01	0.04	0.02	0.02	0.02	0.01
Human Experience										
Author Experience	-0.12	-0.05	-0.01	-0.08	-0.14	-0.15	-0.06	-0.13	-0.10	-0.19
Reviewers Experience	0.10	0.01	0.02	0.14	0.08	0.24	0.11	0.21	0.30	0.17
Project Management										
Overall Workload	< 0.01	0.04	< 0.01	0.02	0.01	-0.01	0.01	-0.01	0.02	0.04
Directory Workload	0.01	< 0.01	-0.01	0.01	0.02	0.05	0.03	< 0.01	0.02	0.03
#Days since Last Modification	0.02	< 0.01	< 0.01	0.01	0.04	0.08	0.04	0.03	0.03	0.06
#Prior Defects	-0.01	0.03	< 0.01	0.01	0.02	-0.07	0.03	0.07	< 0.01	0.05

they submit patches. We find that the more the author is experienced, the less likely that the number of general comments and the number of words in the general comments will decrease across our studied systems.

Table 12 also shows that patch churn can be observed as a relatively impactful feature in Chromium, AOSP, Qt, and Eclipse for the number of general comments and number of words in the general comments. This result suggests that the larger the patch churn is, the more likely the review will receive comments and words. Moreover, dispersion of changes (i.e., #Subsystems and #Directories) mostly shows the increase impact on the number of general comments and number of words in the general comments. The results imply that when modification spreads across multiple subsystems or directories, reviewers may raise a broader discussion to avoid unexpected problems that might affect external components. Indeed, maintainability issues are raised more frequently than functional defects in reviews [53] [11].

The project management segment shows both increase and decrease impacts across our studied systems. For example, overall workload feature is increased and relatively impactful compared to other features in AOSP. The results imply that the impact of the project management segment varies due to a system size. Indeed, our RQ2 shows that the growth of the number of comments and number of words in the comments vary across systems (see Figure 10).

Patch property features are likely to drive inline comments. However, different from the number of general comments and number of words in the general comments, patch churn feature plays a considerable role on both the number of inline comments and number of words in the inline comments. Table 13 shows that Patch Churn is ranked at the 1st place in CHROMIUM, AOSP, QT, and ECLIPSE. Similarly, the feature is most considerable in CHROMIUM, AOSP, and QT for the number of words in the inline comments. This result indicates that if the modified lines of code increase, the number of inline comments and number of words in the inline comments will rise due to various issues (e.g., typo, code style).

Table 13 also shows that author experience has a decrease impact on the number of inline comments and words in the inline comments, similar to general comments. Moreover, reviewer experience has an increase impact on the number of inline comments and the number of words in the comments. Similar to our previous observation, our results suggest that novice authors and experienced reviewers can increase the number of inline comments and number of words in the inline comments. Besides, both increase and decrease impacts on the number of inline comments and number of words in the inline comments are shown in overall workload and directory workload features across our studied systems, implying that the impact of project management features for inline comments depends on a system's size.

We now answer *(RQ_3) What (a) patch, (b) human and (c) management features drive reviewer comments?*:

Human experience and patch property features drive reviewer comments and words in the comments. Moreover, both novice authors and experienced reviewers tend to induce more comments and words, while the large and widespread modifications also have the tendency to raise more comments and words across studied systems.

4.4 Practical Suggestions

We discuss practical suggestions and actionable contributions from our findings.

1. **There are no magic number of comments and number of words**

to complete reviews. Our results suggest that reviewers can complete a review in 3–13 comments on average. This is a wide range, showing that review participation can range and cannot be fit into a smaller range. Importantly, it is possible to identify review participation (number of comments) and the comment size (number of words in a comment) that go outside of their ranges as either underwhelming or becoming bloated. This can become actionable, as we can now flag and identify these abnormal reviews outside of their ranges.

2. **The number of comments and number of words fluctuate over time.** Our results show that the number of comments and number of words in the comments change over the lifetime of the system. Moreover, we also find that the number of comments and the number of words in the comments have the potential to increase as studied systems grow. Due to a lazy consensus i.e., reviewers responding only to necessary contents, a lack of interest or maybe if there is an overload of developers, there is no incentive to respond to every comment. For future work, we would like to also explore how commenting is affected by the external factors, such as being a newcomer to a project, the increase in source code and review requests, and the maturity of the software project. The actionable implication is that we now know that outside factors have an impact on reviewer commenting.

3. **Human experience and patch property features have the strongest impact on general and inline comments, respectively.** In RQ_3, our results show that novice authors or experienced reviewers can increase the number of general comments and number of words in the general comments across our studied systems. Our results also show that patch churn and dispersion of modified codes can increase the number of inline comments and number of words in the inline comments. We suggest that to minimize the discussion, the experience of the author and reviewers and the property size of patches should be more carefully considered before starting with the discussion.

We envision this paper as working towards a review cost estimation approach.

In our future plans, we would like to (i) analyze the relationship between review cost and quality; and (ii) build models that automatically determine the extent to which a review will take a certain time by machine learning techniques. We envision that this work can also open up research into the effective management of code reviews.

4.5 Threats to Validity

We present construct, internal and external threats to our study.

Construct Validity Construct threats to validity refer to the concerns between the theory and achieved results of the study. To prevent noise from interfering with our observations, we clean our raw data by removing potential noisy comments. However, there might be cases where our comment extraction mistakenly excludes actual human comments or includes comments of CI or Bot systems. Thus, we preliminarily investigated continuous integration log and bot comments, and then excluded comments that were generated by those automated tools.

Internal Validity Internal threats to validity refer to the concerns that are internal to the study. We discuss two threats. The first threat is the coverage of the features used in the study. We analyze a broad range of patch property, human experience, and project management features; nonetheless, our study has not covered other possible aspects. Therefore, to mitigate the threat, we select features based on similar prior work [43] [25] [87].

The second threat is the validity of our methodology for the modeling study. There might be cases where we could use another sophisticated algorithm. However, we choose regression models which prior studies have used for years [55] [87].

External Validity External threats to validity refer to the generalization concerns of this study's results. The main external threat is the generality of our studied systems. Although we collected studied systems based on their popularity and quality of their datasets, there might be a threat such that our results do not generalize to all software systems. We chose five studied systems in terms

of their popularity in the code review research field to ensure that our studied systems show valid information. Therefore, we are confident that we have addressed the threads of our studied systems as they have been used in prior studies [88] [86] [54] [55] and were released as official MSR datasets [31] [96].

4.6 Conclusions

We conducted exploratory and modeling studies over 1.1 million reviews across five studied systems (i.e., Chromium, AOSP, Qt, Eclipse, and Libreoffice). Through those studies, our results indicate that:

- The number of both general and inline comments varies among reviews and across studied systems.
- Reviewers change their behaviours in commenting as a system evolves.
- Reviewer comments are most likely to be affected by developer experience and patch property size.

The main goal of this work is to move towards a review cost estimation approach for effective code reviews in the modern code review. We also hope that this research has implications for future research.

5. The impact of Review Linkage on Code Review Analytics

5.1 Introduction

Modern Code Review (MCR)—a lightweight variant of traditional code inspections [23]—allows developers to discuss the premise, content, and structure of code changes. Many communities adopt a Review-Then-Commit (RTC) philosophy, where each code change must satisfy review-based criteria before integration into official repositories is permitted. Since MCR tools archive reviewing activities (e.g., patch revisions, review participants, discussion threads), active development communities generate plenty of MCR data.

Researchers have proposed analytics approaches that leverage MCR data to support practitioners. For example, reviewer recommenders [6,65,88,95,100] help developers to select appropriate reviewers and review outcome predictors [28,39,40] estimate the likelihood of a code change eventually being integrated.

When performing code review analytics, each review has traditionally been treated as an independent observation; yet in practice, reviews may be interdependent. By ignoring connections between reviews, review analytics approaches may underperform. For example, the discussion for Review #314319[23] of the OpenStack Neutron project occurred on its linked Review #225995. Without considering the comments on the linked review, a review outcome predictor would mistakenly presume that Review #314319 had no discussion, and would thus be unlikely to be integrated. Moreover, Review #134811[24] of the OpenStack Nova project is abandoned because a competing solution in the linked Review #134853 was integrated. To ensure a fair review process [26], reviewer lists of competing solutions may need to be synchronized; yet links are not analyzed by today's reviewer recommenders.

In this paper, we set out to better understand the impact of review linkage on code review analytics. To do so, we extract the review linkage graph from six active MCR communities—a directed graph where nodes represent reviews and

[23] https://review.openstack.org/#/c/314319/
[24] https://review.openstack.org/#/c/134811/

edges represent links between reviews. We analyze and leverage those graphs to address the following four research questions:

(RQ1) To what degree are reviews linked?

Motivation: To gain an initial intuition about the connectedness of reviews, we first set out to quantitatively analyze the extracted review linkage graphs.

Results: Linkage rates range from 3% to 25%. Linkage tends to be more common in the two largest communities (25% in OpenStack and 17% in Chromium), likely because they have invested more heavily in code reviewing. Indeed, these communities have significantly more comments and reviewers per review (pairwise Mann-Whitney U tests with Bonferroni correction, $\alpha = 0.01$; and non-negligible Cliff's delta effect sizes).

(RQ2) Why are reviews being linked?

Motivation: The potential reasons for review linkage are manifold. To explore these reasons, we set out to qualitatively analyze recovered links between reviews.

Results: Using open coding [18] and card sorting [59], we discover 16 types of review links that belong to five categories, i.e., Patch Dependency, Broader Context, Alternative Solution, Version Control Issues, and Feedback Related. These different link types have different implications for review analytics techniques. For example, while Broader Context links indicate that a discussion may span across linked reviews, Alternative Solution links point out competing solutions.

(RQ3) To what degree can link categories be automatically recovered?

Motivation: The qualitative approach that we used to address RQ2 is not scalable enough for large-scale analyses of link categories. Hence, we want to explore the feasibility of training automatic classifiers to identify link categories.

Results: We train link category classifiers using five classification techniques. These classifiers at least double the performance of a ZeroR baseline, achieving a precision of 0.71–0.77, a recall of 0.72–0.92, and an F1-score of 0.71–0.79.

(RQ4) To what degree do linked reviews impact code review analytics?

Motivation: While RQ1–RQ3 suggest that linkage may impact reviewer analytics, the extent of that impact is unknown. We set out to quantify that impact

by comparing prior review analytics techniques [28, 100] to extended versions that are link aware.
Results: Code review analytics tend to underperform on linked reviews. In 41%–84% of linked reviews, reviewer recommenders omit at least one shared reviewer. Moreover, review outcome predictors misclassify 35%–39% of linked reviews. Link-aware approaches improve the F1-score of reviewer recommenders by 37%–88% (5–14 percentage points).

Our empirical study indicates that linkage is a rich activity that should be taken into consideration in future MCR studies and tools. In addition, this paper contributes a replication package,[25] which includes (a) review linkage graphs that feature 1,466,702 reviews and 231,341 links from the six studied communities; (b) 752 manually coded links spanning 16 types and five categories [RQ2]; and (c) scripts that reproduce our statistical analyses [RQ1, RQ3, RQ4].

Paper Organization. The remainder of this section is organized as follows. Section 5.2 describes general motivations for linkage analytics and the studied communities. Sections 5.3–5.6 present the experiments that we conducted to address RQ1–RQ4, respectively. Section 5.7 discusses the broader implications of our results. Section 5.8 discloses threats to the validity of our study. Finally, Section 5.9 draws conclusions.

5.2 Study Preparation

We first introduce general motivations for linkage-related analysis based on prior studies. Then, we explain our studied systems in this analysis.

5.2.1 Motivation

Linkage of related software artifacts has long been considered an important phenomenon. Canfora et al. [17] found that links between issue reports of different software projects are not uncommon. Boisselle and Adams [13] reported that 44% of bug reports in Ubuntu are linked to indicate duplicated work. Ma et al. [52] showed that linked issues delay the release cycle and increase mainte-

[25] https://github.com/software-rebels/ReviewLinkageGraph

Table 14: An overview of our subject communities.

Product	Language	Studied Period	#Reviews	#Revs	#Projects
OpenStack	Python	09/2011–01/2018	533,050	12,359	1,804
Chromium	JavaScript	04/2011–01/2018	364,079	7,442	410
AOSP	Java	10/2008–01/2018	229,210	7,614	1,049
Qt	C++	07/2011–01/2018	188,981	2,377	170
Eclipse	Java	04/2012–01/2018	106,515	2,191	380
Libreoffice	Python	04/2012–01/2018	44,867	822	34
Total			1,466,702	32,805	3,847

nance costs. Moreover, they found that recovering a link is a difficult task, often taking developers more than one day to do by hand.

To ease the recovery of links, researchers have proposed automatic approaches. Antoniol et al. [2] applied Natural Language Processing (NLP) techniques to detect links between source code and related documents. Alqahtani et al. [1] proposed an automatic approach to recover links between API vulnerabilities. Guo et al. [29] used deep learning techniques that exploit domain knowledge to detect semantic links. Rath et al. [70] detect missing links between commits and issues using process and text-related features.

Linkage also appears in peer code review settings. Zampetti et al. [98] found that developers reference other resources in reviews to enhance documentation of pull requests. Perhaps the most similar prior work is that of Li et al. [50], who reported that 27% of GitHub pull requests from 16,584 Python projects on GitHub have links, which span six types. This paper expands upon the work of Li et al. by studying linkage in six large and successful software communities (rather than a broad sample of GitHub projects), and the impact of linkage on review analytics approaches.

5.2.2 Studied Communities

The goal of our study is to extract and analyze review graphs of large software communities that have invested in their MCR processes. To do so, we focus on the six popular communities that appear in the recent related work [31, 88, 96]. OpenStack is a cloud computing platform. Chromium is an open source web browser. AOSP is a mobile software platform. Qt is a cross-platform application framework. Eclipse is an Integrated Development Environment (IDE) and associated tools. Libreoffice is a free and open implementation of the Office software suite. The studied communities use Gerrit for code review. To conduct our study, we collect MCR data using the Gerrit API. Table 14 provides an overview of the collected data.

5.3 Review Graph Extraction (RQ1)

We set out to study the extent to which reviews are linked to one another in our studied communities. To do so, we extract review graphs from the studied communities. The review graph $RG = (R, L)$ is a directed graph with the following properties:

- Graph nodes R represent review entries. Each review entry $r \in R$ is comprised of a set of properties, such as ID_r (a unique review identifier), D_r (the change description), PR_r (the set of patch revisions), REV_r (the set of reviewers). GC_r and IC_r (general and inline comments, respectively).

- Graph edges L represent links between review entries. Each edge $l \in L$ has a type T_l that describes why the link was recorded, which we study in Sections 5.4–5.6.

The set of reviews R is what has typically been extracted and analyzed by prior work on code review. We first propose a lightweight approach to recover graph edges L from the D_r, GC_r, and IC_r fields of each $r \in R$ (5.3.1). Then, we apply that approach to the studied communities and analyze the extracted graphs to address RQ1 (5.3.2).

Table 15: Review graph characteristics in the subject communities.

Product	Linked Reviews		Per Review (Mean)		Per Reviewer	Description		General Comment		Inline Comment		Cross-project	
	#Reviews	%Reviews	Comments	Reviewers	Comments	#Links	%Links	#Links	%Links	#Links	%Links	#Links	%Links
OpenStack	133,650	25%	17	5	3	77,400	28%	151,411	54%	50,110	18%	110,964	40%
Chromium	62,065	17%	12	3	4	10,295	5%	182,929	93%	3,968	2%	44,869	23%
AOSP	11,584	5%	8	2	3	6,082	22%	21,491	77%	466	2%	2,508	9%
Qt	14,266	8%	8	2	3	574	3%	19,450	87%	2,309	10%	4,353	19%
Eclipse	8,227	8%	6	2	3	1,255	6%	19,995	90%	917	4%	1,359	6%
Libreoffice	1,549	3%	4	2	2	211	10%	1,774	86%	88	4%	107	5%

5.3.1 Link Recovery Approach

We perform a preliminary analysis of 410 randomly selected reviews to gain insight into the linkage habits within the studied communities. We observe that links appear in review description fields (D_r), as well as within general and inline comment threads (GC_r and IC_r). Moreover, we observe two ways that links are recorded:

1. By Change ID (e.g., `Ic8aaa0728a43936cd4c6e1ed590e01b-a8f0fbf5b`), i.e., a 40-digit hexadecimal identifier assigned to each review at creation time (prefixed with an `I` to avoid confusion with Git commit IDs).

2. By URL (e.g., `https://review.openstack.org/#/c/1111`).

Thus, to recover links from a review $r \in R$, we scan its description D_r as well as all of the general and inline comments (GC_r, IC_r) using regular expressions. To detect Change IDs, our regular expression scans for terms of the form `I[0-9a-f]{40}`. To detect URLs, our regular expression is of the form `https?://PROJ/#/c/[1-9]+[0-9]*`, where `PROJ` is replaced with the base URL of the project (e.g., `review.openstack.org`). For the Chromium community, our regular expressions needed to be adapted to: `https?://chromium-review.googlesource.com/c/REPO/+/[1-9]+[0-9]*`, where `REPO` corresponds to any of the 410 repositories within the Chromium community. The link recovery process is repeated $\forall r \in R$.

5.3.2 Results

By applying our link recovery approach to the six studied communities, we set out to better understand (i) the prevalence of review linkage and (ii) linkage trends

over time. To do so, we measure a linkage rate for each studied system, i.e., the proportion of reviews that are connected by at least one link.

Prevalence of Linkage. Table 15 shows that 17% and 25% of Chromium and OpenStack reviews are connected by at least one link, respectively. Although the linkage rate in Qt, Eclipse, and AOSP reviews are lower (5%–8%), linkage is not uncommon. However, the Libreoffice linkage rate is only 3%. We suspect that this result is reflective of the differences in the importance that the studied communities have placed on code review. Indeed, reviews in the OpenStack and Chromium communities receive 17 and 12 comments on average, whereas reviews in other subject communities receive 4–8 comments on average. Two-tailed, unpaired Mann-Whitney U tests between OpenStack and the other studied communities (after Bonferroni correction to control for family-wise errors, $\alpha = \frac{0.05}{5} = 0.01$) indicate that OpenStack receives significantly more comments than the other studied communities ($p < 0.001$) with Cliff's delta effect sizes of negligible when compared to Chromium ($\delta = 0.021 < 0.147$), small when compared to AOSP and Qt ($0.147 \leq \delta = 0.288, 0.287 < 0.330$), medium when compared to Eclipse ($0.330 \leq \delta = 0.413 < 0.474$), and large when compared to Libreoffice ($0.474 \leq \delta = 0.580$). Furthermore, OpenStack reviews tend to involve more reviewers than the other studied communities, averaging two reviewers per review more than the next highest community (Chromium). Two-tailed, unpaired Mann-Whitney U tests indicate that OpenStack reviews have significantly more reviewers than the other studied communities ($p < 0.001$) with Cliff's delta effect sizes of medium when compared to Chromium ($0.330 \leq \delta = 0.407 < 0.474$), and large when compared to AOSP, Qt, Eclipse, and Libreoffice ($0.474 \leq \delta = 0.553, 0.520, 0.703, 0.712$). We also find that there is no tendency where the larger systems are likely to receive more comments in terms of an individual contribution (i.e., 2–4 comments per reviewer on average across our studied systems). Indeed, despite of the significant differences of unpaired Mann-Whitney U tests across our six studied systems, Cliff' s delta effect sizes are negligible when compared to Qt and Libreoffice ($\delta = 0.051, 0.020 < 0.147$), small when compared to AOSP and Eclipse ($0.147 \leq \delta = 0.219, 0.213 < 0.330$), and medium when compared to Chromium ($0.330 \leq \delta = 0.357 < 0.474$).

Table 15 also shows that the largest proportion of links are recovered from discussion threads (*GC*, *IC*). Indeed, 72%–97% of links are recovered from *GC* and *IC* threads. This indicates that links tend to emerge during the review rather than when it is created.

Furthermore, Table 15 shows that 5%–40% of links connect reviews across project boundaries. For example, Review #102704[26] links from the NOVA project to the project of OPENSTACK. Furthermore, OPENSTACK shows a greater rate of cross-project links than the other studied communities, likely because the OPENSTACK community develops more projects than the other studied communities. Indeed, Table 14 shows that OPENSTACK contains at least four times as many projects as the other studied communities.

Linkage Trends. Figure 11 shows that linkage rates in the four communities that have made the largest investments in code review (i.e., personnel and activity per review) have stabilized or peaked in the more recent studied periods. Indeed, the linkage rate in the OPENSTACK community has a rapidly increasing trend until late 2014 and more gradual increases in recent months. The CHROMIUM community began with moderate linkage rates until mid-2014, when rates dropped to below 5%. However, the rates have climbed above 30% in mid-2017. This growth may be due to several factors (e.g., community initiative, growth in task complexity). The QT and AOSP communities had linkage rates below 7% until mid-2014, when rates roughly stabilized at 13% and 9%, respectively.

On the other hand, linkage rates remain stable or are decreasing in the two studied communities that have made the least investment in code review. Indeed, the LIBREOFFICE community shows a stable trend, while the ECLIPSE community's trend is decreasing.

RQ1: Linked reviews occur regularly in communities that have made large investments in code review. Thus, code review analytics should look to linkage as a potential source of useful information.

[26] `https://review.openstack.org/#/c/102704/`

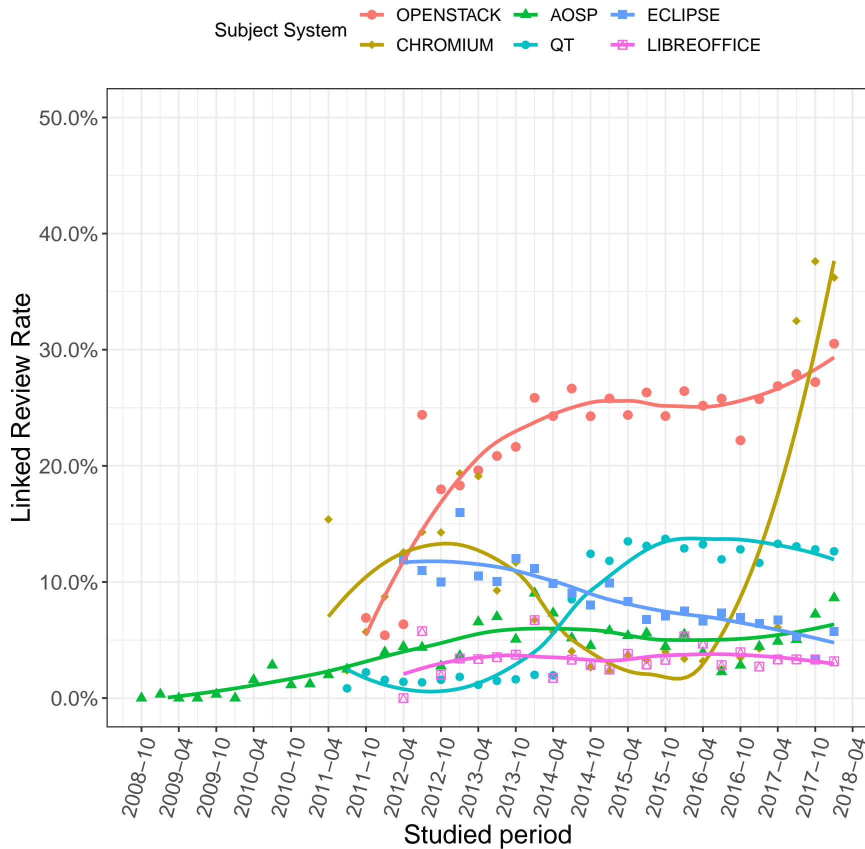

Figure 11: Monthly linkage rate in the studied communities.

5.4 Qualitative Analysis of Review Links (RQ2)

Reviews may be linked to other reviews for several reasons. For example, a link may express a dependency between reviews (e.g., "[Review #106918][27] is dependent on [Review #106274]") or the evolution of an earlier idea into its current form (e.g., "[Review #475649][28] is a follow up patch to [Review #411830]").

In this section, we set out to better understand the underlying reasons for review linkage through a qualitative analysis. To understand why a link has been recorded, we use a manually-intensive research method, which creates practical limitations on the breadth of communities that we can analyze. Thus, we elect to

[27] `https://review.openstack.org/#/c/106918/`
[28] `https://review.openstack.org/#/c/475649/`

analyze the OpenStack community—the largest and most dynamic graph in our set of studied communities (see Tables 14–15 and Figure 11). The OpenStack community is composed of several projects, of which, we select the two largest for analysis, i.e., Nova (the provisioning management module) and Neutron (the networking abstraction interface). Below, we describe our approach to classify links in the studied projects (5.4.1) and present the results (5.4.2).

5.4.1 Approach

We apply an open coding approach [18] to classify randomly sampled links between reviews. Open coding is a qualitative data analysis method by which artifacts under inspection (review links in our case) are classified according to emergent concepts (i.e., codes). After coding, we apply open card sorting [59] to lift low-level codes to higher level concepts. Below, we describe our sampling, coding, and card sorting procedures in more detail.

Sampling. Section 5.3 shows that there are 133,650 linked reviews in the OpenStack community, 16,144 of which appear in the Nova and Neutron projects. Since coding of all of these links is impractical, we randomly sample Nova and Neutron review links for coding.

To discover as complete of a set of link types as possible, we strive for saturation. Similar to prior work [99], we set our saturation criterion to 50, i.e., we continue to code randomly selected links until no new codes have been discovered for 50 consecutive links.

To ensure that we analyze links that appear in descriptions and comments, we aim to achieve saturation twice—once when coding description-based links and again when coding comment-based links. We reach saturation after coding 340 comment-based links and 146 description-based links in Nova, and 161 comment-based links and 105 description-based links in Neutron.

Coding. Coding was performed by the first and second authors during collocated coding sessions. The coders have experience with code reviews both in research and commercial software development settings (acting as both patch authors and reviewers). In total, these coding sessions took 56 hours (or 112 person-hours).

When coding links, the coders focused on the key reasons why the link was recorded. For example, Figure 12 shows that comments from Reviewer 1 on Nova

Reviewer 1
Patch Set 3:
@Author
We faced this kind of problems, and we can change Tempest code before this like:

```
 -      self.assertRaises(exceptions.NotFound,
 +      self.assertRaises((exceptions.NotFound, exceptions.BadRequest),
                          self.client.reserve_fixed_ip,
                          "my.invalid.ip", body)
```

Author
Patch Set 3:
@Reviewer 1
thanks, I'v post a patch to tempest at
https://review.openstack.org/#/c/127457/
let's wait for it merged first.

Figure 12: An example of a review link from Review #126831 in Nova.

Review #126831 have inspired the author to create Review #127457. We also record the direction of the link, e.g., Review #126831 links to Review #127457.

In theory, multiple codes may apply to each link; yet in practice, we find that multi-coded links are rare. Indeed, while a link may have different codes if interpreted in different directions, we do not find any multi-coded directional links.

Since open coding is an exploratory data analysis technique, it may be prone to errors. To mitigate errors, we code in three passes. First, since codes that emerge late in the process may apply to earlier reviews, after completing an initial round of coding, we perform a second pass over all of the links to correct miscoded entries. During the first coding pass, we code only using the link source (description/comment). In several cases, more contextual information was needed. We coded such cases as “Needs Additional Context” during the first coding pass. During a third coding pass, we check additional sources of information (e.g., the content of the patch, the linked review, comments in discussion threads) to code these cases more specifically. After the three coding passes, all of the sampled links have been assigned to a specific code.

Card Sorting. Similar to prior studies [4,30,46,81], we apply open card sorting to construct a taxonomy of codes. This taxonomy helps us to extrapolate general themes from our detailed coded data. The card sorting process is comprised of

two steps. First, the coded links are merged into cohesive groups that can be represented by a similar subgraph. Second, the related subgraphs are merged to form categories that can be summarized by a short title.

Table 16: The frequency of the discovered types of review linkage in OpenStack Nova and Neutron.

Category	Frequency	
	Nova	Neutron
C1: Patch Dependency	269 (55%)	148 (56%)
Patch Ordering	124 (26%)	62 (24%)
Root Cause	50 (11%)	28 (11%)
Shallow Fix	6 (2%)	4 (2%)
Follow-up	28 (6%)	29 (11%)
Merge Related Reviews	19 (4%)	13 (5%)
Multi-part	42 (9%)	12 (5%)
C2: Broader Context	96 (20%)	50 (19%)
Related Feedback	43 (9%)	14 (6%)
Demonstration	29 (6%)	26 (10%)
Additional Evidence	24 (5%)	10 (4%)
C3: Alternative Solution	69 (14%)	39 (15%)
Superseding	35 (8%)	17 (7%)
Duplicated	34 (7%)	22 (9%)
C4: Version Control Issues	27 (6%)	17 (6%)
Integration Concern	15 (4%)	13 (5%)
Gerrit Misuse	5 (2%)	2 (1%)
Revert	7 (2%)	2 (1%)
C5: Feedback Related	23 (5%)	10 (4%)
Fix Related Issues	11 (3%)	3 (2%)
Feedback Inspired Reviews	12 (3%)	7 (3%)

5.4.2 Results

Table 16 provides an overview of the categories that summarize related labels (the complete table is available online[25]). We observe that the frequencies at which the link labels appear are consistent between the two studied projects. Moreover, we only coded two of 486 links from Nova and two of 266 links from Neutron as false positives (i.e., spuriously detected links that do not indicate

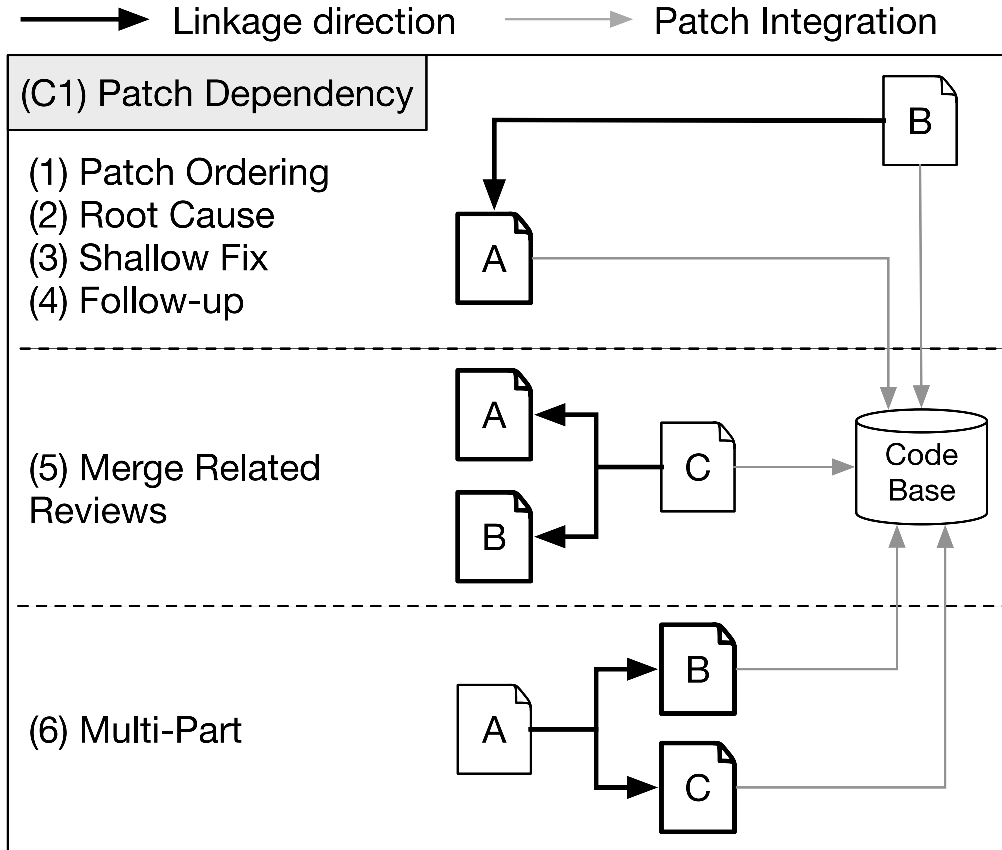

Figure 13: The Patch Dependency subgraphs.

a relationship between reviews), suggesting that our link extraction approach does not produce much noise (precision > 0.99 in both cases). Furthermore, we required additional context information (beyond the link source) to code 63 links, all of which were more specifically coded during the third pass when we analyze additional information sources. Below, we describe the discovered codes according to the categories to which they belong.

Patch Dependency (C1). We find that 55% and 56% of the analyzed links in Nova and Neutron connect reviews to others that they depend upon. Patch Dependency links may influence integration decisions and the reviewers who should be recommended. Indeed, the integration decision in one review may be inherently linked to that of another if they share a dependency. For example, Review

#102704[26] of the Nova project was only abandoned because of its dependency on Review #102705, which was abandoned earlier. Moreover, reviewers of a dependent review may need to review its dependency as well. For example, a reviewer of Review #102749[29] was added only because they reviewed its dependency (Review #101424). We further explore the usefulness of these linkage-based reviewer invitations in Section 5.6 (RQ4).

Figure 13 shows three shapes that patch dependency links take. First, Patch Ordering, Root Cause, Shallow Fix, and Follow-up take the shape of two eventually integrated (or abandoned) reviews that share a link. While they share a shape, the semantics of the patterns differ, i.e., Patch Ordering links indicate a timing dependency that must be respected at integration time, while Follow-up, Root Cause, and Shallow Fix links provide rationale for Review B by pointing to enabling enhancements or limitations in Review A. Second, Merge Related Reviews links merge two or more reviews into a more cohesive whole. Finally, Multi-part links indicate that a large review has been split into a series of smaller reviews.

Weißgerber et al. [92] observed that smaller patches tend to be accepted in two large open source projects. Rigby et al. [73] argue that one of the statutes of an efficient and effective code review process is the "early, frequent review of small, independent, complete solutions". The frequency of the Multi-part pattern (i.e., the splitting of large patches into smaller ones) may be an indication that these prior observations still hold.

Broader Context (C2). We find that 20% and 19% of the analyzed links point to other reviews with relevant resources. The individual analysis of reviews that are connected with Broader Context links may not be valid. Indeed, analyses of review outcome prediction often compute the length of discussion threads [28,40]. However, a discussion may span across several reviews when Broader Context links are present. For instance, a reviewer of Review #155223[30] asks the author to refer to a similar discussion on Review #215608.

Figure 14 shows that our three codes within the Broader Context category share the same shape; however, the codes differ in the artifact to which they refer.

[29] https://review.openstack.org/#/c/102749/
[30] https://review.openstack.org/#/c/155223/

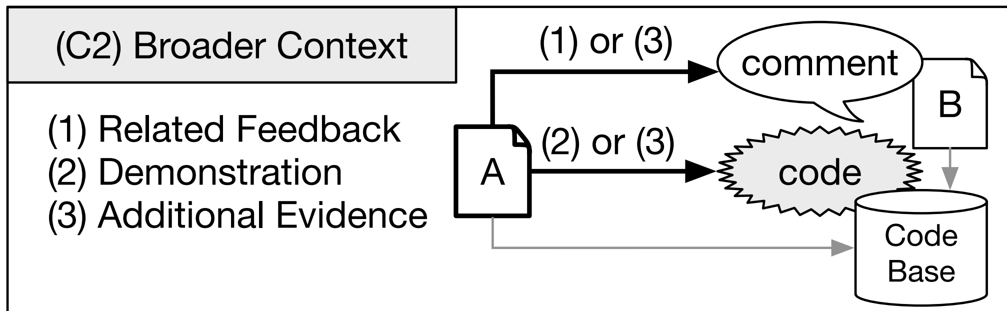

Figure 14: The Broader Context subgraph.

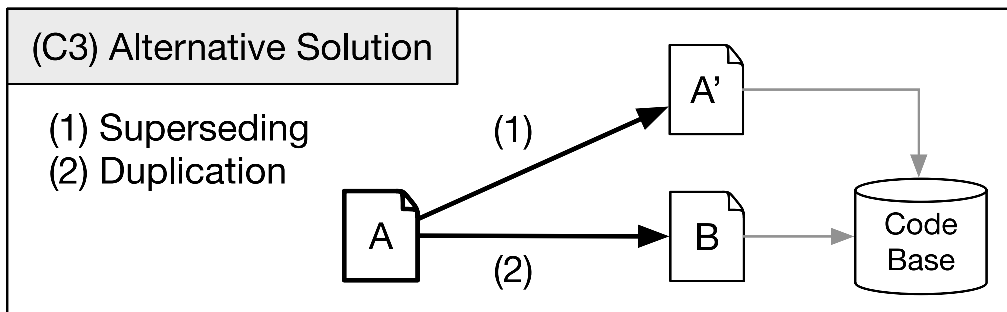

Figure 15: The Alternative Solution subgraph.

Related Feedback links connect discussions on one review to discussions in other reviews, while Demonstration links point to example code from other reviews. Additional Evidence links point to other reviews as proof (code, discussions, specifications) of the existence, removal, or relevance of the problems that are addressed by the review under inspection.

Alternative Solution (C3). We find that 14% and 15% of the analyzed links connect reviews to others that implement similar functionality. Similar to Patch Dependency links, Alternative Solution links may also impact integration decisions and reviewer recommendations. For example, Review #67431[31] was abandoned because another submitted solution for the same underlying issue (Review #61041) was preferred. Especially in such examples where an "either or" de-

[31] `https://review.openstack.org/#/c/67431/`

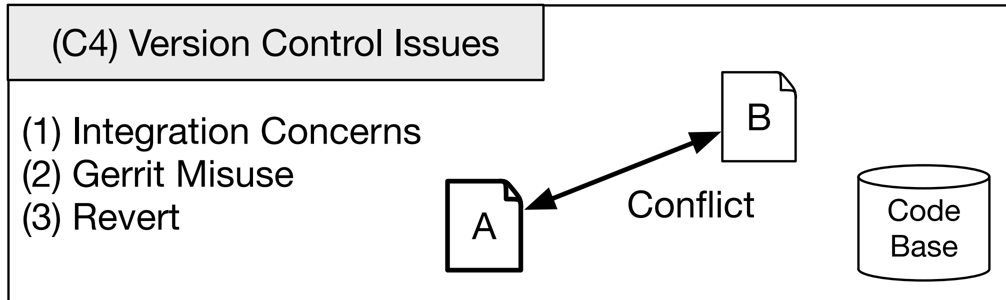

Figure 16: The Version Control Issues subgraph.

cision needs to be made, the same reviewers should likely be invited to all of the competing reviews for the sake of fairness [26]. Furthermore, prior work has demonstrated that a lack of awareness of concurrently developed solutions may result in redundant work [16, 101] and is a key source of software development waste [77]. These conflated integration decisions are not congruent with review outcome or reviewer recommendation models that assume each submission is independently adjudicated [34, 39, 40].

Figure 15 shows that our two codes within the Alternative Solution category share the same shape, yet differ in their semantics. Superseding links show that the solution in an earlier review has been replaced with an updated solution in the current review, while Duplication links highlight the existence of another (competing) solution to the same underlying problem. In a large-scale, cross-company software organization like OpenStack, it is difficult to coordinate development effort. However, the frequency at which work is duplicated suggests that tooling [16, 101] may help.

Version Control Issues (C4). We find that 6% of analyzed links point to reviews that introduced version control issues. Rigby and Storey [74] also found such issues are often discussed during the broadcast-based reviews in several open source systems. Shimagaki et al. [80] found that 5% of commits in a large industrial system were reverted after being integrated. Since Revert is one of the codes within our category, our review graphs can complement version control data to better understand the practice of reverting commits.

Figure 16 shows that our three codes within the Version Control Issues cate-

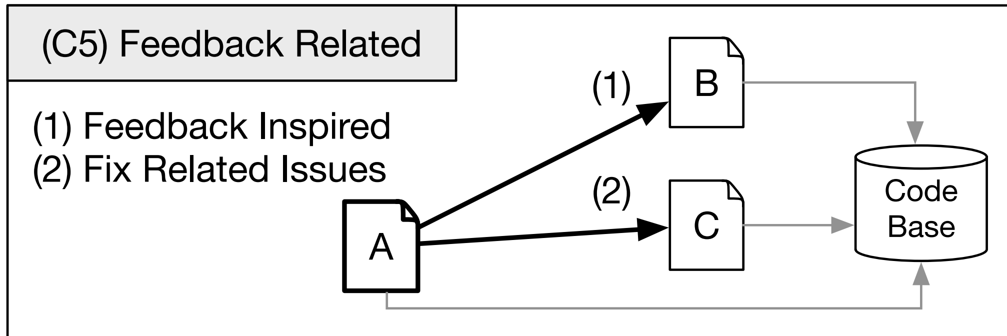

Figure 17: The Feedback Related subgraph.

gory share the same shape. Integration Conflict and Gerrit Misuse links expose technical integration or Gerrit issues, while Revert links indicate that a partial or complete rollback.

Feedback Related (C5). We find that 5% and 4% of analyzed links in NOVA and NEUTRON connect reviews to others that resolve or were inspired by reviewer comments. Reviews that were inspired by feedback in another review might be more likely to be accepted, since one reviewer is already in favour of the idea. For example, in Review #167100,[32] a reviewer's feedback inspired the creation of the new Review #167082. Then, one of reviewers of #167100 joined Review #167082, and eventually approves it for integration. There are two possible ways to act upon C5-linked reviews. The reviewer who inspired the change may be well suited to review the inspired change. Thus, reviewer recommenders may need to recommend them. On the other hand, since the reviewer who inspired the change may not be impartial when reviewing the inspired review, reviewer recommenders may need to recommend other reviewers.

Figure 17 shows that our two codes within the Feedback Related category share the same shape. Feedback-inspired links show a new contribution where feedback on Review A inspires the creation of a new patch. Fixed Related Issues links show that a raised concern has been addressed by another review.

RQ2: A broad variety of reasons for linkage exist. These different links may introduce noise in or opportunities for improvement of code review analytics.

[32]`https://review.openstack.org/#/c/167100/`

5.5 Automated Link Classification (RQ3)

In Section 5.4, we find that several types of links may impact reviewer recommendation and outcome prediction. Since different review analytics techniques may need to traverse or ignore links depending on the type, a more scalable approach to link type recovery is needed. Indeed, it took the authors 112 person-hours to code 752 review links (see Coding in Section 5.4.1). If we continue to code at this rate, it would take an additional 19,793 person-hours to code the remaining 132,898 reviews in the OpenStack data set.

In this section, we study the feasibility of using machine learning techniques to automatically classify links by categories. To do so, we use the manually coded data from Section 5.4 as a sample on which to train and evaluate classifiers that identify the link category (C1–C5) based on the document that it appears within (i.e., the review description field or the comment in the general or inline discussion thread). Below, we describe our approach to automated link category classification (5.5.1) followed by the results (5.5.2).

5.5.1 Classification Approach

Feature Extraction. We apply standard text preprocessing techniques to lessen the impact of noise on our classifiers. We first tokenize the document and remove stop words using the Python NLTK stop word list. Next, we apply lemmatization to handle term conjugation using the Python NLTK `lemmatize` function. Finally, we convert each sampled description or comment to a vector of the Term Frequency-Inverse Document Frequency (TF-IDF) weights of its terms. Broadly speaking, terms that appear rarely across documents, and/or often within one document are of higher weight. We use the Python Scikit-Learn `TfidfVectorizer` function to compute TF-IDF scores for all documents in a training sample.

Classifier Validation Technique. To estimate classifier performance on unseen data, we apply the out-of-sample bootstrap validation technique [22], which tends to yield more robust results than other validation techniques (e.g., k-fold cross

validation) [82]. First, a bootstrap sample of size N is randomly drawn with replacement from the original sample of the same size N. This bootstrap sample is used to train our classifiers, while the documents from the original sample that do not appear in the bootstrap sample are set aside for testing. Since the bootstrap sample is selected with replacement, on average, 36.8% of the documents will not appear in the bootstrap sample and can be used to evaluate classifier performance. We perform 1,000 iterations of the bootstrap procedure (reporting the mean performance scores across these iterations) to ensure that our performance measurements are robust.

Classification Techniques. To train our classifiers, we experiment with a broad selection of popular classification techniques. Support Vector Machines (SVM) use a hyperplane to classify documents by first transforming feature values into a multidimensional feature space. Random Forest is an ensemble learning technique that builds a large number of decision trees, each using a subset of the features, and then aggregates the results from each tree to classify documents. Multinomial Naïve Bayes (MNB) is a conditional probability model that uses a multinomial distribution for each of the features. Multi-Layer Perceptron (MLP) is a supervised learning technique where weighted inputs are delivered through neurons in sequential layers. Multinomial Logistic Regression (MLR) generalizes the logistic regression technique to the multi-class classification setting. We use the Python Scikit-Learn implementations of the classification techniques (`svm.SVC`, `RandomForestClassifier`, `MultinomialNB`, `MLPClassifier`, and `LogisticRegression`).

Hyperparameter Optimization. The classification techniques that we use have configurable parameters that impact their performance. Similar to prior work [82], we use a grid search to tune the parameter settings. Grid search is an exhaustive searching technique that examines all of the combinations of a specified set of candidate settings to find the best combination. We explore the same set of candidate settings as Tantithamthavorn et al. [82, p. 5]. We search for the optimal parameter settings for each classification technique in each bootstrap sample (i.e., without using the testing data) using the Scikit-Learn `GridSearchCV` function.

Performance Evaluation. To evaluate our classifiers, we use common perfor-

Table 17: The performance of our five link category classifiers, all of which outperform the ZeroR and random guessing baselines in all cases.

Model	Nova				Neutron			
	Prec.	Rec.	F1.	AUC	Prec.	Rec.	F1.	AUC
SVM	0.75	0.88	0.79	0.72	0.72	0.92	0.77	0.82
RF	0.77	0.72	0.71	0.57	0.76	0.84	0.72	0.89
MNB	0.71	0.80	0.74	0.72	0.71	0.82	0.75	0.76
MLP	0.74	0.81	0.76	0.61	0.75	0.84	0.74	0.68
MLR	0.75	0.88	0.79	0.65	0.72	0.92	0.77	0.79
ZeroR	0.26	0.50	0.34	–	0.27	0.50	0.34	–

mance measures. Precision is the proportion of links that are classified as a given category that are correct. Recall is the proportion of links of a given category that a classifier can detect. The F1-score is the harmonic mean of precision and recall. The Area Under the Curve (AUC) computes the area under the curve that plots the true positive rate against the false positive rate as the threshold that is used for classifying documents varies. AUC ranges from 0 to 1, where random guessing achieves an AUC of 0.5.

Since our links have more than two categories, we need to use multi-class generalizations of these performance measures. Each measure is computed for each category before being aggregated into an overall score. Since the link categories are imbalanced (see Table 16), we weigh the category scores by their overall proportion.

We also compare our classifiers to a ZeroR classifier, which always reports the most frequently occurring class. In our setting, a ZeroR classifier achieves a recall of one and a precision equal to the frequency of the most frequently occurring category (C1) for that class, and a precision and recall of zero for the other categories. Note that AUC does not apply to ZeroR classifiers because likelihood estimates are not produced. We use the Scikit-Learn `metrics` library to compute our performance measurements.

Table 18: The mean performance scores of cHRev [100] and our proposed link-aware reviewer recommenders at different prediction delays. The bulk of the performance improvement is achieved in the no delay (0 hour) setting.

	Nova									Neutron								
	Top 1			Top 3			Top 5			Top 1			Top 3			Top 5		
Model	Pre.	Rec.	F1.	Pre.	Rec.	F1.	Pre.	Rec.	F1.	Pre.	Rec.	F1.	Pre.	Rec.	F1.	Pre.	Rec.	F1.
Baseline	0.25	0.06	0.09	0.28	0.18	0.22	0.30	0.32	0.30	0.24	0.05	0.08	0.26	0.17	0.20	0.28	0.31	0.29
0 hour	0.39	0.09	0.14	0.40	0.26	0.31	0.40	0.43	0.41	0.39	0.09	0.14	0.40	0.26	0.31	0.41	0.44	0.42
1 hour	0.39	0.09	0.14	0.40	0.26	0.31	0.40	0.43	0.41	0.41	0.09	0.14	0.41	0.26	0.32	0.42	0.44	0.43
3 hours	0.40	0.09	0.14	0.40	0.26	0.31	0.40	0.43	0.41	0.41	0.09	0.14	0.41	0.26	0.32	0.42	0.45	0.43
6 hours	0.40	0.09	0.14	0.40	0.26	0.31	0.40	0.43	0.41	0.41	0.09	0.15	0.41	0.27	0.32	0.42	0.45	0.43

5.5.2 Results

Table 17 shows that while no classification technique consistently outperforms the others, the classifiers achieve a precision of 0.71–0.77, a recall of 0.72–0.92, and F1-scores of 0.71–0.79. Since these performance scores are on par with those of prior classification studies [51,67,69], we believe that our classifiers show promise. Moreover, Table 17 shows that our classifiers outperform baseline approaches, achieving precision, recall, and F1-scores that are 22–51 percentage points better than the ZeroR baseline and AUC values above the 0.5 random guessing benchmark.

RQ3: Despite the complexity of the five-class classification problem, our classifiers achieve precision, recall, and F1-scores that exceed the ZeroR baseline by 22–51 percentage points.

5.6 Linkage Impact Analysis (RQ4)

Our analysis in RQ2 suggests that linkage can impact code review analytics. In this section, we measure that impact on reviewer recommendation (5.6.1) and review outcome prediction (5.6.2).

5.6.1 Reviewer Recommendation

Approach. In Section 5.4, we observe that Patch Dependency links (C1), Alternative Solution links (C3), and Feedback Related links (C5) may impact reviewer

recommendation because reviewers of a linking review may need to review its linked review.

To measure the degree to which reviewers contribute to both linking and linked reviews, we compute the Overlapping Reviewer Rate (ORR), i.e., the proportion of reviewers from the linking review who also review the linked review. We compute the ORR rates on our sampled links from Section 5.4.

To study the extent to which state-of-the-art reviewer recommenders identify overlapping reviewers, we apply cHRev [100], which makes recommendations based on prior contribution and working habits. We approximate data sets where C1, C3, and C5-linked reviews are known by applying our top-performing classifier (SVM) from Section 5.5 to the linked Nova and Neutron reviews. We exclude C5-linked reviews from further analysis because we detect too few instances (i.e., five) to draw meaningful conclusions.

We then compare the performance of cHRev to an extended version that ranks reviewers of linking reviews at the top of the list for linked reviews. Since links appear as reviews evolve, we select only those links that are available at prediction times zero, one, three, and six hours after the review has been created. Moreover, we only identify candidate overlapping reviewers who have commented on linking reviews at or before those prediction time settings.

Results. The ORR rates of C1 and C3 are 50%–51% and 65%–77%, respectively. By way of comparison, we find that the ORR rate is less than 1% for randomly selected pairs of non-linked reviews. The results indicate that reviewers are more likely to participate in both reviews when links are present.

A closer inspection reveals that cHRev misses at least one overlapping reviewer in 96%–100% of linked reviews across Top 1–5 recommendation lists. Since medians of 12 and 15 reviewers participate in Nova and Neutronreviews, respectively, missing at least one overlapping reviewer is a concern. Moreover, none of the overlapping reviewers are recommended in 41%–81% of Nova and 48%–84% of Neutron reviews. When overlapping reviewers are omitted, they appear in 13th–15th place on average. Increasing the weight of overlapping reviewers may improve recommendations.

Table 18 shows that the precision, recall, and F1-scores of cHRev improve by 33%–88% (3–17 percentage points) if reviewers of a linked review are recom-

mended at the top of the list. Moreover, the degree of the improvement remains consistent across the four studied prediction time settings, indicating that no prediction delay is necessary to gain the bulk of the value. Longer delays may achieve better results but would be less useful in practice, since waiting more than six hours for reviewer recommendations is impractical.

5.6.2 Review Outcome

Approach. In Section 5.4, we report that C1, C2 (Broader Context), and C3 links may impact review outcome prediction because the integration decision in one review may be inherently linked to that of the other. Similar to the reviewer recommendation experiment above, we apply our SVM classifier to identify C1, C2, and C3 links in the full Nova and Neutron data sets.

For C1, C2, and C3 reviews, we compute the Identical Outcome Rate (IOR), i.e., the rate at which linking and linked reviews result in the same outcome (i.e., integration or abandonment). Moreover, to study the extent to which state-of-the-art outcome predictors mispredict identical outcomes, we apply the outcome prediction approach of Gousios et al. [28]. More specifically, we train prediction models that classify reviews as integrated or abandoned based on review properties (e.g., # comments, # participants). In our setting, we train the predictors on unlabeled linked reviews and evaluate them on our labeled samples.

Results. The IOR rates of intergrated C1 and C2-linked reviews are 73%–87% in Nova and 55%–71% in Neutron, while the IOR rates for abandonment are 57%–86% and 45%–75%, respectively. This suggests that reviews that are connected with C1 and C2 links tend to have the same outcome. Since C3 links connect competing solutions, it is unlikely that they will have the same outcome. This is reflected in lower IOR rates of 18%–26% for integration, respectively. On the other hand, the IOR rates for abandonment are 46% in Nova and 62% in Neutron, indicating that it is not uncommon for both competing solutions to be abandoned.

Outcome predictors may misclassify review outcomes for linked reviews when the feature values span multiple reviews (e.g., discussion contexts, # participants). Indeed, we find that prior approach misclassifies 35% and 39% of C1, C2, and C3-linked reviews in Nova and Neutron, respectively.

RQ4: Reviewer recommenders tend to omit or poorly rank reviewers who participate in both linking and linked reviews. Moreover, review outcome predictors tend to misclassify linked review outcomes. Leveraging links that are available at prediction time can yield considerable performance improvements.

5.7 Practical Suggestions

Linkage should be taken into consideration in code review analytics. Our quantitative study (Section 5.3) shows that up to 25% of reviews in our subject communities link to at least one other review. Moreover, in the four studied communities that have made the largest investment in reviewing (OpenStack, Chromium, AOSP, and Qt), we observe increasing or stable trends in the monthly linkage rate. Prior work has shown that linked artifacts can impact repository mining analytics [13, 33]. Indeed, our impact analysis (Section 5.6) shows that reviewer recommenders and outcome predictors can be improved by taking linkage into account.

Duplicate detection would enhance review efficiency. Our qualitative study shows that 14%–15% of congruent links in Nova and Neutron fall in the Alternative Solution category (i.e., superseding and duplication). Duplicate contributions are a form of waste in software development [77]. Boisselle and Adams [13] argued that automatic classification of bug reports that are linked due to duplication would be helpful. However, in large projects like OpenStack and Chromium, it is difficult to keep track of duplicated work [101]. Systematic detection of duplicates would be important from a review fairness [26] perspective as well, since the competing solutions should be subject to the same level of scrutiny.

Link category recovery may not be necessary. The link types that we identified potentially impacting reviewer recommendation (C1, C3, and C5) and outcome prediction (C1, C2, and C3) in Section 5.6 account for the vast majority of links. Since we found that all of the link categories are consistently useful, it may be possible to assume that all links are useful and omit link type classification.

5.8 Threats To Validity

Construct validity. Construct validity is concerned with the degree to which our measurements capture what we aim to study.

We recover links between reviews using regular expressions that detect Change IDs and URLs of other reviews. However, these regular expressions may extract Change IDs or URLs that are not intended to be links (false positives). In Section 5.4, we find that the false positive rate in our manually analyzed sample is quite low (<0.1%). Thus, we do not believe that false positives are significantly impacting our measurements.

In our qualitative analysis, links may be miscoded due to the subjective nature of our open coding approach. We take several precautions to mitigate the miscoding threat. First, the code for each link is agreed upon by two coders who have experience with code review in academic and commercial settings. Furthermore, we employ a three-pass approach, where each code is revisited at least once to ensure that the correct code was selected.

Internal validity. Internal validity is concerned with our ability to draw conclusions from the relationship between study variables.

Links may not be detected in all of the cases when they should be, which may introduce noise in our linkage rate observations in Section 5.3. Hence, our observations in Section 5.3 should be interpreted as lower bounds rather than as exact linkage rate values.

If we stop coding too early during our qualitative analysis (Section 5.4), it may threaten the completeness of the discovered set of link types. To mitigate the threat, we continue to code until our samples saturate—a concept that we operationalized by continuing until we coded a span of 50 coded links without discovering any new codes. Others have used similar saturation criteria [74, 99].

Other classification techniques or hyperparameter settings may yield better results than the ones that we studied in Section 5.5. To combat this, we select a broad set of popular classification techniques and use an automatic parameter optimization approach to select the best configuration of hyperparameter settings for each bootstrap iteration. Nonetheless, exploration of other classification techniques and hyperparameter settings may yield better results.

External validity. External validity is concerned with our ability to generalize

based on our results.

We extract review linkage graphs (Section 5.3) from six subject communities that use the Gerrit code review tool. Due to the manually intensive nature of our link coding approach, we focus on an in-depth analysis of the two largest projects from the OpenStack community (Section 5.4–5.6). As such, our linkage types, classifiers, and impact analyses may not generalize to code reviewing environments in all software communities. Replication studies may be needed to arrive at more general conclusions. To simplify replication, we have made our scripts and data publicly available.[25]

5.9 Conclusion

Researchers have recently proposed several analytics-based techniques to support stakeholders in the MCR process. However, those techniques have tacitly or explicitly treated each review as an independent observation, which overlooks relationships among reviews.

Our empirical study suggests that linkage is not uncommon in six studied software communities. Moreover, adding linkage awareness to review analytics approaches yields considerable performance improvements. Thus, review linkage should be taken into consideration in future MCR studies and tools.

6. Conclusions & Future Work

Code review plays an important role in software quality assurance. In recent software developments, the Modern Code Review has been practiced. Unlike traditional inspection, the modern code review provides the informal and asynchronous reviewing style. The openness of the MCR process can encourage any developer to participate in reviews and the divergence leads to different thoughts.

While prior studies have performed empirical studies to analyze the impacts of the divergence and openness on code review resolutions, a fine-grained study at a comment level is still needed to have a deep understanding of how reviews are resolved as developers provide comments. This thesis provides empirical studies in three dimensions: (i) reviewer discussion, (ii) review participation, and (iii) code review analytics.

6.1 Contributions

- **Reviewer Discussion**— We find that patches with divergent scores are not rare in OpenStack and Qt, with 15%–37% of patches that receive multiple review scores having divergent scores. Patches with divergent scores that are eventually integrated tend to involve one or two more reviewers than patches without divergent scores. Patches that are eventually abandoned with strong divergent scores more often suffer from external issues than patches with weak divergent and without divergent scores.

  Suggestion. Software organizations should be aware of the potential for divergent review scores. Moreover, external dependencies mostly cause divergent scores among reviewers, suggesting that automation for notifying authors about external dependencies before their submissions would be helpful to prevent unnecessary dependencies from occurring.

- **Review Participation**— We find that there are no magic number of comments and number of words to resolve review processes. Our in-depth analysis also finds that the number of comments and number of words fluctuate over time. Our empirical study reports that human experience and patch property features have the strongest impact on general and inline comments

and their words, respectively.

Suggestion. Experienced reviewers should be invited when rich participation is needed. Moreover, the larger patch churn is, the more likely that inline comments increase.

- **Code Review Analytics**— Reviews that are linked to others have increased over time. Moreover, there are five linkage categories that comprise of 16 linkage patterns, one of which is duplicate work. Therefore, duplicate detection would enhance review efficiency. Our impact analysis shows that linkage information has the potential to improve code review analytics.

  Suggestion. Review linakge information should be taken into consideration for improving code review analytics.

Our fine-grained studies provide evidence-based results to understand the mechanism of code review resolutions at a comment level.

6.2 Opportunities for Future Work

This thesis brought a lot of practical implications to code review resolutions and an improvement to code review analytics. We believe that those contributions can bring opportunities for future research and open up new research fields. We outline those opportunities below.

6.2.1 Improving reviewer guideline based on practical implications at a comment level

Section 3 showed that external dependencies have the potential to lead to divergent discussions. However, it is still difficult for novice reviewers and authors to catch the dependency issues before they start with reviews. Our practical implications motivate the software communities to update the basic guideline for contributers by taking into consideration the fact that external dependencies may cause too divergent discussions which might be unnecessary. Indeed, lazy consensus concept (i.e., a reviewer avoids joining a discussion when other reviewers raise concerns) is practiced in the Modern Code Reviews.[33]

[33] `https://community.apache.org/committers/lazyConsensus.html`

Section 4 suggested that human factors can impact review participation until a review ends. Our findings complement prior work that investigates the impact of human factors at the initial review participation [75,87]. However, when taking a deeper look at inline comments, we find that patch property factors are more impactful than human factors. Reviewer guidelines should take the practical implications into consideration to enrich those guidelines, which would generalize the impact of this study and influence team transparency (see Figure 1). One of the potential implications that we can suggest for those guidelines would be that the larger the patch churn is, the more likely that participants would discuss code-related issues with inline comments and the discussion point will be specific. Nonetheless, this implication would influence knowledge transfer (see Figure 1) since developers can share their detailed thoughts through inline commenting.

6.2.2 Integrating linkage approaches into code review tools

Our Section 5 showed the impact of linkage information on code review analytics (i.e., reviewer recommendation and outcome prediction). Although there have been many code review tools in code review research field, little is known about the potential impact of linkage approaches on other review tools. We believe that our linkage extraction approach can help researchers to enrich their review datasets. This also helps developers be aware of potential linkage among review artifacts, which would impact team transparency (see Figure 1). Furthermore, the approach in which we integrate linkage information into recommendation and prediction could be applied to other analytics (e.g., review time estimation). A review may be split into multiple reviews due to the large amount of related changes. In that case, the context of those multiple reviews should be treated as one sequential review, which would impact context understanding (see Figure 1).

6.2.3 Opening up new opportunities for code review studies

We believe that there are opportunities for dependency-related studies in code review research. Based on Section 3 and 5, we are able to understand how reviews are linked to other reviews and how their linkage impacts code review resolutions. Predicting the complexity of review artifacts would be helpful for practitioners to eliminate the risk of dependencies before developers submit their reviews.

The review complexity prediction should predict the degree to which a review is complicated due to external dependencies. Moreover, eliminating the risk of complex dependencies can also impact code review quality, which would impact code improvement (see Figure 1). Indeed, related work [52] to this study showed that linked issues ended up the cause of delaying the release cycle and increasing maintenance costs.

Acknowledgements

To achieve the graduation for my Ph.D. program, I have had so much support from professors, friends and my family. Here is my acknowledgement for them.

First of all, thanks to Professor Kenichi Matsumoto for supervising me in this Ph.D. program. He has taken care of my degrees since I was in my master's program. I would not have been able to achieve this graduation without his support and guidance for the thesis.

I would like to thank Professor Hajimu Iida for co-supervising my thesis and giving lots of support to my work. His feedback that I have gotten during the Ph.D. program was helpful to structure the thesis and it strengthened my thesis overall. I really appreciate his help and thoughtful advice.

I am grateful to Professor Takashi Ishio that he co-worked with me in industrial research projects. Without his support, the thesis and those projects would not have been possible. I would like to express my gratitude to his support in writing academic papers and conducting research.

I feel lucky that I started working with Professor Raula Gaikovina Kula since the first year of my Ph.D. program. He has taken care of my studies and given lots of support to me. In addition, he taught me lots of things that improve my international communication skill. Moreover, with regard to the support in my research, he and Dr. Christoph Treude supported IWESEP2019 symposium in which I was a co-general chair.

Special thanks to Professor Shane McIntosh! Without his support I would not have been able to achieve this thesis. He accommodated me to McGill University (Québec, Canada) when I was in my master's and Ph.d programs. I visited him and worked on our research projects for around twelve months in total during the programs. His supervision for research projects in Canada helped me a lot. Once again, thank you for your kind support and participating in this committee!

Professor Akinori Ihara has supported my work since I was in my master's program. I could not have had an opportunity to go to Canada and other countries to do research without his support. In IWESEP2019, he supported our work as a registration co-chair. I really appreciate his support a lot!

I would like to thank Professor Hideaki Hata and Noriko Takagishi. Professor Hideaki Hata gave me a lot of suggestions about my research work and supported

me in many ways. Noriko Takagishi helped me out when I worked on documents that were related to my research work. I would like to appreciate their help!

During my stay at ABB Corporate Research in North Carolina, I have gotten a lot of great experiences through working with David Shepherd, Andrew Cordes, Nicholas A. Kraft, Patrick Francis, and Mauricio Soto. I appreciate their great help. It was one of my luckiest things in my life that I met Jerome J. Sanders and Alberto Moreno in North Carolina. I am really respectful of them!

I have had a privilege of being with my friends in software engineering lab and others! I would like to thank all of my lab-mates in software engineering lab of NAIST. I would not have been able to have great research experiences without their support. Special thanks to Sultan Wehaibi for working with me and sharing a lot of experiences with me! It was remarkable time in my life to have many great discussions with you! During my stay at The Software REBELs lab, Keheliya Gallaba, Ruiyin Wen, Farida El Zanaty, and Christophe Rezk helped me out in research and my life. Special thanks to my lab-mates for having me in The Software REBELs lab group! Moreover, I enjoyed the research life with my great friends, Márton Búr, Oscar Delgado, El-Nasser Youssef, Rania Gamal, Emily Jonas, and Dong Geun Kim! I would like to thank them as well. I am really grateful for all of my friends to give me wonderful experiences.

I would like to show my appreciation to Masahiro Desaki for helping me out and working with me in many ways. I am always respectful of what you have worked on and achieved!

Finally, without my family's support, the thesis would not have been possible. Since they supported my things so much during my Ph.d program, I got ready enough for what I would like to start after my graduation! Thank you so much for lots of help and being patient for my graduation.

References

[1] Sultan S. Alqahtani, Ellis E. Eghan, and Juergen Rilling. Recovering semantic traceability links between apis and security vulnerabilities: An ontological modeling approach. In Proceedings of International Conference on Software Testing, Verification and Validation, pages 80–91, March 2017.

[2] Giuliano Antoniol, Gerardo Canfora, Gerardo Casazza, Andrea De Lucia, and Ettore Merlo. Recovering traceability links between code and documentation. In IEEE Transactions on Software Engineering, volume 28, pages 970–983, 2002.

[3] Aybuke Aurum, Håkan Petersson, and Claes Wohlin. State-of-the-art: software inspections after 25 years. Software Testing, Verification and Reliability, 12(3):133–154, 2002.

[4] Alberto Bacchelli and Christian Bird. Expectations, outcomes, and challenges of modern code review. In Proceedings of the 35th International Conference on Software Engineering, pages 712–721, 2013.

[5] Alberto Bacchelli, Michele Lanza, and Romain Robbes. Linking e-mails and source code artifacts. In Proceedings of the 32nd International Conference on Software Engineering, pages 375–384, 2010.

[6] Vipin Balachandran. Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation. In Proceedings of the 35th International Conference on Software Engineering, pages 931–940, 2013.

[7] Tobias Baum, Olga Liskin, Kai Nikla, and Kurt Schneider. Factors influencing code review processes in industry. In Proceedings of the 24th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, pages 85–96, November 2016.

[8] Olga Baysal, Oleksii Kononenko, Reid Holmes, and Michael W. Godfrey.

The secret life of patches: A firefox case study. In Proceedings of the 19th Working Conference on Reverse Engineering, pages 447–455, 2012.

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

[10] Olga Baysal, Oleksii Kononenko, Reid Holmes, and Michael W. Godfrey. Investigating technical and non-technical factors influencing modern code review. Empirical Software Engineering, 21(3):932–959, 2016.

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

[12] Christian Bird, Alex Gourley, and Prem Devanbu. Detecting patch submission and acceptance in oss projects. In Proceedings of the Fourth International Workshop on Mining Software Repositories, pages 26–29, 2007.

[13] Vincent Boisselle and Bram Adams. The impact of cross-distribution bug duplicates, empirical study on debian and ubuntu. In Proceedings of the 15th International Working Conference on Source Code Analysis and Manipulation, pages 131–140, 2015.

[14] Amiangshu Bosu, Jeffrey C. Carver, Munawar Hafiz, Patrick Hilley, and Derek Janni. Identifying the characteristics of vulnerable code changes: An empirical study. In Proceedings of the 22nd Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, FSE 2014, pages 257–268. ACM, 2014.

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

[16] Yuriy Brun, Reid Holmes, Michael D. Ernst, and David Notkin. Crystal: Precise and unobtrusive conflict warnings. In Proceedings of the 19th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, pages 444–447, 2011.

[17] Gerardo Canfora, Luigi Cerulo, Marta Cimitile, and Massimiliano Di Penta. Social interactions around cross-system bug fixings: The case of freebsd and openbsd. In Proceedings of the 8th Working Conference on Mining Software Repositories, pages 143–152, 2011.

[18] Kathy Charmaz. Constructing grounded theory. SAGE Publications, 2014.

[19] Jacek Czerwonka, Michaela Greiler, and Jack Tilford. Code reviews do not find bugs: How the current code review best practice slows us down. In Proceedings of the 37th International Conference on Software Engineering, pages 27–28, 2015.

[20] Jacek Czerwonka, Michaela Greiler, and Jack Tilford. Code reviews do not find bugs: How the current code review best practice slows us down. In Proceedings of the 37th International Conference on Software Engineering, pages 27–28, 2015.

[21] Felipe Ebert, Fernando Castor, Nicole Novielli, and Alexander Serebrenik. Confusion in code reviews: Reasons, impacts, and coping strategies. In Proceedings of the 26th International Conference on Software Analysis, Evolution and Reengineering, pages 49–60, 2019.

[22] Bradley Efron and Robert J. Tibshirani. An introduction to the bootstrap. Springer; Softcover reprint of the original 1st ed., 1993.

[23] Michael E. Fagan. Design and code inspections to reduce errors in program development. IBM Systems Journal, 15(3):182–211, 1976.

[24] Anna Filippova and Hichang Cho. The effects and antecedents of conflict in free and open source software development. In Proceedings of the 19th ACM Conference on Computer-Supported Cooperative Work & Social Computing, pages 705–716, 2016.

[25] Takafumi Fukushima, Yasutaka Kamei, Shane McIntosh, Kazuhiro Yamashita, and Naoyasu Ubayashi. An empirical study of just-in-time defect prediction using cross-project models. In Proceedings of the 11th Working Conference on Mining Software Repositories, pages 172–181, 2014.

[26] Daniel M. German, Gregorio Robles, Germán Poo-Caamaño, Xin Yang, Hajimu Iida, and Katsuro Inoue. “was my contribution fairly reviewed?”: A framework to study the perception of fairness in modern code reviews. In Proceedings of the 40th International Conference on Software Engineering, pages 523–534, 2018.

[27] G. Gousios, A. Zaidman, M. A. Storey, and A. v. Deursen. Work practices and challenges in pull-based development: The integrator’s perspective. In Proceedings of the 37th IEEE International Conference on Software Engineering, pages 358–368, 2015.

[28] Georgios Gousios, Martin Pinzger, and Arie van Deursen. An exploratory study of the pull-based software development model. In Proceedings of the 36th International Conference on Software Engineering, pages 345–355, 2014.

[29] Jin L. C. Guo, Jinghui Cheng, and Jane Cleland-Huang. Semantically enhanced software traceability using deep learning techniques. In Proceedings of the 39th International Conference on Software Engineering, pages 3–14, May 2017.

[30] Anja Guzzi, Alberto Bacchelli, Michele Lanza, Martin Pinzger, and Arie van Deursen. Communication in open source software development mailing lists. In Proceedings of the 10th Working Conference on Mining Software Repositories, pages 277–286, 2013.

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

[32] Ahmed E. Hassan. Predicting faults using the complexity of code changes. In Proceedings of the 31th International Conference on Software Engineering, pages 78–88, 2009.

[33] Hideaki Hata, Christoph Treude, Raula G. Kula, and Takashi Ishio. 9.6 million links in source code comments: Purpose, evolution, and decay. In Proceedings of the 41st International Conference on Software Engineering, pages 1211–1221, 2019.

[34] V. J. Hellendoorn, P. T. Devanbu, and A. Bacchelli. Will they like this? evaluating code contributions with language models. In 2015 IEEE/ACM 12th Working Conference on Mining Software Repositories, pages 157–167, May 2015.

[35] Toshiki Hirao, Akinori Ihara, and Kenichi Matsumoto. Pilot study of collective decision-making in the code review process. In Proceedings of the Center for Advanced Studies on Collaborative Research, pages 248–251, 2015.

[36] Toshiki Hirao, Akinori Ihara, Yuki Ueda, Passakorn Phannachitta, and Kenichi Matsumoto. The impact of a low level of agreement among reviewers in a code review process. In The 12th International Conference on Open Source Systems, pages 97–110, 2016.

[37] Wenjian Huang, Tun Lu, Haiyi Zhu, Guo Li, and Ning Gu. Effectiveness of conflict management strategies in peer review process of online collaboration projects. In Proceedings of the 19th ACM Conference on Computer-Supported Cooperative Work & Social Computing, pages 717–728, 2016.

[38] Franz J. Ingelfinger. Peer review in biomedical publication. The American Journal of Medicine, 56(5):686 – 692, 1974.

[39] Gaeul Jeong, Sunghun Kim, Thomas Zimmermann, and Kwangkeun Yi. Improving code review by predicting reviewers and acceptance of patches. Technical report, Reserach On Software Analysis for Error-free Computing, 2009.

[40] Yujuan Jiang, Bram Adams, and Daniel M. German. Will my patch make it? and how fast?: Case study on the linux kernel. In Proceedings of the 10th Working Conference on Mining Software Repositories, pages 101–110, 2013.

[41] Yujuan Jiang, Bram Adams, Foutse Khomh, and Daniel M. German. Tracing back the history of commits in low-tech reviewing environments. In Proceedings of the 8th International Symposium on Empirical Software Engineering and Measurement, pages 51:1–50:10, 2014.

[42] Jirayus Jiarpakdee, Chakkrit Tantithamthavorn, Akinori Ihara, and Kenichi Matsumoto. A study of redundant metrics in defect prediction datasets. In Proceedings of the International Symposium on Software Reliability Engineering, pages 51–52, 2016.

[43] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi. A large-scale empirical study of just-in-time quality assurance. IEEE Transactions on Software Engineering, 39(6):757–773, 2013.

[44] Yasutaka Kamei, Shinsuke Matsumoto, Akito Monden, Ken-ichi Matsumoto, Bram Adams, and Ahmed E. Hassan. Revisiting common bug prediction findings using effort-aware models. In Proceedings of the IEEE International Conference on Software Maintenance, pages 1–10, 2010.

[45] C. F. Kemerer and M. C. Paulk. The impact of design and code reviews on software quality: An empirical study based on psp data. IEEE Transactions on Software Engineering, 35(4):534–550, July 2009.

[46] Noureddine Kerzazi, Foutse Khomh, and Bram Adams. Why do automated builds break? an empirical study. In Proceedings of the 30th International Conference on Software Maintenance and Evolution, pages 41–50, 2014.

[47] Oleksii Kononenko, Olga Baysal, and Michael W. Godfrey. Code review quality: How developers see it. In Proceedings of the 38th International Conference on Software Engineering, pages 1028–1038, 2016.

[48] Oleksii Kononenko, Olga Baysal, Latifa Guerrouj, Yaxin Cao, and Michael W. Godfrey. Investigating code review quality: Do people and participation matter? In Proceedings of the 31st International Conference on Software Maintenance and Evolution, pages 111–120, 2015.

[49] Vladimir Kovalenko, Nava Tintarev, Evgeny Pasynkov, Christian Bird, and Alberto Bacchelli. Does reviewer recommendation help developers? IEEE Transactions on Software Engineering, August 2018.

[50] Lisha Li, Zhilei Ren, Xiaochen Li, Weiqin Zou, and He Jiang. How are issue units linked? empirical study on the linking behavior in github. In Proceedings of the 25th Asia-Pacific Software Engineering Conference, pages 386–395, 2018.

[51] Zhixing Li, Yue Yu, Gang Yin, Tao Wang, Qiang Fan, and Huaimin Wang. Automatic classification of review comments in pull-based development model. In Proceedings of the 29th International Conference on Software Engineering and Knowledge Engineering, 2017.

[52] Wanwangying Ma, Lin Chen, Xiangyu Zhang, Yuming Zhou, and Baowen Xu. How do developers fix cross-project correlated bugs? a case study on the github scientific python ecosystem. In Proceedings of the 39th International Conference on Software Engineering, pages 381–392, 2017.

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

[54] Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E. Hassan. 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, pages 192–201, 2014.

[55] Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E. Hassan. An empirical study of the impact of modern code review practices on software quality. Empirical Software Engineering, 21(5):2146–2189, 2016.

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

[57] R. Mishra and A. Sureka. Mining peer code review system for computing effort and contribution metrics for patch reviewers. In 2014 IEEE 4th Workshop on Mining Unstructured Data, pages 11–15, Sep. 2014.

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

[59] Peter Morville and Louis Rosenfeld. Information Architecture for the World Wide Web: Designing Large-Scale Web Sites. O'Reilly Media, 2006.

[60] S. Panichella, V. Arnaoudova, M. Di Penta, and G. Antoniol. Would static analysis tools help developers with code reviews? In 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering, pages 161–170, 2015.

[61] Luca Pascarella, Davide Spadini, Fabio Palomba, Magiel Bruntink, and Alberto Bacchelli. Information needs in contemporary code review. Proc. ACM Hum.-Comput. Interact., 2:135:1–135:27, November 2018.

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

[63] Foyzur Rahman and Premkumar Devanbu. Ownership, experience and defects: A fine-grained study of authorship. In Proceedings of the 33rd International Conference on Software Engineering, pages 491–500, 2011.

[64] Foyzur Rahman and Premkumar Devanbu. How, and why, process metrics are better. In Proceedings of the 2013 International Conference on Software Engineering, pages 432–441, 2013.

[65] Mohammad M. Rahman, Chanchal K. Roy, and Jason A. Collins. Correct: Code reviewer recommendation in github based on cross-project and technology experience. In Proceedings of the 38th International Conference on Software Engineering, pages 222–231, 2016.

[66] Mohammad M. Rahman, Chanchal K. Roy, and Jason A. Collins. Correct: Code reviewer recommendation in github based on cross-project and technology experience. In Proceedings of the 38th International Conference on Software Engineering, pages 222–231, 2016.

[67] Mohammad Masudur Rahman, Chanchal K. Roy, and Raula G. Kula. Impact of continuous integration on code reviews. In Proceedings of the 14th International Conference on Mining Software Repositories, pages 499–502, May 2017.

[68] Mohammad Masudur Rahman, Chanchal K. Roy, and Raula G. Kula. Predicting usefulness of code review comments using textual features and developer experience. In Proceedings of the 14th International Conference on Mining Software Repositories, pages 215–226, 2017.

[69] Mohammad Masudur Rahman, Chanchal K. Roy, and Raula G. Kula. Predicting usefulness of code review comments using textual features and developer experience. In Proceedings of the 14th International Conference on Mining Software Repositories, pages 215–226, 2017.

[70] Michael Rath, Jacob Rendall, Jin L. C. Guo, Jane Cleland-Huang, and Patrick Mäder. Traceability in the wild: Automatically augmenting incomplete trace links. In Proceedings of the 40th International Conference on Software Engineering, pages 834–845, 2018.

[71] P. Rigby, B. Cleary, F. Painchaud, M. Storey, and D. German. Contemporary peer review in action: Lessons from open source development. IEEE Software, 29(6):56–61, Nov 2012.

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

[73] Peter C. Rigby, Daniel M. German, and Margaret-Anne Storey. Open source software peer review practices: a case study of the apache server. In Proceedings of the 30th International Conference on Software Engineering, pages 541–550, 2008.

[74] Peter C. Rigby and Margaret-Anne Storey. Understanding broadcast based peer review on open source software projects. In Proceedings of the 33rd International Conference on Software Engineering, pages 541–550, 2011.

[75] Shade Ruangwan, Patanamon Thongtanunam, Akinori Ihara, and Kenichi Matsumoto. The impact of human factors on the participation decision of reviewers in modern code review. Empirical Software Engineering (EMSE), 24(2):9731016, 2019.

[76] Caitlin Sadowski, Emma Sderberg, Luke Church, Michal Sipko, and Alberto Bacchelli. Modern code review: A case study at google. In International Conference on Software Engineering, Software Engineering in Practice track, 2018.

[77] Todd Sedano, Paul Ralph, and Cécile Péraire. Software development waste. In Proceedings of the 39th International Conference on Software Engineering, pages 130–140, 2017.

[78] Emad Shihab, Zhen Ming Jiang, and Ahmed E. Hassan. Studying the use of developer irc meetings in open source project. In Proceedings of the 25th International Conference on Software Maintenance, pages 147–156, 2009.

[79] Junji Shimagaki, Yasutaka Kamei, Shane McIntosh, Ahmed E. Hassan, and Naoyasu Ubayashi. A Study of the Quality-Impacting Practices of Modern Code Review at Sony Mobile. In Proc. of the International Conference on Software Engineering, pages 212–221, 2016.

[80] Junji Shimagaki, Yasutaka Kamei, Shane McIntosh, David Pursehouse, and Naoyasu Ubayashi. Why are commits being reverted? a comparative study of industrial and open source projects. In Proceedings of the 32nd International Conference on Software Maintenance and Evolution, pages 301–311, 2016.

[81] Mini Shridhar, Bram Adams, and Foutse Khomh. A qualitative analysis of software build system changes and build ownership styles. In Proceedings of the 8th International Symposium on Empirical Software Engineering and Measurement, pages 29:1–29:10, 2014.

[82] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto. The impact of automated parameter optimization on defect prediction models. IEEE Transactions on Software Engineering, pages 1–1, 2018.

[83] Yida Tao, Donggyun Han, and Sunghun Kim. Writing acceptable patches: An empirical study of open source project patches. In Proceedings of the International Conference on Software Maintenance and Evolution, pages 271–280, 2014.

[84] Patanamon Thongtanunam. Studying Reviewer Selection and Involvement in Modern Code Review Processes. PhD thesis, Nara Institute of Science and Technology (NAIST), Japan.

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

[86] Patanamon Thongtanunam, Shane McIntosh, Ahmed E. Hassan, and Hajimu Iida. 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, pages 1039–1050, 2016.

[87] Patanamon Thongtanunam, Shane McIntosh, Ahmed E. Hassan, and Hajimu Iida. Review Participation in Modern Code Review: An Empirical Study of the Android, Qt, and OpenStack Projects. Empirical Software Engineering, 22(2):768–817, 2017.

[88] Patanamon Thongtanunam, Chakkrit Tantithamthavorn, Raula Gaikovina Kula, Norihiro Yoshida, Hajimu Iida, and Kenichi Matsumoto. Who should review my code? a file location-based code-reviewer recommendation approach for modern code review. In Proceedings of the 22nd International

Conference on Software Analysis, Evolution, and Reengineering, pages 141–150, 2015.

[89] Jason Tsay, Laura Dabbish, and James Herbsleb. Influence of social and technical factors for evaluating contribution in github. In Proceedings of the 36th International Conference on Software Engineering, pages 356–366, 2014.

[90] Jason Tsay, Laura Dabbish, and James Herbsleb. Let' s talk about it: Evaluating contributions through discussion in github. In Proceedings of the 22nd International Symposium on Foundations of Software Engineering, pages 144–154, 2014.

[91] Jing Wang, Patrick Shih, C, and John Carroll, M. Revisiting linus' s law: Benefits and challenges of open source software peer review. International Journal of Human-Computer Studies, pages 52–65, May 2015.

[92] Peter Weißgerber, Daniel Neu, and Stephan Diehl. Small patches get in! In Proceedings of the 5th International Working Conference on Mining Software Repositories, pages 67–76, 2008.

[93] Ruiyin Wen. Decision Support for Investment of Developer Effort in Code Review. Master's thesis, McGill University, 3480 Rue University, Montral, QC, Canada, August 2018.

[94] Karl E. Wiegers. Peer reviews in software: A practical guide. Addison-Wesley Longman Publishing Co., Inc., 2002.

[95] Xin Xia, David Lo, Xinyu Wang, and Xiaohu Yang. Who should review this change? putting text and file location analyses together for more accurate recommendations. In Proceedings of the 31st International Conference on Software Maintenance and Evolution, pages 261–270, 2015.

[96] Xin Yang, Raula Gaikovina Kulay, Norihiro Yoshida, and Hajimu Iida. Mining the modern code review repositories: A dataset of people, process and product. In Proceedings of the 13th Working Conference on Mining Software Repositories, 2016.

[97] Robert K. Yin. Case study research: Design and methods. SAGE Publications, 2017.

[98] Fiorella Zampetti, Luca Ponzanelli, Gabriele Bavota, Andrea Mocci, Massimiliano Di Penta, and Michele Lanza. How developers document pull requests with external references. In Proceedings of the 25th International Conference on Program Comprehension, pages 23–33, 2017.

[99] Farida El Zanaty, Toshiki Hirao, Shane McIntosh, Akinori Ihara, and Kenichi Matsumoto. An empirical study of design discussions in code review. In Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, pages 11:1–11:10, 2018.

[100] Motahareh Zanjani, Huzefa Kagdi, and Christian Bird. Automatically recommending peer reviewers in modern code review. IEEE Transactions on Software Engineering, 42(6):530–543, 2015.

[101] Shurui Zhou, Ştefan Stănciulescu, Olaf Leßenich, Yingfei Xiong, Andrzej Wąsowski, and Christian Kästner. Identifying features in forks. In Proceedings of the 40th International Conference on Software Engineering, pages 105–116, 2018.

Appendix

A. The Mechanism of Divergent Discussion at a Fine-Grained Level

Table 19: The taxonomies of abandoned divergent and non-divergent patches.

	Label	Nova					QtBase				
		SP-SN	SN-SP	WP-WN	WN-WP	Non	SP-SN	SN-SP	WP-WN	WN-WP	Non
External Concern	*Unnecessary Fix*										
	Unclear Intention	3	-	6	7	2	2	-	2	7	-
	Already Fixed	9	-	-	-	18	5	1	-	-	3
	Not an Issue	11	-	-	-	5	4	1	-	2	3
	Evolution	2	-	-	-	2	1	-	-	-	1
	Integration Planning										
	Patch Dependency	4	-	2	-	2	1	-	-	-	1
	Blueprint	6	1	-	-	5	-	-	-	-	-
	Release Schedule	15	1	1	-	5	-	-	1	2	-
	Community Buy-in	2	-	-	-	-	-	-	-	-	-
	Communication Breakdown	1	-	-	-	-	-	-	-	-	-
	Persuasion	-	-	-	3	-	-	-	1	2	-
	Policy Compliance										
	Branch Placement	1	-	-	-	3	-	-	1	-	3
	VCS Cleanliness	2	-	-	-	-	-	-	-	-	-
	Lack of Interest										
	Lost by a reviewer	-	-	-	-	4	-	-	-	-	-
	Lost by an author	-	-	-	-	1	-	-	-	-	3
Internal Concern	*Design*										
	Alternative Solution	9	-	-	7	11	2	-	7	8	1
	Flawed Changes	5	1	-	3	10	1	-	1	-	1
	Side Effect	2	-	1	3	3	-	-	3	1	-
	Shallow Fix	10	-	5	4	11	1	-	5	6	1
	Underlying Problem	-	-	-	-	2	-	-	-	-	-
	Documentation	1	-	2	2	-	-	-	2	2	-
	Code Policy Violation	1	-	-	-	-	-	-	-	-	-
	No code	2	-	-	-	-	1	-	-	-	-
	Fix Location	1	-	-	-	1	-	-	-	-	-
	Merged with Related Changes	6	-	-	-	1	1	-	-	-	3
	Split Unrelated Changes	7	-	2	-	5	-	-	-	-	-
	Refactoring	-	-	-	-	1	-	-	-	-	-
	Implementation										
	Bug	-	-	-	-	-	1	-	-	-	-
	Backward Compatibility	1	-	1	1	1	-	-	-	2	-
	Readability	1	-	-	-	-	1	-	-	-	-
	Rebase	-	-	-	1	-	-	-	-	-	-
	Code Issues	-	-	-	-	-	-	-	7	6	-
	Testing										
	Test Design	-	-	2	-	2	1	-	1	-	-
	Test Coverage	1	-	2	1	7	-	-	-	-	-
	Test Failure	-	-	-	-	5	-	-	-	-	2
	Reproduction	-	-	-	-	-	-	-	-	1	-
	Exception	1	-	-	-	-	-	-	-	-	-
	Legal Issues	-	-	-	-	1	-	-	-	1	-
WIP	*Not Intended for Integration*										
	Experimentation	-	-	-	-	-	-	-	-	-	1
	Early Feedback	-	-	-	-	3	-	-	-	-	2