

Doctoral Dissertation

The Practice of Link Sharing in Code Review

Wang Dong

Program of Information Science and Engineering
Graduate School of Science and Technology
Nara Institute of Science and Technology

Supervisor: Kenichi Matsumoto
Software Engineering Lab. (Division of Information Science)

Submitted on February 28, 2022

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

Wang Dong

Thesis Committee:

Supervisor Kenichi Matsumoto
(Professor, Division of Information Science)
Keiichi Yasumoto
(Professor, Division of Information Science)
Takashi Ishio
(Associate Professor, Division of Information Science)
Raula Gaikovina Kula
(Assistant Professor, Division of Information Science)
Patanamon Thongtanunam
(Lecturer, The University of Melbourne)

The Practice of Link Sharing in Code Review*

Wang Dong

Abstract

Code review (CR) is the cornerstone for software quality assurance and a crucial practice for software development. From being a formal code inspection process, nowadays Modern Code Review (MCR) becomes more flexible with asynchronous collaboration through online review tools. Not only improving the quality of code changes, but MCR also serves as a mechanism to increase awareness and share information. Literature review points out that an effective review requires proper understanding. However, it is challenging to identify and acquire the needed information to have a proper understanding to conduct a review.

This thesis presumes that the practice of link sharing can help developers fulfill the information needs in the review process. To address this, first, an empirical study is carried out to explore the prevalence of link sharing, investigate its effect, and qualitatively analyze the intentions. The results show that link sharing is increasingly used, the number of internal links has a positive correlation with the review time, and the intention is often used to provide context understanding. Second, a study is conducted to explore the cross-patch collaborations via patch linkage, and the results reveal that the collaboration contributions are not trivial like voting. Third, this thesis proposes an automatic patch linkage detection model to aid link sharing. The evaluation results show that patch linkage detection is promising, especially for Alternative Solution Linkage.

In all, this thesis emphasizes the role of link sharing in fulfilling information needs during the review process. Furthermore, this thesis provides practical implications to improve the review efficiency and the potential to facilitate the existing code review tools.

*Doctoral Dissertation, Graduate School of Science and Technology, Nara Institute of Science and Technology, February 28, 2022.

Keywords:

Software Engineering, Code Review, Mining Software Repository, Information Needs, Link Sharing

Acknowledgements

I would like to thank the following people for their wisdom, guidance, support, and dedicated effort on my work. Without these people, this thesis would never have been possible.

First and foremost, I would like to express sincere gratitude to my supervisor, Prof. Kenichi Matsumoto for giving me precious opportunities to study Master and Ph.D. programs in his laboratory. He also provides me with insightful guidance and encouragement during my student life. Without his support, I would not successfully accomplish the doctoral degree.

I am deeply grateful to Assist. Prof. Raula Gaikovina Kula for his assistance at every stage of the research project since the first year of Master program. His wisdom helps me to resolve lots of research difficulties, improve my writing skills, and shape the critical thinking. Without his support, I would not have chance to collaborate with excellent software engineering researchers: Lecture Patanamon Thongtanunam, Senior Lecturer Christoph Treude. Moreover, he acts as a friend to relieve the stress during my daily life.

I would like to offer my special thanks to Lecture Patanamon Thongtanunam for her support and patience. She is a very kind and insightful mentor. Not only for her research advice, she also took a serious attention to my research, provided constructive feedback, and brought my work to a higher level. Once again, thank you for your kind support and participating in this committee!

I would also like to express my gratitude to the rest of my thesis committee, including Prof. Keiichi Yasumoto, Assoc. Prof. Takashi Ishio. They give me invaluable comments and suggestions to improve the quality of my research.

I would like to extend sincere thanks to my lab-mates in software engineering lab and close friends (Chen Zheng and Xiao Tao). I would not have been able to have great research experiences without their support.

In addition, I would like to thank my beloved parents for their encouragement all the time. Without them, I would not have had a chance to pursue my dream in Japan. Finally, I would like to extend my sincere thanks to Wang Siyu for her wise counsel and sympathetic ear. She is always there for me and went through this wonderful journey together.

List of Publications

Journal paper

- **Can We Benchmark Code Review Studies? A Systematic Mapping Study of Methodology, Dataset, and Metric?**
Dong Wang, Yuki Ueda, Raula Gaikovina Kula, Takashi Ishio, Kenichi Matsumoto. Journal of Systems and Software (JSS), 180:111009, 2021. (Chapter 3)
- **Understanding Shared Links and Their Intentions to Meet Information Needs in Modern Code Review: A Case Study of the OpenStack and Qt Projects**
Dong Wang, Tao Xiao, Patanamon Thongtanunam, Raula Gaikovina Kula, Kenichi Matsumoto. Empirical Software Engineering (EMSE), 26(5), 1-32, 2021. (Chapter 4)
- **Automatic Patch Linkage Detection in Code Review Using Textual Content and File Location Features**
Dong Wang, Raula Gaikovina Kula, Takashi Ishio, Kenichi Matsumoto. Information and Software Technology (IST), 139:106637, 2021. (Chapter 6)

Contents

Abstract	ii
Acknowledgements	iii
List of publications	iv
Contents	iv
List of Figures	viii
List of Tables	x
1 Introduction	1
1 Problem Statement	2
2 Contributions	3
3 Thesis Outline	4
2 Related Studies	7
I Systematic Mapping Study	11
3 Can We Benchmark Code Review Research?	12
1 Introduction	12
1.1 Chapter Organization	14
2 The Systematic Mapping Process	15
3 Results: Maps of CR Research	25

4	Comparative Analysis	36
5	Towards a Common Benchmark of Dataset and Metric	38
6	Threats To Validity	40
7	Summary	41
II Link Sharing in Code Review		43
4	Understanding Shared Links and Their Intentions to Meet Information Needs	44
1	Introduction	44
1.1	Chapter Organization	47
2	Motivating Example	47
3	Case Study Design	50
4	Case Study Results	62
5	Discussions	76
5.1	Developer Feedback	76
5.2	Suggestions	79
6	Threats to Validity	82
7	Summary	83
5	An Exploration of Cross-Patch Collaborations via Patch Linkage	85
1	Introduction	85
2	Data Collection	87
3	Preliminary Study	88
3.1	Requesting Collaboration	88
3.2	Collaboration after Patch Linkage	91
4	Threats to Validity	94
5	Challenges and Opportunities	94
III Automatic Patch Linkage Detection		96
6	Patch Linkage Detection Using Textual Content and File Location Features	97

1	Introduction	97
	1.1 Chapter Organization	100
2	Motivating Example	101
3	Impact of Patch Linkage on the Review Process	101
4	Patch Linkage Detection	110
5	Discussion	124
6	Threats to Validity	127
7	Summary	128
7	Conclusion	129
	1 Contributions	129
	2 Opportunities for Future Work	132

List of Figures

1.1	An overview of the scope of the thesis.	5
3.1	Defined terms used in the search strings	16
3.2	The distribution of paper publication and their research types yearly from 2011 to 2019. CR papers keep an upward trend and the journal becomes a popular choice for publication. Mixed-Method papers becomes popular in the recent time.	22
3.3	Visual Map for RQ ₁ , showing the contribution and methodology of CR research. The figure shows that evaluation is the most popular methodology, particularly targeting the contributions of understanding and socio-technical effects.	25
3.4	Visual Map for RQ ₂ , showing the replicability of the collected papers. Note that papers analyzed in RQ2 are limited to quantitative and mixed-method papers. The figure shows that 42 papers (50%) provide the public datasets.	31
3.5	Nine research topics with their target metric sets. The figure shows that different research topics tend to target particular metric sets.	35
4.1	Motivating examples of link sharing in MCR process.	49
4.2	An overview of data preparation.	51
4.3	An overview of the RQ2 quantitative analysis.	55
4.4	The proportion of reviews that have links in an interval of three months. In 2015-2019, 25% and 20% of the reviews have at least one link shared in a review discussion within the OpenStack and Qt.	62

4.5	The proportion of internal and external links. 93% and 80% of links that are shared in code reviews are internal links within the OpenStack and Qt.	63
4.6	The direction of the relationships between the number of internal links and the code review time. The light grey area shows the 95% confidence interval. It shows that the more internal links are shared during the discussion, the longer review time will be taken.	69
4.7	Distribution of seven intentions behind sharing links across the studied projects. The results show that <i>Providing Context</i> and <i>Elaborating</i> are the most common intentions for internal and external links, respectively.	75
5.1	A conceptual illustration that describes (1) a linkage between two patches is identified and (2) a collaboration activity happens where a developer on one patch contributes to the review of the other patch.	87
6.1	A real world example to motivate the Alternative Solution linkage between patch #86771 and patch #84977 in OpenStack. The example suggests that the linked patches share similar textual content and modify similar set of file paths.	102
6.2	Box-plots showing comparison among linkage types (<i>Notify-to-Decision-Time</i> and <i>Notify-to-Decision-Revisions</i>). The results show that the patch with an Alternative Solution linkage tends to have a quicker review process after the notification, compared with other patch linkages.	107
6.3	Box-plots showing comparison against a control group (<i>Submit-to-Decision-Time</i> and <i>Submit-to-Decision-Revisions</i>).The results show that compared to patches having no linkages, patches with linkages tend to take a longer time to complete the review process.	107
6.4	The overview of our linkage detection process. To calculate the similarity between the two patches, we focus on the following features: textual content feature (the concatenation of title and description text in a patch) and file location feature (a set of file paths that the patch modifies).	111

6.5	<i>Recall@k_{all}</i> for the detection based on textual content (the concatenation of title and description text in a patch). The results suggest that the recall rates for the textual content model decrease when the time intervals get larger.	118
6.6	<i>Recall@k_{all}</i> for the detection based on file location (a set of file paths that the patch modifies). The results show that the file location as a patch feature overall does not perform as well as the textual content feature with relatively lower recall rates (16%–50% for Qt, 19%–37% for OpenStack, and 19%–51% for AOSP).	120
6.7	<i>Recall@k_{all}</i> for the detection based on file location and textual content for studied projects.	122

List of Tables

3.1	Caption for LOF	17
3.2	Statistics of the filtering of the papers during the conduct search and screening process	19
3.3	Summary of the classification scheme used to identify contribution, methodology, replication, and metric.	21
3.4	Top 5 combination of contribution and methodology	27
3.5	Metric sets used in code review paper	34
3.6	Existing Systematic Review Characteristics	37
3.7	A summary of common metric sets and datasets used in various SE topics.	39
4.1	Studied projects.	51
4.2	The studied explanatory variables.	56
4.3	The taxonomy of intentions for sharing links.	61
4.4	The five most common domains in OpenStack and Qt.	64
4.5	Frequency of link target types in our representative samples. The bold target categories are complemented from the work by Hata et al. [66].	65
4.6	Review time model statistics.	68
4.7	The three most frequent intentions of sharing review links.	75
4.8	Feedback on findings of RQ1 and RQ2, using the Likert-scale scale below: 1 = Strongly disagree, 2 = Partially disagree, 3 = No opinion, 4 = Partially agree, 5 = Strongly agree.	78
4.9	Respondents feedback on the intentions for sharing links.	78

5.1	The prevalence of link types and their timing nature.	88
5.2	The collaboration between the source patch and target patch. . .	91
5.3	The definition of contribution types and their distribution across the link types. Note that one review message can be labeled with more than one contribution type.	93
6.1	Collected dataset including three open source projects: Qt, OpenStack, and AOSP. In total, 11,353 patch linkages are retrieved from these projects.	103
6.2	Ground-Truth based on Hirao et al. [69] and Control Group (patches with no patch linkages).	105
6.3	Statistics showing comparison among linkage types (<i>First-Notify-Revisions</i> and <i>First-Notify-Time</i>). The results suggest that latency exists in the notification of a patch linkage (i.e., the median of 1.2, 3.1, and 2.2 days for Qt, OpenStack, and AOSP).	108
6.4	File path comparison technique descriptions. Similar to the work of Thongtanunam et al. [147] four comparison techniques are included: LCP, LCS, LCSubstr, and LCSubseq.	113
6.5	Dataset used in experiment based on time intervals. Time intervals are divided into 2 days, 7 days, 14 days, and 30 days.	115
6.6	Evaluation results ($Recall@k_{type}$ and $MRR@10$) for the textual content model. The recall rates for detecting the Alternative Solution linkage range from 34% to 69%, 34% to 80%, and 31% to 82% for Qt, OpenStack, and AOSP.	118
6.7	Evaluation results ($Recall@k_{type}$ and $MRR@10$) for the file location model. The recall rates for detecting the Alternative Solution linkage range from 23% to 69%, 34% to 60%, and 26% to 72% for Qt, OpenStack, and AOSP.	120
6.8	Evaluation results ($Recall@k_{type}$ and $MRR@10$) for the feature combination model. The higher $MRR@10$ scores show that the model can detect more patch linkages in higher ranks (i.e., 33%–44% for Qt, 33%–43% for OpenStack, and 40%–53% for AOSP). .	121

6.9	Evaluation results ($Precision@k_{type}$) for the Alternative Solution linkage. The results show that precision is relatively higher in the separate models than in the feature combination models (i.e., 60%–74% and 43%–67% in the textual content model for Qt and OpenStack; 56%–67% in the file location model for AOSP).	123
6.10	Evaluation results using sample-based datasets. The statistics show that in the sample-based evaluation, the file location model performs better than the textual content model.	126

1 | Introduction

Code review is a well-documented practice for software quality assurance, where developers discuss and examine the code changes. It is recognized as a valuable tool by development community for reducing software defects [3, 4]. Fagan [47] found that code inspections with in-person meetings reduce the number of post-release defects. Shull et al. [136] also reported that code review often catches more than half of a product’s defects. Moreover, Aurum et al. [12] stated that code review improves the overall quality of software systems.

Code review practice has been constantly evolving. The formal variant of code review, better known as software inspection or Fagan-inspection [47], was first introduced in 1976, and has been an effective quality improvement practice for a long time [48]. It is a well-structured structured process for reviewing source code with the single goal of finding defects, usually conducted by groups of reviewers in extended meetings. Fagan inspection involves six standard steps, i.e., Planning, Overview, Preparation, In-spection Meeting, Rework, and Follow-up. However, with the increasing popularity of distributed software development, formal code inspection is hard to adopt due to the requirement of synchronous and physical meetings.

Recently, Modern Code Review (MCR), an informal, lightweight, and tool-based code review, has been widely used in both open source software and industrial software [13]. The MCR process comprises of patch upload, reviewer assignment, examination and discussion, and integration. Broadly speaking, developers (the reviewers) other than the code change author manually inspect the submitted changes and provide feedback that needs authors to address before the code changes are finally integrated into the codebase [26]. Existing studies

have proved that MCR has the potential to improve the software quality and dependability [101, 105]. Not only improving the quality of code changes, MCR also has additional benefits. MCR is a collaborative process, where developers and authors relying on review tools conduct an online discussion asynchronously. Bacchelli and Bird [13] found that knowledge transfer and team awareness are cited as the frequent motivations as well for code review. Indeed, their interview cited that MCR reveals as an effective mechanism to increase awareness and share information: “Code reviews are good FYIs [for your information].”

1 Problem Statement

Nevertheless, code reviews also incur cost on software development since they can delay the integration of the code change and further slow down the overall development process [46, 113]. Tao and Kim [143] pointed out that the time spent by a developer on reviewing code changes is non-negligible, especially when multiple issues are addressed in a single change. The code change integration can even further be delayed if the reviewers have difficulty in understanding the changes, i.e., not sure about the correctness, run-time behavior [13, 144]. Similarly, Baum et al. [18] found that the erratic outcome of review process is caused by the cognitive-demanding nature of reviewing. Reviewers often request additional information about correct understanding and alternative solution during reviews [113]. Ebert et al. [46] observed that the three most frequent reasons for confusion are missing rationale, discussion of the solution, and lack of familiarity with existing code. Hence, identifying and acquiring the needed information to gain a proper understanding is needed.

Recent work shows that the shared links in review discussions can be used to provide information. Jiang et al. [73] found that various types of links are shared in pull-based reviews. Meanwhile, Hirao et al. [69] show that shared links between reviews can be used to indicate the information about patch dependency, broader context, and alternative solution. Despite the above attempts, it still remains unclear about (i) the effect of shared links in the review process, (ii) how shared links fulfill information needs, (iii) the feasibility of automatically identifying links between reviews to improve traceability. Therefore, I state this

thesis as follows:

Thesis Statement: An effective review requires proper understanding. If the needed information would be at hand, it will reduce developers' cognitive load and further improve the review efficiency. However, it is challenging for developers to identify and acquire the needed information to conduct a code review. This thesis presumes that the practice of link sharing, one common and convenient way of knowledge sharing, could fulfill such information needs.

2 Contributions

The main contributions of this thesis can be classified into three categories: empirical observations, survey insights, and future CR research directions.

Empirical Observations

1. Links are increasingly shared in the review discussion, i.e., 20% to 25% of the reviews have at least one link shared in studied projects. (Chapter 4)
2. Linear regression model results show that the internal link (project related links) has a significant correlation with the code review time. (Chapter 4)
3. Seven intentions behind sharing links are identified. The most popular intention for internal links is to provide context, while for the external links, to elaborate the review discussions is the most common intention. (Chapter 4)
4. Collaboration contributions across review links are not trivial. (Chapter 5)
5. There exists latency in the notification of linked patches. (Chapter 6)
6. Patch linkage detection is promising using textual content and file location features, especially for the alternative solution linkage. (Chapter 6)

Survey Insights

1. The information brought from the shared links is useful and could aid the code review process. (Chapter 4)
2. Existing functionalities integrated with review tools, which provide review links of related changes and the same topic, has limitations. (Chapter 4)

Future CR research directions

1. All available datasets and metrics of existing CR research are collected through a mapping study, which would be beneficial for future research. (Chapter 3)
2. There is a lack of researches that report the CR experience and propose solutions to deal with CR problems. (Chapter 3)
3. At this stage, a benchmark for CR research is not mature but has a much-needed potential. (Chapter 3)

3 Thesis Outline

In this section, I provide an outline of this thesis. Figure 1.1 presents the structure of the thesis and the potential research outcomes. In the remainder of the thesis, first of all I introduce the general studies that are related to this thesis.

- **Chapter 2**— Three main related topics are discussed as follows: (i) code review models, (ii) the practice of link sharing in software engineering domain, and (iii) duplicate software artifact detection.

To systematically understand the challenge of code review research field, a mapping study is performed (**Part I Systematic Mapping Study**).

- **Chapter 3**— This chapter studies the contribution and methodology, datasets, and metrics of the code review researches that are published in the last decade. This study makes it convenient for researchers to use the existing datasets and guides researchers to choose appropriate metrics in specific

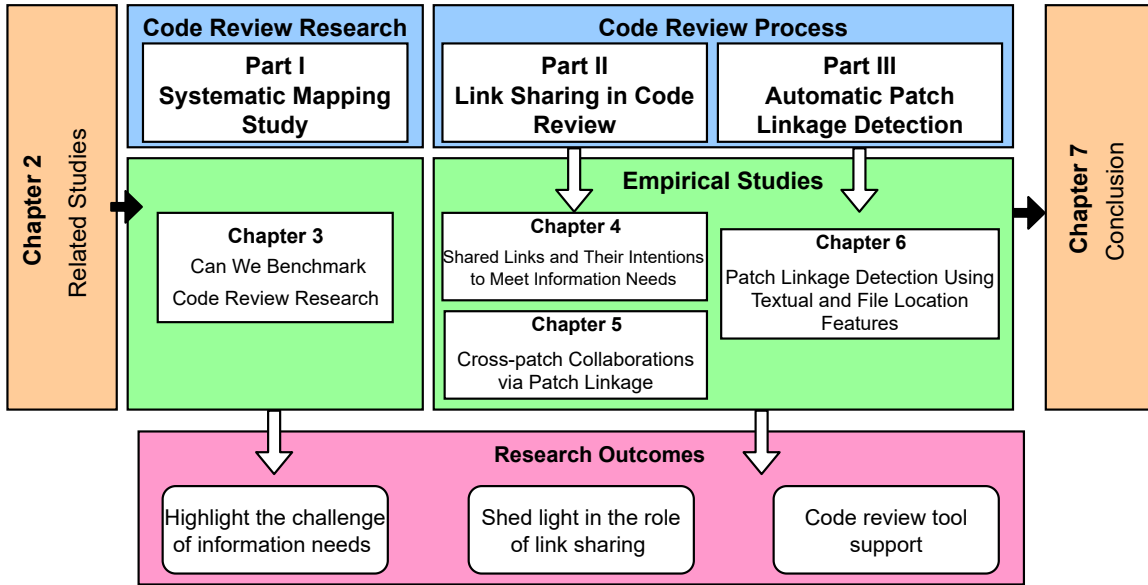


Figure 1.1: An overview of the scope of the thesis.

topics. The mapping results also reveal that one of challenge is to fulfill information needs during review process. At the same time, little is known about how to identify and acquire the needed information.

Motivated by the systematic mapping study, I then present the main empirical studies that are split into three chapters, to explicitly address the challenge of information needs during review process. These chapters are organized into **Part II Link Sharing in Code Review** and **Part III Automatic Patch Linkage Detection**. Each chapter has corresponding potential research outcomes.

- **Chapter 4**— This chapter introduces the empirical study to investigate the practice of link sharing and their intentions. The results show that links are increasingly shared in the review discussions. The internal link has a significant correlation with the code review time. Moreover, seven intentions behind link sharing are identified.
- **Chapter 5**— This chapter explores the collaboration activities across the patch, after the patch link is provided in the review discussion. Mixed-method results show that the collaboration contributions are non-trivial,

with key contributions like voting which affects the review outcome of the target patch, or revising which improves the patch.

- **Chapter 6**— This chapter investigates the feasibility of automatically detecting patch linkage, in order to improve the patch traceability and further make an efficient review process. The model evaluation results show that the detection model using textual content and file location features is promising, with high recall rates.

Finally, **Chapter 7** provides our conclusion that includes the main research results and contributions of this thesis. We also provide potential opportunities for future code review research.

2 | Related Studies

Code review models - Code review is used to examine the changes made by other developers, to find potential defects and improve project quality since the 1970s [47]. In an earlier time, it was well known as code inspection. Many tools were developed to support the formal process of code inspection [38, 53, 97, 115]. Over the last decade, code review relying on modern review tools has been widely adopted by many open source projects [125], i.e., Gerrit tool in OpenStack, pull request in GitHub projects, CodeFlow in Microsoft [118]. There are two review styles: review-then-commit (RTC) and commit-then-review (CTR). In the pioneering work, Rigby et al. [128] empirically examined the comparison between these two review techniques. For the style of CTR, projects allow trusted developers to commit contributions before they are reviewed. In contrast to CTR, RTC is a technique where a review is made before committing it to the original code [129]. For instance, the Android project adopts RTC style using the Gerrit tool to provide a discussion platform for the review process before the patch is merged into the codebase [106]. Another example is that GitHub projects apply pull-based development to conduct the code review [56, 58]. The study of open source projects that use either the Gerrit tools or GitHub pull requests has been extensively studied [126, 147, 178]. Our proposed solution is not applicable to all projects. For instance, Baum et al. [21] reported that about 50% of commercial teams use CTR review style. However, with the growth of large open source projects, it is possible that teams are not able to be aware of the related patches in RTC. ***This thesis*** focuses on the risk when team awareness is lacking, which is only applicable for large teams or reviewing models that are RTC. This includes all Gerrit tools and all GitHub Open Source projects, as they are pull-based

developments.

Collective knowledge through links sharing - Link sharing has become an important activity in the area of software engineering, which encourages developers to exchange knowledge, increases the learning purpose, and mitigates potential issues. The value of link sharing has been widely explored in the Q&A site and Github. Gomez et al. [54] found that a significant proportion of links shared on Stack Overflow (i.e., the Q&A site for professional and enthusiast programmers) are referring readers to software development innovations like libraries and tools. Ye et al. [170] used the URLs shared in StackOverflow to generate the structural and dynamic properties of the emergent knowledge network, aiming to enable more effective knowledge sharing in the community. With the increasing growth of Github, 9.6 million links exist in source code comments across 25,925 repositories [66]. They identified more than a dozen different kinds of link targets, with dead links, licenses, and software homepages being the most prevalent. As the survey conducted by Baltes and Diehl [16], 40% of participants added a source code comment with a Stack Overflow link to the corresponding question or answer in Github projects. They analyzed how often these URLs are present in Java files and found that developers more often refer to questions, i.e., the whole thread, than to specific answers. Related to the issue linking, existing studies have demonstrated the value of such links in identifying complex bugs and duplicate issue reports. For instance, Boisselle and Adams [31] reported that 44% of bug reports in Ubuntu are linked to indicate duplicated work. The results of Zhang et al. [179] showed that developers tend to link issues more cross-project or cross-project over time. To ease the recovery of links, Zhang et al. [180] proposed iLinker to address the problem of acquiring related issue knowledge so as to improve the development efficiency. Rath et al. [122] showed that on average only 60% of the commits were linked to specific issues and proposed an approach to detect missing links between commits and issues using process and text-related features.

Link Sharing in Code Reviews - Links are also been studied in peer code review settings. Zampetti et al. [173] investigated to what extent and for which purpose developers refer to external online resources when raising pull requests. Their results indicate that external resources are useful for developers to learn

something new or to solve specific problems. Jiang et al. [73] found that 5.25% of pull requests have links in review comments on average in ten GitHub projects. Hirao et al. [69] suggested that review linkage is not uncommon in six studied software communities. They observed five types of review linkage, such as patch dependency, broader context, alternative solution. *This thesis* expands upon the work of Hirao et al. by studying not only review links but all kinds of links. In addition, this thesis systematically studies the effect of links on the review process and identifies intentions behind links.

Duplicate pull request detection - Within code review models, the topic of the duplicate pull request is specifically investigated. Yu et al. [171] created a dataset extracted from 26 open source projects in Github by using a semi-automatic approach. Their analysis found 21% of duplicates were identified after a relatively long latency (more than one week). Additionally, their statistics show that the redundant review efforts were spent on the duplicates (i.e., on average 2.5 reviewers participating in the redundant review discussions and 5.2 review comments are generated before the duplicate relation is identified. To address the duplicate pull request detection, by now there are two threads of work: one is using information retrieval and the other one is using classification. For the information retrieval thread, Li et al. [88] used text information of pull request to detect duplicates on three popular projects hosted in GitHub. The evaluation shows that about 55.3%–71.0% of the duplicates can be found when we use the combination of title similarity and description similarity. For the classification thread, Ren et al. [123] calculated the similarity of nine pull request features where title and description are both included, and then they adopted a machine learning algorithm to aggregate the nine features. The result shows that the proposed classifiers achieve 57–83% precision for detecting duplicate code changes from the maintainer’s perspective, which outperforms the Li et al.’s work [88]. Recently, Wang et al. [162] integrated the time feature to the nine features proposed by Ren et al. and the experimental results show that it can substantially improve the performance of Ren et al.’s work by 14.36% and 11.93% in terms of F1-score@1 and F1-score@5, respectively. *This thesis* is different from the duplicate pull request detection, as focus is not only limited to the duplication, but also consider other patch linkages (i.e., alternative solution, broader context, and dependency).

Duplicate bug report detection - Although the duplicate pull request detection is not widely studied, there has been much work on investigating the detection of other duplicate artifacts in SE domains, such as duplicate bug reports in the issue tracking system. Bug reports provide textual description that involves the natural language bug description reported by developers, i.e., title and description. Natural language information and information retrieval (IR) techniques are widely used to calculate the similarity scores between a given data and the retrieved data. For instance, Runeson et al. [131] took natural language text of bug reports and performed standard tokenization, stemming, and stop word removal. Their results recognize the importance of using textual information with two-thirds of the duplicates can be found through NLP technologies. Bettenburg et al. [29] studied the importance of duplicate bug reports and found that the additional information provided by duplicates helps to resolve bugs quicker. At the same time, their classification model achieved an average precision and recall of 68% and 60%. Researchers also found that in addition to natural language information, other extra report information can improve duplicate retrieval. Wang et al. [163] combined natural language and execution information to detect the duplicate reports. The approach they proposed can detect more duplication compared to only using natural language information alone (i.e., 67%–73% of duplication can be detected). Sun et al. [142] proposed REP to improve the accuracy of duplicate bug retrieval. Not only text but also other available information such as product, component, and priority are fully utilized in REP.

Part I

Systematic Mapping Study

3 | Can We Benchmark Code Review Research?

The rise of contemporary review tools has brought the availability of data and has driven a large body of code review researches since the last decade. This chapter introduces a systematic mapping study to investigate the Contribution and Methodology, Datasets, and Metrics that are used in the code review related research. The mapping study would help researchers and practitioners to understand the challenge and keep track of the best practices and state-of-the-art.

1 Introduction

Code Review (CR) has always been the cornerstone for software quality assurance and is a crucial practice for software development teams. CR not only has the benefits of finding defects, but assists with other activities such as knowledge transfer and team awareness within software teams [13]. Microsoft reveals how “CRs at Microsoft are an integral part of the development process that thousands of engineers perceive it as a great best practice and most high-performing teams spend a lot of time doing”.¹ The rise of contemporary review tools has brought the availability of data, with the review process now being light-weight. Contemporary tool-based reviews (such as Gerrit², Codestriker³, and ReviewBoard⁴)

¹<https://www.codegrip.tech/productivity/how-microsoft-does-its-code-review/>

²<https://www.Gerritcodereview.com/>

³<http://codestriker.sourceforge.net/>

⁴<https://www.reviewboard.org/>

are widely used in both open source and proprietary software projects. As CR research increases, so does the diversity of research methodologies, datasets, and metrics also increases, making it difficult to keep track of best practices.

This paper collects methodology, dataset, and metric for CR studies, with the end-goal to investigate the potential of benchmarking. Other fields, such as bio-medicine⁵, use benchmarking studies to address the issue of: ‘*as increasing numbers of methods are published in certain fields, it can be difficult to keep track of best practices for their use*’. Large-scale studies that benchmark these methodologies on a wide range of datasets can be extremely useful to the scientific community. In this regard, we conduct a systematic mapping study, executing the guidelines provided by Petersen et al. [116]. The scope of the systematic study revolves around three research questions: to uncover (RQ1) the state of contributions and methodologies for research, (RQ2) replicability of existing research, and (RQ3) the metrics used in CR research.

Through a collection of 19,847 papers from the high-impact SE venues, we generate visual maps for the 112 collected papers including 80 conferences and 32 journals. For *RQ1*, we find that evaluation is the most common methodology (i.e., 73 papers), targeting particularly socio-technical and the understanding aspects of the CR process. However, there is a lack of papers that report the experience and propose solutions to deal with CR problems (i.e., four papers and thirteen papers, respectively). For *RQ2*, the results show that CR research not only relies on the data sources from the CR process but also largely uses the data sources from the software development process (i.e., issue tracking system and GitHub). We observe that 50% of researches provide replicable datasets, i.e., 42 papers out of 84 papers that use quantitative or mixed-method. For *RQ3*, we grouped 457 metrics that are used in the quantitative research into sixteen core sets (i.e., *experience, code, ownership, comment, file, participant, temporal, revision, description, module, defect, queue, workload, decision, language, log, and others*) and classified nine research topics (i.e., *Quality Assurance, Review Process Prediction, Acceptance Predication, Review Process, Review Participation, Review Process Prediction, Review Process Comments, CI & Review, Test & Review and Technical/Non-technical & Review*). We observe that the SE topic of quality as-

⁵<https://www.biomedcentral.com/collections/benchmarkingstudies>

insurance is more likely to use metrics to conduct the research, with thirteen papers being studied. In addition, the mapping shows that experience and code metric sets are the two most frequent metrics used in the quantitative study. From the mapping between metric sets and research topics, we find that different research topics tend to use particular metric sets.

Upon further mapping between the common datasets and the metrics, we conclude that at this stage, a benchmark for CR studies is not mature but has a much-needed potential. The map shows that papers prefer to construct their own metrics and datasets. To promote the common dataset usage, we encourage the future researches to strive for a replicable dataset. With the rise of machine learning and AI techniques, CR researchers will soon need to evaluate performance accurately against a state-of-the-art benchmark. We envision that such a benchmark will facilitate new researchers, including experts from other fields, to propose new techniques and build on top of already established methodologies.

To highlight the novelty of the mapping study, we did a comparative analysis of existing systematic reviews in CR, following the protocol provided by Petersen et al. [116]. Three secondary studies [14, 42, 80] are identified before or in 2019. Kollanus and Koskinen [80] conducted a mapping study on the software inspections, which is outdated, not targeting the tool-based reviews. Coelho et al. [42] focused on the specific theme of tool-based code review, i.e., refactoring-awareness. Badampudi et al. [14] did a preliminary study on understanding the research topic evolution of the tool-based review field. Although these two systematic studies are highly relevant to the field, the study first gives a visual summary of the potential of the benchmark with in-depth analysis. Specifically, we only study those papers that are published in the premium venues, as we believe that high-quality papers are deemed to form a best representative view for future researches. Furthermore, the outcome of this mapping study is a listing of methodologies and contributions, 42 available datasets, and 457 metrics.

1.1 Chapter Organization

The remainder of this chapter is organized as follows. Section 2 presents the systematic mapping process, including research questions, search conduction, screening process, classification schemes, and data extraction. Section 3 shows

the results of the systematic mapping study. Section 4 describes the comparative analysis of existing systematic reviews relevant to CR. Section 5 discloses the challenges for the benchmark of dataset and metric. Section 6 explains the threats to the validity of the research. Finally, we summarize this paper in Section 7.

2 The Systematic Mapping Process

The process is based on the work of Petersen et al. [116] and similar to the systematic mapping study performed by Abelein and Paech [2]. Essentially, the process steps of the systematic mapping study are the definition of the research questions, search conduction of papers, screening process, keywording for the mapping, and data extraction.

Research Questions

To define the scope of the mapping study, we formulate the following research questions:

1. *(RQ1): What contributions and methodologies does CR research target?*
The motivation for the first research question is to understand the current focus of research. Based on the work of Bacchelli and Bird [13], we would like to map out the outcomes, expectations, and contributions that the most impactful CR research tackles from the point of view of both a practitioner and researcher.
2. *(RQ2): How much CR research has the potential for replicability?* The motivation for the second research question is to understand how the data source impacts CR research. Understanding the sources can provide insight into the current state and gaps in terms of the data collection and availability. Furthermore, there has been growing initiatives to make data open and replicability which is encouraged in the community ⁶.

⁶A recent initiative by the Springer EMSE Journal shows how research is working towards open science and replicable studies at <https://github.com/emsejournal/openscience>

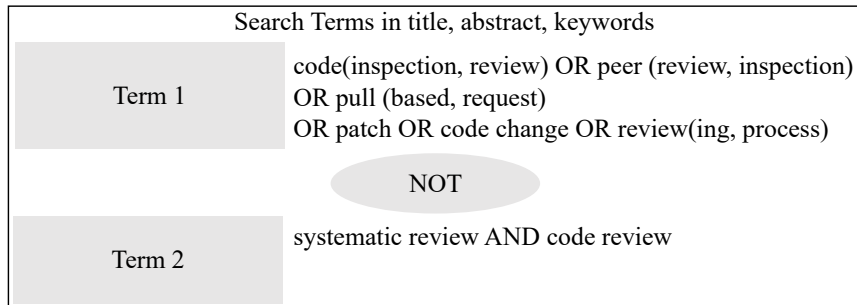


Figure 3.1: Defined terms used in the search strings

3. (*RQ3*): *What metrics and topics are used with CR studies?* The motivation for the third research question is to uncover what kinds of metrics are used in CR empirical studies. Understanding how the metrics are used and the associated topics can motivate the potential to benchmark CR studies.

Conduct Search

We use the following strict characteristics as recommended by A. Kitchenham [1] to formulate our search string: (C1) a defined search strategy, (C2) a defined search string, based on a list of synonyms combined by ANDs and ORs, (C3) a broad collection of search sources, (C4) strict documentation of the search, (C5) paper selection should be checked by at least two researchers. Figure 3.1 shows the defined search string. For Term 1 in the search string, as shown in Figure 3.1, We apply commonly used terminologies in CR to search, such as code inspection, code review, peer review, peer inspection, or tools such as pull requests or pull-based. To enlarge the paper dataset, other terminologies such as patch, code changes, and reviewing are also considered as appropriate for CR research. For Term 2 in the search string, we do not include the CR related papers that are in the form of the systematic review. Nevertheless, we separately conduct a comparative analysis of existing systematic reviews in Section 4.

Based on RQ1, to ensure papers of high quality and understand the state-of-the-art in the field, we specifically searched for papers published in the high-impact journals and conferences from the software engineering domain between 2011 and 2019. Inspired by the work of Badampudi et al. [14], we select the

Table 3.1: Corpus of venues (conferences and journals) studied in this paper. Note that ICSM now is called ICSME; and WCRE and CSMR are fused into SANER. In addition, the GPCE h5-index is not retrieved but is ranked as B.

Journal	Name	Impact factor	Established
TSE	IEEE Transactions on Software Engineering	6.112	1991
ESE	Empirical Software Engineering	3.156	1996
IST	Information and Software Technology	2.726	1992
S/W	IEEE Software	2.589	1991
TOSEM	Transactions on Software Engineering and Methodology	2.516	1992
JSS	The Journal of Systems and Software	2.450	1991
REJ	Requirements Engineering Journal	1.933	1996
SOSYM	Software and System Modeling	1.876	2002
ASEJ	Automated Software Engineering Journal	1.857	1994
SPE	Software: Practice and Experience	1.786	1991
SQJ	Software Quality Journal	1.460	1995
STVR	Software Testing, Verification and Reliability	1.226	1992
SMR	Journal of Software: Evolution and Process	1.178	1991
ISSE	Innovations in Systems and Software Engineering	0.950	2005
IJSEKE	International Journal of Software Engineering and Knowledge Engineering	0.886	1991
NOTES	ACM SIGSOFT Software Engineering Notes	0.490	1999
Conference	Name	h5-index	Established
ICSE	International Conference on Software Engineering	75	1994
FSE	ACM SIGSOFT Symposium on the Foundations of Software Engineering	51	1993
ASE	IEEE/ACM International Conference on Automated Software Engineering	40	1994
MSR	Working Conference on Mining Software Repositories	38	2004
ISSTA	International Symposium on Software Testing and Analysis	35	1989
ICSM	IEEE International Conference on Software Maintenance	33	1994
ICPC	IEEE International Conference on Program Comprehension	33	1997
SANER	IEEE International Conference on Software Analysis, Evolution and Re-engineering	30	2014
ICST	IEEE International Conference on Software Testing, Verification and Validation	27	2008
RE	IEEE International Requirements Engineering Conference	25	1993
CSMR	European Conference on Software Maintenance and Re-engineering	25	1997
WCRE	Working Conference on Reverse Engineering	22	1995
MDLS	International Conference On Model Driven Engineering Languages And Systems	21	2005
ESEM	International Symposium on Empirical Software Engineering and Measurement	20	2007
FASE	International Conference on Fundamental Approaches to Software Engineering	18	1998
SSBSE	International Symposium on Search Based Software Engineering	15	2011
SCAM	International Working Conference on Source Code Analysis & Manipulation	15	2001
GPCE	Generative Programming and Component Engineering		2000

2011 time-frame as our starting point, since Badampudi et al. identified a steady upward trend regarding publications related to the contemporary tool-based review starting in 2011. In addition, several well-known open-source projects (e.g., OpenStack and Qt projects) use the Gerrit platform since 2011. Table 3.1 shows the summary of paper collection source. Regarding the publication venue selection, similar to the mapping study conducted by Mathew et al. [98], papers were extracted from 18 conferences (i.e., International Conference on Software Engineering) with relatively high h5-index and 16 journals with high impact factors. h5 is the h-index for articles published in a period of 5 complete years obtained from Google Scholar. We rely on Guide2Research⁷ to retrieve the h5-index. Although Generative Programming and Component Engineering (GPCE) is not recorded with the h5-index, it is ranked as B according to core ranking.⁸ The Impact Factor (IF) is an index (numerical value) to evaluate how much impact a journal (scientific journals) has, based on Clarivate Analytics.⁹ The conferences with higher h5-index or the journals with higher impact factors are deemed to carry more intrinsic prestige in their respective fields. To reduce its selection bias, we selected from a wide range of digital resources to follow (C3) a broad collection of search sources: ACM Digital Library, IEEE Xplore, Science Direct, and SpringerLink databases. For example, the data from 2012 to 2019 for Mining Software Repositories Conference is collected through IEEE Xplore, but the data from 2011 is available in ACM Digital Library. We extracted 19,847 papers from the above four search sources that were published in the last nine years (i.e., 2011~2019), as shown in Table 3.2.

Assessing the quality of primary papers can be used as an additional criterion for the exclusion [77]. As part of our quality assessment, we exclusively consider papers from these premium venues as we assume they are of high quality and widely get recognized within the SE domain. Additionally, to ensure only technical contributions, in the further data processing, we filter out short papers, editorials, tutorials, panels, poster sessions and prefaces, and opinions (8 pages or less). Nonetheless, internal and conclusion threats may exist, and we further discuss the threats with regard to this assessment in Section 6. After the conduct

⁷<https://www.guide2research.com/>

⁸<http://portal.core.edu.au/conf-ranks/>

⁹<https://clarivate.com/webofsciencegroup/essays/impact-factor/>

Table 3.2: Statistics of the filtering of the papers during the conduct search and screening process

		# of Papers
Conduct Search		
	Search String Result	19,847
<i>All Papers</i>		437
Screening of Papers		
	Conference paper	80
	Journal paper	32
<i>Total Papers</i>		112

search, we were able to get 437 initial papers, as shown in Table 3.2.

Screening Process

Our screening process is comprised of inclusion and exclusion criteria. For this manual exclusion, the following inclusion and exclusion criteria were applied to the abstract of each paper. *Inclusion criteria:* Three inclusion criteria are defined, namely, (*IC1*): paper should focus on topics on code inspections, code review, code review tools, pull request, (*IC2*): *the paper is peer reviewed*, (*IC3*): the paper is written in English and the paper has full text available. *Exclusion criteria:* Four exclusion criteria were defined that cover the datasets, purposes and the evaluation of the studies. The following papers were excluded that met these criteria: (*EC1*): the paper does not mention any CR activities, (*EC2*): the paper focuses on other software development process, e.g., issue tracking process, continuous integration, testing, (*EC3*): the paper is out of scope with focusing on other sub-fields such as program analysis, code clone, defect prediction, refactoring, social technique, (*EC4*): the paper is outside our studied time-frame.

To reduce bias and follow (C5), this manual paper selection was conducted by the first and the second authors. As a result of the screening process, we were able to collect 112 papers out of 437 initial papers, which include 80 premium conference papers and 32 high-impact journal papers, as shown in Table 3.2.

Figure 3.2(a) depicts the distribution of these 112 papers based on conferences and journals during our studied time frame. In detail, the figure shows that the CR research publications keep an upward trend in the recent three years, i.e., fourteen, sixteen, and twenty papers are published in 2017, 2018, and 2019. We also observed that papers submitted to journals are on the upward trend from 2015, i.e., eight papers were submitted to journals in 2019.

To further explore the trend of the research type, we manually classified the types of research papers (e.g., quantitative and qualitative) according to the work of Bernard [27]. We classify the research types into four categories: i) Quantitative only, ii) Quantitative only, iii) Mixed-Method, and iv) Survey only. The Mixed-Method refers to those papers using the combination of quantitative method and qualitative/survey method. For the Survey only type, it not only includes survey but also includes interview and user/control studies. We classify papers which do not fit the above types into others. We classify research paper types with two rounds. First, two authors classified them in the first round. In the second round, the third author full with research experience joined to validate each collected paper. Figure 3.2(b) shows the research type distribution of 112 papers during our studied time frame. We observe that the Mixed-Method papers become popular in the recent time, i.e., eleven Mixed-method papers are published in 2019.

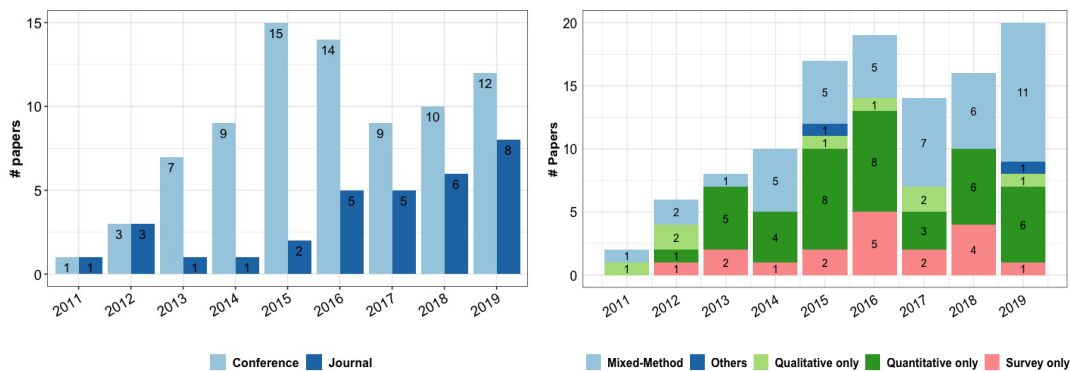
Keywording of Relevant Papers

Inspired by Petersen et al. [116], we classified each paper based on the scope outlined in each research question with results shown in Table 3.3. During the classification, it not only includes the detailed reading of the abstract, but sometimes requires a careful reading of the whole paper itself.

Contributions and Methodologies (RQ1). To classify research contributions of the papers, we base our work on the work of Bacchelli and Bird [13]. They classify contributions for two objectives (i.e., contributions to benefit practitioner and researcher). For the classification process, three co-authors sat in a round-table and labeled each contribution based on seven category features shown in Table 3.3. The process was to first read the abstract and decide the classification.

Table 3.3: Summary of the classification scheme used to identify contribution, methodology, replication, and metric.

Class	Sub-class	Category	Description
Contributions	Practitioner	Communication [13]	The developers are provided with the need of richer communication than comments annotating the changed code when reviewing. Teams should provide mechanisms for in-person or, at least, synchronous communication.
		Potential Benefit [13]	Modern CR provides benefits beyond finding defects. CR can be used to improve code style, find alternative solutions, increase learning, share code ownership, etc. This should guide CR policies.
		Quality Assurance [13]	CR does not result in identifying defects as often as project members would like and even more rarely detects deep, subtle, or “macro” level issues.
		Understanding [13]	When reviewers have prior knowledge of the context and the code, they complete reviews more quickly and provide more valuable feedback to author.
	Researcher	Automation [13]	Tools for enforcing team code conventions, checking for typos, and identifying dead code already exist. Even more advanced tasks such as checking boundary conditions or catching common mistakes have been shown to work in practice on real code. Automating these tasks frees reviewers to look for deeper, more subtle defects.
		Program comprehension [13]	Context and change understanding are challenges that developers face when reviewing, with a direct relationship to the quality of review comments.
Socio-technical effect [13]		These are studies that involves the consideration of both human and technical aspects. In terms of CR, Studies can be designed and carried out to determine if and how team collaboration, coordination, awareness and learning occurs.	
Methodologies	-	Validation Research [165]	Techniques investigated are novel and have not yet been implemented in practice. Techniques used are for example experiments, i.e. work done in lab
		Evaluation Research [165]	Techniques are implemented in practice and an evolution of the technique is conducted. That means, it is shown how the technique is implemented in practice (solution implementation) and what are the consequences of the implementation in terms of benefits and drawbacks (implementation evaluation). This also includes to identify problems in industry.
		Solution Proposal [165]	A solution for a problem is proposed, the solution can be either novel or a significant extension of an existing technique. The potential benefits and the applicability of the solution is shown by small example or a good line of argumentation.
		Experience Paper [165]	Experience papers explain on what and how something has been done in practice. It has to be the personal experience of the author
		Survey Paper	These papers are qualitative studies that use a questionnaire or interviews to evaluate some phenomena
Replication	-	Private Datasets	Neither dataset nor the source code is available. The study may not be replicated.
		Partial Datasets	Part of the dataset is available. The study could not be replicated fully with partial datasets.
		Public Datasets	Replication including either full dataset or the source code is provided via hyperlinks or paper references. The study is deemed to be replicable using provided datasets.
Metric	-	Metric sets used in empirical studies	Metrics that are used in CR research can be classified according to the level of three aspects: product, process, and people.



(a) Distribution of paper publication

(b) Distribution of research types

Figure 3.2: The distribution of paper publication and their research types yearly from 2011 to 2019. CR papers keep an upward trend and the journal becomes a popular choice for publication. Mixed-Method papers becomes popular in the recent time.

If there was a dispute, then the paper was quickly analyzed and a discussion of the paper started between the co-authors before the consensus reached. To classify methodologies that were applied to the studies, we used existing definitions of research facets [116]. For the classification, three co-authors sat in a round-table and labeled each methodology based on the category features. The first keywords relating to the methodology were searched and discussed. Similar to the keywording of contributions, the full contents of the paper were consulted if a dispute arose among the co-authors.

Replication (RQ2). To classify the replicability of papers, we identified the source of the data, whether the dataset is either available via the link or is referred to a prior dataset. Our scope is limited to the quantitative and mixed-method papers. Since detailed information of the dataset is not likely to be in the abstracts, co-authors were required to scan the papers to extract any online links of a dataset or a reference to an existing dataset. Furthermore, as shown in Table 3.3, authors classified the papers according to the nature of the studied systems (i.e., open source projects or industry). Note that the classification is non-exclusive as some studies involved projects that were both open and closed

data.

Metrics (RQ3). To group the metrics used in collected papers, we only scan the papers conducted in quantitative method and mixed-method. For the classification, we do the following two: (1) metrics mapping research aspects and (2) metrics mapping research topics. For the first classification, we pick up all metric description tables from papers. We then apply open card sorting to construct a taxonomy of codes of the metrics. In detail, following the metric descriptions, the coded metrics are merged into cohesive groups that can be represented by a similar high-level code, i.e., metric sets. In the card sorting process, three authors sit together and sort metrics until all achieve the consensus. Based on prior work [169], the aspects of CR research can be divided into three targets: product, process, and people. Following these aspects, we then classify the constructed high-level metric sets using ticks. For the second classification, the first author classified initial research topic groups by reading the abstracts and introductions. After that, another experienced author did the validation to assure the constructed topic groups that were distinguished.

Data Extraction and Mapping of Studies

Using the classification scheme, we then utilize visual mappings of the results to highlight states in the collected papers. To identify which categories have been emphasized in past research and show possible opportunities for future work, we use three plot types to show maps (i) tables, (ii) bar plots, and (iii) bubble maps. Once the scheme is in place, we used excel spreadsheets to store the data and applied R scripts to extract and categorize the papers. Furthermore, we put rationales to decide why we believe each paper is categorized. Below are the visual techniques and rationale for answering each RQ:

Visual Map of RQ1. To answer RQ1, we show a visual mapping of the contributions (with the researchers and practitioners separately) against the methodologies. We intend to find out how the methodologies influence the contributions and what is the popular combination of contributions and methodologies. A bubble map will be used to show results. The map should show what contributions

are saturated and which perceived contributions have the potential for future work. We will also pick up examples of each classified paper for an in-depth discussion of the maps.

Visual Map of RQ2. To answer RQ2, we show a visual mapping of the replicability of the collected papers. We intend to determine how much CR research has the potential to be replicated. A bar chart will be used to visualize the main results. The map should show the proportion of how many papers can be replicated and show what forms are used to provide replication (i.e., via links or reference to the dataset). For a deeper understanding of the data sources, we perform additional sub-classification of the source: (i) research that extracts data from pure code review tools (e.g., Gerrit tools in OSS and special review systems or tools in industry such as CodeFlow tool in Microsoft), (ii) research that extracts data that not only contains CR, but expands on other software development tools such as mailing lists, version control system, GitHub, and issue tracking system, (iii) research that extracts data from observational experiments in the form of interviews, survey, and control study. Additionally, we classify the platforms where the available datasets are provided into four types: (i) online storage, i.e., dropbox, (ii) permanent storage, i.e., zenodo, (iii) GitHub /BitBucket, and (iv) personal or university.

Visual Map of RQ3. To answer RQ3, we show a visual mapping of the metric benchmark in terms of research aspects (i.e., product, process, and human) and research topics. We intend to formulate a systematic metric group for a future research guide and understand in which topic what kinds of metrics should be included. Two tables will be drawn to present our benchmark details. The first table map should show 1) how many different metrics are used with their frequency in papers and (2) what research aspects do these metrics target. The second table map should show different combinations of metrics are used in different research topics.

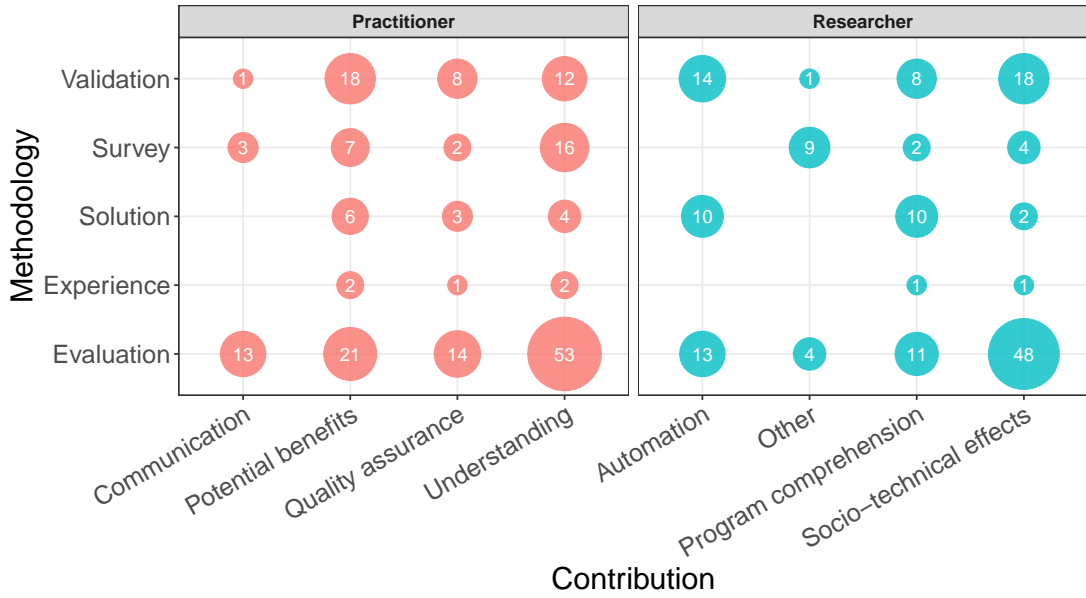


Figure 3.3: Visual Map for RQ₁, showing the contribution and methodology of CR research. The figure shows that evaluation is the most popular methodology, particularly targeting the contributions of understanding and socio-technical effects.

3 Results: Maps of CR Research

The results will answer the research questions, with the visual maps of the categories of the papers.

(RQ1): What contributions and methodologies does CR research target?

Figure 3.3 shows both the saturation of papers as well as the potential research opportunities for the field. Note that a paper can target multiple contributions, using more than one methodology. The figure clearly shows that evaluation is the most popular methodology, benefiting both the practitioners and researchers. For practitioners, most of the papers have contributions to potential benefits and understanding aspects. Potential benefits mean that modern CR provides benefits beyond the fundamental need to find defects. CR is demonstrated to be useful

for other tasks. We introduce three examples in detail below. For the task of improving code style, Zhang et al. [177] presented an interactive approach named *CRITICS* for inspecting systematic changes and the results show that it should improve developer productivity during this process. For the task of increasing the learning, Gousios et al. [57] conducted a large-scale survey to investigate work practices and challenges in pull-based development model and results show that integrator should consider several factors in their decision making. For the task of review comments usefulness, Rahman et al. [118] found that useful comments share more vocabulary with the changed code, contain salient items like relevant code elements, and their reviewers are generally more experienced. For instance, exploring how CR is conducted can be used for practitioners to better implement review activity and improve the review quality. Understanding is when reviewers have prior knowledge of the context and the code, they complete reviews more quickly and provide more valuable feedback to the author. Key examples are researches that look into the quality of review and types of defects. Kononenko et al. [82] provided a deep insight into how developers define review quality, what factors contribute to how they evaluate submitted code, and what challenges they face when performing review tasks. Beller et al. [26] conducted a manual research to increase the understanding of practical benefits that the MCR process produces on reviewed source code. Their results show that types of changes due to the MCR process in OSS are strikingly similar to those in the industry and academic systems.

On the other hand, researcher-oriented CR studies mostly focus on socio-technical contributions. Socio-technical related papers are studies that involve the consideration of both human and technical aspects. One popular topic is reviewers recommendation and many studies have been done on this topic. Xia et al. [167] put textual information and file location analyses together to recommend reviewers more accurately. Hannebauer et al. [63] recommended code reviewers based on their expertise. Apart from reviewer recommendation topic, many other human related researches have been conducted such as review participation Thongtanunam et al. [150], evaluation of contributions Tsay et al. [154] and broadcast during CR process Rigby and Storey [127]. In terms of CR, studies can be designed and carried out to determine if and how team collaboration,

Table 3.4: Top 5 combination of contribution and methodology

		Contribution		
Methodology		Socio-technical effects	Understanding	Potential Benefits
	Evaluation	[11, 24, 25, 26, 34, 36, 44, 46, 49, 55, 56, 67, 69, 71, 72, 74, 75, 76, 81, 83, 84, 99, 100, 101, 103, 105, 114, 118, 126, 127, 129, 130, 134, 138, 140, 147, 148, 149, 150, 153, 154, 159, 167, 172, 178, 183]	[10, 20, 22, 24, 25, 26, 28, 34, 36, 43, 46, 55, 56, 59, 67, 69, 71, 74, 76, 81, 83, 93, 99, 100, 101, 103, 105, 111, 112, 114, 126, 127, 129, 130, 134, 138, 140, 145, 148, 149, 150, 153, 157, 159, 161, 172, 173, 174, 175, 178, 182, 183, 184]	[25, 35, 44, 45, 49, 59, 72, 75, 92, 93, 101, 110, 118, 126, 129, 133, 143, 152, 161, 167, 172]
	Validation	[6, 23, 49, 72, 75, 99, 100, 101, 103, 109, 117, 118, 148, 149, 150, 167, 172]		[7, 15, 17, 23, 49, 63, 72, 75, 79, 101, 102, 109, 117, 118, 167, 172, 176, 177]

coordination, awareness and learning occurs. In terms of opportunities, Figure 3.3 highlights the lack of experience papers. This is crucial and shows a lack of reporting and feedback from developers. Instead, we see that there are a stable number of survey papers. Other notable potential methodologies are the experience and solution, which indicates that more practical tools need to be developed to help practitioners in reality.

Table 3.4 shows a listing of top five paper contributions with the methodologies used, illustrating how evaluation studies are dominant. According to our results, two most popular combinations are the evaluation study that has a understanding contribution (e.g., 53 papers), then followed by the study with socio-technical effect target following the evaluation methodology (e.g., 48 papers). For the understanding target with evaluation methodology, we introduce two representative studies. In the work of Tao et al. [145], they investigate the patch rejected reasons with 300 samples and formulate the practical suggestions for patch author submission. This work helps patch authors to better understand what kinds of patch should be submitted so as to reduce the rejection chance. Another work conducted by Zampetti et al. [173] empirically analyze how developers document pull requests with external references using mixed-method. Their results indicate that for developers, external resources are useful to learn something new or to solve specific problems. One example of the evaluation study with a social-technical effect is Thongtanunam et al. [148], which investigates CR

practices in defective files combined with human factors (e.g., the participation of the reviewer in the process). In detail, authors evaluate the results using a detailed empirical study of the Gerrit review system within the Qt project. Similarly, Kononenko et al. [81] reported on a case study investigating CR quality for Mozilla and explore the relationships between the reviewers' code inspections and a set of factors, both personal and social. It is interesting to note that 36 out of 53 papers, i.e., around 68%, that contribute to better understanding also have socio-technical contributions. For example, Thongtanunam et al. [150] studied what factors influence review participation in the CR process which in turn helps practitioners understand the situation when they tend to join. While within the validation methodology, we find the most popular combination is papers that target contributions of potential benefits with such methodology. The majority of the validation papers are based on recommendation or prediction models. An example of this type of paper is Rahman et al. [117], where authors suggest an approach of reviewer recommendation based on cross-project and technology experience.

Apart from the referred papers using two popular methodologies that are listed in the Table 3.4, we also provide the complete paper list for those papers using survey, solution, and experience methodology. Among these three methodologies, survey is relatively frequent with eighteen papers being retrieved [10, 13, 18, 19, 30, 32, 33, 46, 51, 57, 58, 82, 86, 91, 119, 132, 144, 174]. Thirteen papers are classified as solution [7, 17, 36, 62, 89, 96, 102, 117, 135, 155, 160, 164, 177]. The last experience methodology is the most rare case, only four papers being found [36, 125, 135, 137].

Answering RQ1: Our results show that 65% of CR researches published in premium SE venues use sound evaluation methodology (i.e., 73 papers), targeting particularly socio-technical and understanding of CR processes. However, there is a lack of papers that report the experience and propose solutions to deal with CR problems (i.e., four papers and thirteen papers, separately). The implication of RQ1 is that, we encourage the practitioners to more emphasize and share the experience with CR. At the same time, future research could propose more solution tool support to facilitate the developers to make the CR process more efficient.

(RQ2): How much CR research has the potential for replicability?

We divide all premium papers into different classification according to the definition of data sources shown in Section 2.5. We describe each data source in detail below. In *CR Process*, for example, McIntosh et al. [101] conducted the research to see the impact of code reviews on software quality. They only focus on review process and extract the data from QT, VTK, ITK projects using review tools (e.g., Gerrit). For *Software Development Process*, for example, Kononenko et al. [81] investigated whether people and participation matter the quality of review. In their research, they collected data from issue tracking system (e.g., Bugzilla) which belongs to the development process. In *Interview/Survey/Control Study*, for instance, Bosu et al. [33] analyzed the process aspects and social dynamics of CR from the diverse surveys of Microsoft and other open source projects. In another example, Floyd et al. [50] researched the representation of code in the brain with fMRI study. They involved 29 participants to carry out the controlled experiment and got result feedback. We find that *code review process* related dataset is the most extracted from the well-studied Gerrit tool. One advancement has been the release of the rest API, in which anyone is able to download and collect data on projects. We summarize and draw the top 3 popular projects using Gerrit tools. We observe that for these CR papers, Qt project is the most studied project, with sixteen, fifteen, and twelve papers investigating Qt, OpenStack, and

Android respectively.

Figure 3.4 shows two important findings with the proportion of replicability. The **first** finding is that there are in total 38 papers (i.e., 9 papers with closed data, 23 papers open data, and 6 papers open data/closed data) out of 84 papers (around 45%) that do not provide any access to their datasets. Taking a closer look at the closed data, studies are usually conducted within industries, surveys and control studies. An example of this paper is Balachandran [15], where the authors conducted research on how to reduce human efforts and improve review quality using the data from industry project named VMware. For papers that labeled as open data, the researchers collected data from open source projects but did not share a replication package. For instance, Mirhosseini and Parnin [103] investigated whether or not pull requests encourage developers to upgrade out-of-date dependencies with the data from OSS (i.e., 7,470 projects in GitHub). It could be argued that since the data is open source and available for anyone to download themselves. The **second** finding is that, as shown in Figure 3.4, we observe that 42 papers out of 84 papers (around 50%) released a replication package, either referred to a published dataset or released their own dataset via an online link. For the work of Thongtanunam et al. [149], authors referred to a dataset that was previously published [61] to revisit code ownership and its relationship with software quality. Usually, papers release a link to the dataset. For instance, Baysal et al. [25] shared the dataset link (e.g., WebKit and Blinkin projects). Upon a closer inspection on the dataset platforms of these replication package links, we found that GitHub/Bit Bucket and Personal/University are the most common dataset platforms (i.e., seventeen, respectively). While few researches make their dataset immutable (i.e., Permanent Storage), with three papers being classified. We summarize the available replication datasets with their URL links from these quantitative and mixed-method papers, referred to our Appendix.

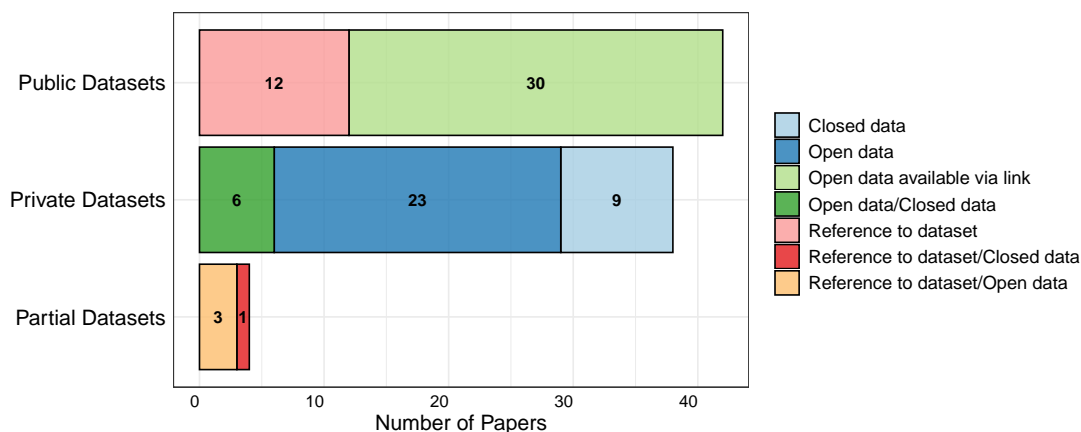


Figure 3.4: Visual Map for RQ₂, showing the replicability of the collected papers. Note that papers analyzed in RQ₂ are limited to quantitative and mixed-method papers. The figure shows that 42 papers (50%) provide the public datasets.

Answering RQ₂: CR research not only relies on the data sources from the CR process but also largely uses the data sources from the software development process (i.e., issue tracking system and GitHub). We observed that 50% of CR papers (i.e., 42 papers out of 84 papers) that use the quantitative method or mixed-method provide the public datasets (i.e., the replication links are provided in the papers). The implication of RQ₂ is that, to promote the validity of scientific findings, we encourage future researches to strive for a replicable dataset.

(RQ3): What metrics and topics are used with CR studies?

Table 3.5 shows sixteen core metrics sets based on 457 identified metrics with their corresponding review aspect and frequency. We summarize three findings. **First**, we observe that **Experience** and **Code** are the two most frequently used metric sets, which are far more than other classified metric sets. In detail, 76 and 72 metrics are involve in **Experience** and **Code** sets respectively. **Experience** is referred to those metrics computing the patch author or reviewer experience in submit-

ting historical patches. Given an example, Ruangwan et al. [130] took four experience related metrics into account: **Reviewer Code Authoring Experience**, **Reviewer Reviewing Experience**, **Patch Author Code Authoring Experience**, **Patch Author Reviewing Experience**. **Code** denotes to those metrics that focus on measuring the source code of a system. For instance, McIntosh and Kamei [99] computed the number of added and deleted code lines. Apart from the general patch size, Maddila et al. [92] proposed in-depth code related metrics such as **Class churn**, **Loop churn**, and **Method churn**. **Others** refers to those metric sets that can not fit to the common metric sets with their definition. **Second**, from the view of the referred paper number, the result indicates that **Code** is considered as the first ranking basic metric set applied to model constructions. 28 out of total 31 papers (i.e., around 90%) take such metric set in to account. The **third** observation is that as we see from Table 3.5, one metric set can represent multiple CR aspects. For example, **File** metric set can either belong to **Product** and **Process**. Those metrics that compute the number of added and deleted files in the patch are regarded as **Process** aspect [49, 56, 75, 178]. On the other hand, researchers calculate the file entropy [75, 99, 150]. Such metrics are more likely to be dynamic and are viewed as **Process** aspect. To answer the correlation between metrics and their corresponding research topics, we summarize nine topics regarding code review and list related papers below:

- *Quality Assurance*: refers to papers focusing on the code quality such as bug fixing [75, 81, 99, 100, 101, 105, 135, 139, 148, 149, 152]
- *Acceptance Predication*: refers to papers focusing on predicting the decision of the patches [49, 74, 83] .
- *Review Process*: refers to papers exploring or comparing the different peer review models [10, 56].
- *Review Participation*: refers to papers focusing on the reviewer participation [130, 138, 150].
- *Review Process Prediction*: refers to papers focusing on predicting the period taken to complete the review [26, 92, 178, 182].

- *Review Process Comments*: refers to papers focusing on predicting usefulness of review comments [118].
- *CI & Review*: refers to papers focusing on the correlation between CI implementation and code review [28, 174].
- *Test & Review*: refers to papers focusing on the correlation between test and code review [140, 141].
- *Technical/Non-Technical & Review* refers to papers investigating the technical and non-technical factor impact on the code review [24, 25, 153].

Figure 3.5 maps the metric sets with nine CR research topics. Two findings are observed based on the related paper list and Figure 3.5. **First**, we find that *Quality Assurance* related topics are more likely to use CR metrics to conduct the research, i.e., eleven papers are identified within this topic. For instance, McIntosh et al. [101] conducted an empirical study to investigate the relationship between post-release defects and code review practices such as coverage, participation, and reviewer expertise. Their findings confirmed the intuition that poorly-reviewed code has a negative impact on software quality. The second popular topic is *Reviewer Process Prediction*, i.e., four papers retrieved, which refers to papers focusing on predicting the review period. One example of the process prediction topic is that in the study of Zhao et al. [182], they applied a couple of eighteen metrics like source code, textual information, experience, and social connection related metrics to recommend pull requests that can be quickly reviewed by reviewers. The **second** finding is that different research topics use particular metric sets. As shown in Figure 3.5, *Review Process Comments* only adopt **Experience**, **Code**, and **Comment** into their statistic model construction. However, for *Quality Assurance*, *Acceptance Predication*, and *Review Participation*, the metric sets are diverse considering comprehensive angles. For instance, the *Review Participation* topic takes almost all kinds of metric sets into accounts except for **language**, **Queue**, and **Log**. In addition, for those specific research topic, the particular metric sets will be computed specially such as **Build related** metric sets used in the *CI & Review* topic. The detailed metrics within each metric set are listed in the Appendix.

Table 3.5: Metric sets used in code review paper

Metric Set	Product	Process	People	Others	Referred papers	# Related metrics	
Experience		✓	✓		[24, 25, 28, 49, 56, 74, 75, 81, 83, 92, 99, 100, 105, 118, 135, 138, 141, 150, 152, 161, 174, 178, 182]	76	
Code	✓	✓			[10, 24, 25, 26, 28, 49, 56, 74, 75, 81, 83, 92, 99, 100, 101, 105, 130, 135, 139, 140, 148, 149, 150, 152, 153, 174, 178, 182]	72	
Ownership			✓		[26, 49, 74, 81, 100, 101, 105, 118, 135, 138, 139, 140, 148, 149, 150, 152, 174]	38	
Comment		✓			[28, 56, 74, 81, 83, 99, 105, 118, 130, 135, 139, 148, 150, 152, 153, 174, 178]	37	
File	✓	✓			[26, 28, 49, 56, 74, 75, 81, 83, 92, 99, 100, 135, 138, 139, 149, 150, 152, 153, 174, 178, 182]	31	
Participant			✓		[10, 26, 56, 74, 75, 81, 83, 92, 99, 100, 105, 130, 135, 138, 139, 148, 149, 153, 182]	31	
Temporal		✓			[10, 28, 74, 75, 99, 105, 135, 139, 141, 148, 150, 152, 174]	26	
Revision	✓	✓			[10, 28, 49, 56, 74, 75, 83, 92, 99, 100, 105, 138, 148, 152, 174, 178, 182]	24	
Description	✓	✓			[26, 28, 49, 75, 92, 150, 152, 174, 182]	21	
Module	✓				[24, 25, 26, 49, 56, 74, 75, 81, 92, 99, 140, 150, 152, 153, 174, 182]	21	
Defect		✓			[10, 24, 25, 101, 139, 150]	14	
Queue		✓			[10, 24, 25, 28, 81, 92]	6	
Workload			✓		[28, 130, 150]	5	
Decision		✓			[74, 92, 138, 178]	5	
Language	✓				[49, 92]	4	
Log		✓			[28]	1	
Email				✓	[74]	11	
Collaboration				✓	[49, 153]	10	
Build related				✓	[174]	4	
Project				✓	[25, 153]	9	
Others				✓	[24, 81, 83, 92, 105, 130, 138, 141, 174]	11	
Total						31	457

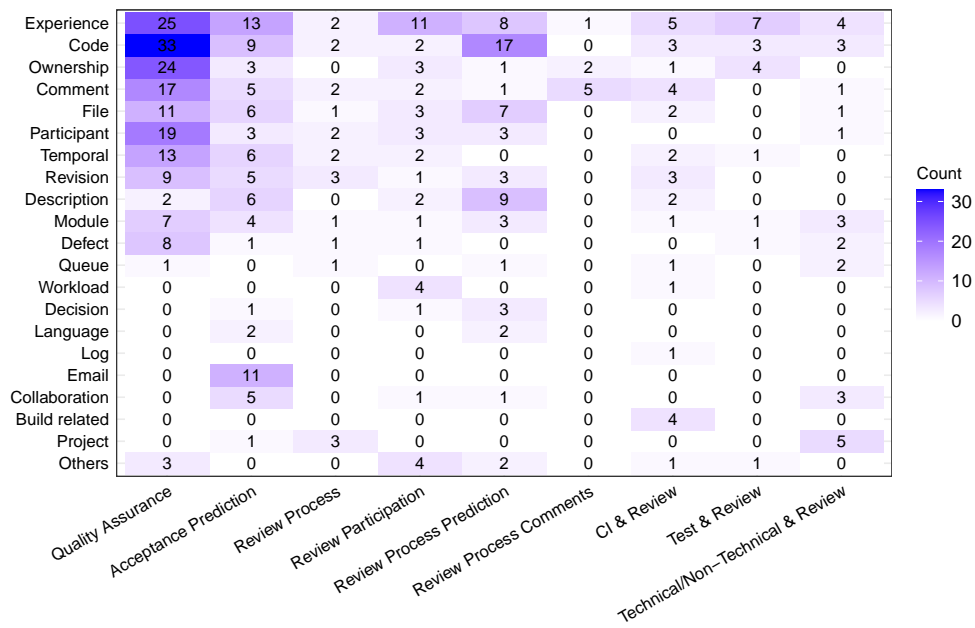


Figure 3.5: Nine research topics with their target metric sets. The figure shows that different research topics tend to target particular metric sets.

Answering RQ3: Sixteen core metrics sets are grouped based on 457 metrics extracted from the quantitative papers, and nine research topics that use these metrics are classified (quality assurance, review process prediction, acceptance prediction, and so on). We observe that the SE topic of quality assurance is more likely to use CR metrics to conduct the research, i.e., eleven papers. In addition, experience and code are the two most frequently used metric sets. From the mapping between metric sets and research topics, we find that it has the potential to benchmark the metric based CR research, as different research topics tend to use particular metric sets. The implication of RQ3 is that, we encourage the researchers to take the existing metrics into account when conducting a certain topic.

4 Comparative Analysis

In this section, we followed the protocol of comparative analysis provided by the work of Petersen et al. [116] to compare and highlight the novelty of our work against existing systematic reviews. The systematic review studies were identified using the following search string: “*systematic review*” AND “*code review*” and by searching a broad collection source (i.e., ACM Digital Library, IEEE Xplore, Science Direct, and SpringerLink databases). We exclude papers that did not explicitly state in the title or abstract that they were systematic reviews or were not published in the traditional SE domain based venues. The search results in a total of three systematic reviews [14, 42, 80] before or in 2019.

For each of the three CR systematic reviews, we characterize them based on their research goals, criteria for inclusion requirements, the number of papers, and means of analysis. Table 3.6 shows the characteristics of the three existing systematic reviews. Below, we now discuss the differences between our study and the existing systematic reviews in detail, with the aspect of research goals, the process, and the breadth and depth:

- **Difference in Research Goals:** The systematic review conducted by Kollanus and Koskinen [80] aims at reviewing how the software inspection field

Table 3.6: Existing Systematic Review Characteristics

Reference Systematic Reviews	Kollanus and Koskinen [80]	Badampudi et al. [14]	Coelho et al. [42]
Research Goals			
Identify Best and Typical Practices	✓	✓	
Classification and Taxonomy	✓		✓
Emphasis on Topic Categories	✓	✓	
Identify Publication Fora	✓	✓	✓
Inclusion Requirements			
Research is Within Focus Area	✓	✓	✓
Empirical Methods Used	✓	✓	✓
Number of Included Articles			
Potentially Relevant Studies	229	873	-
Relevant Studies (Included)	153 (1980-2008)	177 (2005-2018)	13 (2007-2018)
Means of Analysis			
Meta Study	✓	✓	✓
Comparative Analysis			
Thematic Analysis	✓	✓	
Narrative Summary	✓		

has evolved between 1980 and 2008. Their focus is set in the context of the software inspection. However, our mapping study addresses the contemporary tool-based code review, which is a light variant of the software inspection and has been widely adopted in both industrial and open-source projects. Although Badampudi et al. [14] and Coelho et al. [42] conducted the systematic review related to contemporary tool-based code review, the goal of our mapping study is also different from them. The main goal of the work by Badampudi et al. [14] is to observe the evolution of the research topic, while the work by Coelho et al. [42] is to gather evidence on the extent of the work related to refactoring-awareness during code review. Differently, our study aims at investigating the potential of benchmarking in the aspect of datasets and metrics. We believe that a common benchmark could facilitate future CR related researches and help researchers to propose new approaches and compare against existing ones.

- **Difference in Process:** We observe two main differences in the process when compared to the two systematic reviews related to the tool-based review. First, compared to the work of Coelho et al. [42], we include the thematic analysis (i.e., classification schema of methodologies, contributions,

data, and metrics). For the systematic mapping study, thematic analysis is an interesting analysis method, which helps to see which categories are well covered in terms of number of publications [116]. Second, compared to both work by Coelho et al. [42] and Badampudi et al. [14], we conduct an in-depth narrative summary analysis with qualitative review of each paper, as both of them are served as a preliminary study.

- **Difference in Breadth and Depth:** Two differences are summarized, based on the systematic reviews related to the tool-based review. On the one hand, Coelho et al. [42] focused on the specific theme of tool-based code review, i.e., refactoring-awareness. However, our study covers all potential themes. On the other hand, in the study of Badampudi et al. [14], they extracted CR related papers from all possible venues (i.e., 177 papers are retrieved between 2005 and 2018). While to ensure the paper quality and form a best representative view for future researches, we only focus on the papers that are published in the premium venues (i.e., 112 papers are retrieved between 2011 and 2019).

5 Towards a Common Benchmark of Dataset and Metric

Although our results suggest that CR research is mostly driven by empirical evaluation, we conclude that at this stage, we cannot benchmark CR studies. However, the existing datasets and metrics do show potential for creating a benchmark. With the rise of machine learning and AI techniques, CR researchers will soon need to agree on the common set of metrics that should be included to accurately compare such techniques against each other. Having a benchmark will facilitate new researchers, including experts from other fields, to innovate new techniques and build on top of already established methodologies. This mapping commonalities between metrics and datasets is shown in Table 3.7.

Table 3.7 suggests that there exists a regular group of metric set combinations commonly used for papers that are addressing a specific SE topic. We selected metrics that are commonly mentioned in more than 60% of the classified papers

Table 3.7: A summary of common metric sets and datasets used in various SE topics.

CR Topic	# Papers	Common Metric Sets	Common Datasets	Review Settings
Quality Assurance	11	Code (100%), Ownership (82%), File, Participant (73%), Experience (64%)	18%	Gerrit (8), Pull-based (0), Others (3)
Review Process Prediction	4	Code (100%), Participant, File, Module, Description, Experience, Revision (75%)	0	Gerrit (1), Pull-based (3), Others (0)
Acceptance Prediction	3	Code, Experience, File (100%), Comment, Ownership (66%)	0	Gerrit (1), Pull-based (1), Others (1)
Review Participation	3	Experience (100%), Code, File, Workload, Participant, Comment (66%)	0	Gerrit (2), Pull-based (1), Others (0)
Review Process	2	Code, Revision, Participant (100%)	0	Gerrit (0), Pull-based (1), Others (1)
CI & Review	2	Code, Comment, Description, Experience, File, Revision, Temporal (100%)	0	Gerrit (0), Pull-based (2), Others (0)
Test & Review	2	-	0	Gerrit (1), Pull-based (0), Others (1)
Technical/Non-Technical & Review	3	Code, Module (100%), Defect, Experience, Defect (66%)	67%	Gerrit (0), Pull-based (1), Others (2)

(i.e., from RQ3). For instance, in *Quality Assurance* related papers, *Code* is computed for all (i.e., 100%) and around 82% of papers take *Ownership* into account. In *Acceptance Prediction* related papers, all three papers compute the metrics of *Code*, *File*, and *Experience*. On the other hand, Table 3.7 also shows that common datasets were not commonly adopted by researchers. For instance, since two of the three papers in *Technical/Non-Technical and Review* were written by the same authors, the ratio for having a common dataset is high (i.e., 67%). This means that researchers from different groups prefer to construct their own datasets to conduct their study. Another reason is because the technology used to generate datasets constantly evolve, thus, deeming any prior datasets as being outdated. Furthermore, since more datasets are taken from either the Gerrit or GitHub Pull-request API, they sometimes miss the essential elements needed for a specific study. This process can be time-consuming, and could be easily resolved by using a benchmark. We also find that different review settings (i.e., Gerrit and Pull-based) have different emphases on the SE topics. For example, *Quality Assurance* datasets are almost from Gerrit while in *CI and Review* datasets are all from Pull-based review settings. One possible reason is that some specific metric sets are not easily available to be retrieved from different review settings.

6 Threats To Validity

I now discuss threats to the validity of our mapping study.

External validity. External validity is concerned with our ability to generalize based on our results. The results of this mapping study are considered with regard to the CR domain, while the validity of conclusions is applicable only to the CR context. The external validity threats are thus not applicable.

Construct validity. Construct validity is concerned with the degree to which our measurements capture what we aim to study. During the qualitative analysis, especially for the methodology and the contribution classification, methodologies and contributions may be miscoded due to the subjective nature of our coding approach. To mitigate this threat, three co-authors sat in a round-table and

did the classification. If a dispute occurred, the full contents of the papers were discussed before the consensus was reached.

Internal validity. Internal validity is the approximate truth about inferences regarding cause-effect or causal relationships. We summarize three potential internal threats. The first threat is related to the venue selection. In this mapping study, we only consider 34 top venues considering their online citation indices and feedback from the software engineering community, similar to the work of Mathew et al. [98]. Thus, there will always be a venue missing from such a study and can be considered. However, we believe these 34 top venues can represent the best practice for the CR research. The second threat is related to the paper selection of the studies during the screening process. Due to the large amount of hits from our search string, our initial step includes the first author scanning through and discarding papers based on titles and abstracts, which potentially raises a bias in the paper selection. Nevertheless, we are confident of this threat, as the first author is an existing code review researcher and is familiar with the domain. The third possible internal threat is with regard to the terms that are used in our search string. The case might exist that the search string will not cover all terms. To reduce such risk, in the initial round, we manually checked twenty CR related papers to group the term candidates, and we were confident that the existing search term is sufficient.

Conclusion validity. Conclusion validity is the degree to which conclusions we reach about relationships in our data are reasonable. In the case of our datasets, there is a threat that our grouping is not accurate. Since there is no related work that has similar results, we cannot verify our findings. To mitigate this, we rely on the systematic guidelines for our outcomes. In addition, we publish a website and open it to both researchers and practitioners to criticize or add to our results.

7 Summary

Code review (CR), as a well-known practice, plays a vital role in software quality assurance. In the recent decade, the contemporary review tools have made the

review process now being light-weight and have been widely adopted in both open-source and industrial projects. Due to the availability of datasets brought by these review tools, CR related researches are largely carried out. In order to understand the state-of-the-art practices, we first conduct a systematic mapping study to provide a visual summary of the benchmark potential within the CR domain through 112 papers that are published in premium conferences and journals.

Three main maps are visualized. First, concerning the map between methodologies and contributions, we find that 65% of CR researches use sound evaluation methodology (i.e., 73 papers), targeting particularly socio-technical and understanding of CR. While, there is a lack of papers that report the experience and propose solutions to deal with CR problems (i.e., four papers and thirteen papers, separately). Second, concerning the map of the datasets, our results show that few researches provide replicable datasets, i.e., around 50%. Third, concerning the map of metrics, we identify 457 metrics which are grouped into sixteen core metrics sets, and we observe that the SE topic of quality assurance is more likely to use CR metrics to conduct the research, with eleven papers being classified. Additionally, we find that different research topics use particular metric sets, which provides the potential for a benchmark.

The next step is the creation of a benchmark to facilitate future research against the state-of-the-art. With the rise of machine learning and AI techniques, a common benchmark is needed as it will facilitate CR researchers to accurately compare techniques against each other and propose new approaches. To encourage this framework construction, we provide a listing of methodologies and contribution, 42 public datasets, and 457 identified metrics across nine research topics which is available at <https://naist-se.github.io/code-review/>.

Part II

Link Sharing in Code Review

4 | Understanding Shared Links and Their Intentions to Meet Information Needs

In the previous chapter, through the mapping study, the literature reviews reveal that it is challenging to identify and acquire needed information in the context of code review research field. Hence, in this chapter, an empirical study is performed to gain a deeper understanding of the role of sharing links in fulfilling information needs during code review process.

1 Introduction

Software code reviews serve as quality assurance for software teams [70, 127]. From being a formal code inspection process conducted by face-to-face meetings [47], nowadays the Modern Code Review (MCR) process becomes more flexible with asynchronous collaboration through an online tool (such as Gerrit¹, Codestriker², and ReviewBoard³). These online tools are now widely adopted in both open source and proprietary software projects [126, 132]. Not only improving the overall quality of a patch (i.e., software changes), Bacchelli and Bird [13] also reported that MCR also serves as an effective mechanism to increase awareness and share information: “Code reviews are good FYIs [for ythe information].”

¹<https://www.gerritcodereview.com/>

²<http://codestriker.steceforge.net/>

³<https://www.reviewboard.org/>

An effective review requires proper understanding. However, it is challenging to identify and acquire the needed information to have a proper understanding to conduct a review. This is especially a case for a large software project like OpenStack, which has code reviews of over 20 million lines of codes that are submitted by over 100,000 contributors spread more than 600 code repositories [181]. Pascarella et al. [113] find that during reviews, reviewers often request additional information about correct understanding, alternative solution, to improve patch quality. Ebert et al. [46] report that reviewers often suffer from confusion due to a lack of information about the intention of a patch (i.e., a rationale for a change).

Recent work points out that the link shared in the review discussion can be used to provide information related to a review. Hirao et al. [69] show that shared links between reviews can be used to indicate the information about patch dependency, broader context, and alternative solution. However, the other types of links other than review links were not studied in the work of Hirao et al. [69]. On the other hand, Jiang et al. [73] conducted a quantitative analysis to find developers that share various types of links in ten GitHub projects. However, this study does not systematically investigate the correlation between the shared links and the review process, and lacks of qualitative analysis of why these various types of links are shared.

Based on the findings of prior work (Hirao et al., 2019), we hypothesize that sharing links may help developers fulfill the information needs. Yet, it is still unclear about what types of information could be fulfilled by these various types of links. To fill this gap, this work aims to explore the prevalence of link sharing, systematically investigate the correlation between link sharing and review time, and qualitatively investigate what are the intentions of sharing links. In this paper, the intention is defined as the intention of sharing a link to meet a certain type of information needed during code review. Through a case study of the OpenStack and Qt projects (two large-scale and thriving open sthce projects with globally distributed teams that actively perform code reviews), we identify 19,268 reviews that have 34,264 links shared during review discussions. We formulate three research questions to guide the study:

- **RQ1: To what extent do developers share links in the review discussion?** It is not yet known how a project of repositories uses link

sharing in their communities. Using a sample of well-known OpenStack and Qt projects, we would like to investigate the trend of link sharing, common domains in the project, and the link target types.

- **RQ2: Does the number of links shared in the review discussion correlate with review time?** Prior studies [25, 83] analyzed the impact of technical and non-technical factors on the review process (e.g., review outcome, review time). However, little is known about whether or not the practice of sharing links can be correlated with review time. It is possible that link sharing may shorten the review time as it provides the required information to a review, which might help reviewers to conduct a review faster. To address this RQ, we conduct a statistical analysis using a non-linear regression model to analyze a correlation between link sharing and review time.
- **RQ3: What are the common intentions of links shared in the review discussion?** Previous work [113] has identified different types of information that are needed by reviewers when conducting a review. Yet, little is known to what extent can link sharing meets such information needs. Hence, we aim to investigate the intention behind link sharing in order to better understand the role and usefulness of link sharing during reviews.

The key results of each RQ are as follow: *For RQ1*, the results show that in the past five years, 25% and 20% of the reviews have at least one link shared in a review discussion within the OpenStack and Qt. 93% and 80% of shared links are the internal links that are directly related to the project. Importantly, although the majority of the internal links are referencing to reviews, bug reports, and source code are also shared in review discussions. *For RQ2*, the non-linear regression model results show that the internal link has a significant correlation with the code review time. However, the external link is not significantly correlated, for OpenStack and Qt. Furthermore, we observe that the number of internal links has an increasing relationship with the review time. *For RQ3*, we identify seven intentions of sharing links: (1) Providing Context, (2) Elaborating, (3) Clarifying, (4) Explaining Necessity, (5) Proposing Improvement, (6) Suggesting Experts, and (7) Informing Splitted Request. Specifically, for the internal links,

we observe that the most popular intention is to provide context. While for the external links, to elaborate the review discussions (i.e., provide a reference or related information) is the most common intention.

The results lead us to conclude that link sharing is increasingly used as a mechanism of sharing information in a code review process, the number of internal links has a positive correlation with the review time, and the intention of link sharing is often used to provide context understanding. *For patch authors*, they should provide a clear context of the patch by sharing links (i.e., containing implementation related information) for reviewer teams to better understand a patch. *For review teams*, link sharing should be encouraged, as results indicate that it can fulfill information needs and contains crucial knowledge to assist the author, which will support a more efficient review process. *For researchers*, with the increasing usage of links, there is an opportunity for tool support for suggesting shared links to retrieve useful information for both patch authors and review teams. The study contributions are three-fold: (i) a large quantitative and qualitative study on link sharing on the MCR process, (ii) a taxonomy of seven intentions of sharing links, and (iii) a full replication of the study, including the scripts and datasets. ⁴

1.1 Chapter Organization

The remainder of this chapter is organized as follows: Section 2 introduces the background of the study. Section 3 describes the studied projects, the data preparation, and the analysis approach for each research question. Section 4 reports the results of the empirical study. Section 5 discusses the implications from the findings. Section 6 discloses the threats to validity. Finally, we conclude the paper in Section 7.

2 Motivating Example

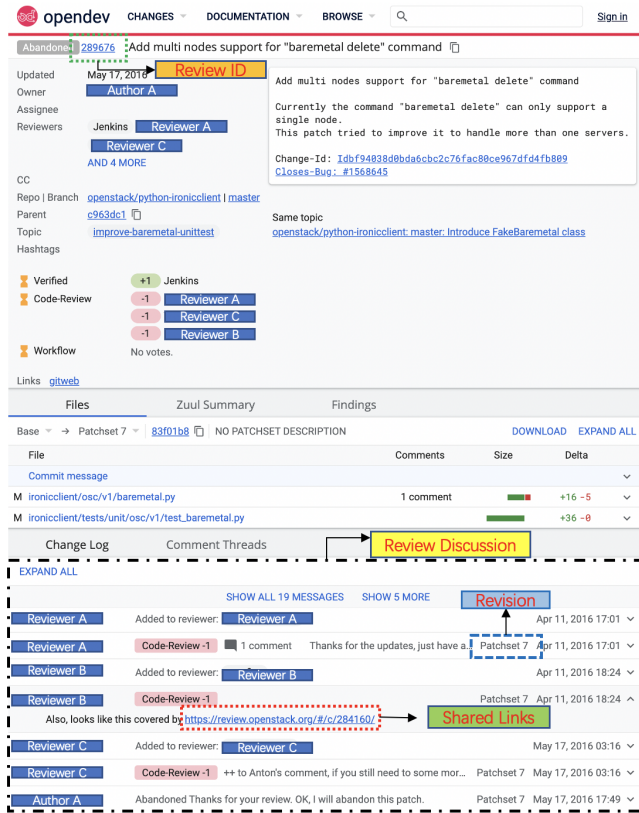
We highlight two challenges in identifying information needs in Code Review. The first challenge is that the rationale to meet information needs is unclear. Prior

⁴<https://github.com/NAIST-SE/LinkIntentioninCR/>

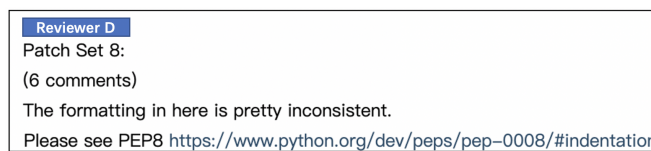
studies found that missing a rationale and a lack of familiarity with existing code (e.g., a lack of knowledge about the code that's being modified) are the most prevalent reasons for causing discussion confusion and low reviewer participation in code reviews [46, 130]. Such confusion can delay the incorporation of a code change into a code base. Patch description is another vital information to help developers understand the changes. Tao et al. [144] stated that one of the most important pieces of information for reviewers is a description of the rationale of a change. In addition, the description length shares an increasing relationship with the likelihood of reviewer participation [150]. The second challenge is understanding which information is needed to facilitate the review. As reported by Bacchelli and Bird [13], understanding is the main challenge for developers when doing code reviews. The most difficult task from the understanding perspective is finding defects, immediately followed by alternative solutions. They point out that context and change understanding must be improved if managers and developers want to match their needs. Recently, Pascarella et al. [113] highlights the presence of seven types of high-level information needs, with the two most prominent being the needs to know (1) whether a proposed alternative solution is valid and (2) whether the understanding of the reviewer about the code under review is correct.

To help developers find related changes, the code review tool like Gerrit has provided functionalities that allow a patch author to share the links of reviews that have related changes or in the same topics.⁵ Yet, we observe that developers still share links in the review discussion. Figure 4.1 shows three motivating examples of how sharing links in the review discussion can be a means to fill in the needed information. The first observation is that various links can be shared in a review discussion to provide information. For instance, as shown in Figure 4.1(a), the reviewer *Anton Arefiev* shared a link of a review in the PatchSet 7 (i.e., the seventh revision of the review #289676) to inform the patch author of the review #289676 that the review #284160 covered the current review #289676. Figure 4.1(b) shows another example where the reviewer *melanle witt* shared a link of Python documentation in a review discussion of the review #207794 in order to improve the coding format.

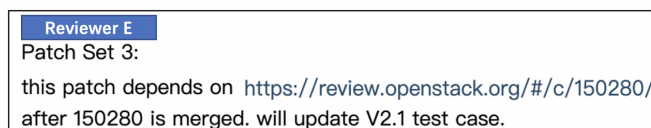
⁵<https://gerrit-review.googlesource.com/Documentation/user-review-ui.html>



(a) Review comment with a review related link in the review #289676 from OpenStack.



(b) Review comment with a python related link in the review #207794 from OpenStack.



(c) Review comment with a review related link in the review #150718 from OpenStack.

Figure 4.1: Motivating examples of link sharing in MCR process.

While the work of Hirao et al. (2019) extensively investigate the review links shared in code reviews, these motivating examples show that links that point to other information stheces are also shared in code reviews. Hence, in this study, we aim to investigate the trend and characteristics of shared links in terms of their types (internal or external) and the kinds of content to which those links point. In this study, we define ‘internal links’ are the links that are directly related to the project (e.g., review links, bug reports, git repository), while external links are the links that refer to restheces outside the project (e.g., Python Document).

Apart from the various links, the second observation is that the intentions behind sharing links can be different even if the shared links have the same type. In Figure 4.1(a), the intention of the reviewer of sharing a review link is to point out that the current review #289696 is no longer necessary. While, in Figure 4.1(c), although the reviewer also shared a link to the review #150718, the intention of the reviewer is to help the patch authors clearly understand the context and code dependency of the current patch.

While the work of Jiang et al. [73] investigate the different types of links shared in pull-based review, the motivating examples show that the intention of providing information can be different even though the types of shared links are the same. Thus, we aim to better understand the intentions behind link sharing and to what extent can link sharing meet the information needs of review teams.

3 Case Study Design

In this section, we describe the studied projects and data preparation. Then we present the analysis approach for each research question.

Studied Projects

Since we want to study the practice of link sharing, we focus on the projects that use a code review tool. In this study, we select the projects that use the well-known **Gerrit** platform, a review tool that is largely adopted by many open sthece projects, where the review data is accessible through REST API. From the range of open sthece projects that are listed in the work of Thongtanunam and Hassan [146], we start with fthe projects: OpenStack, Qt, LibreOffice, and

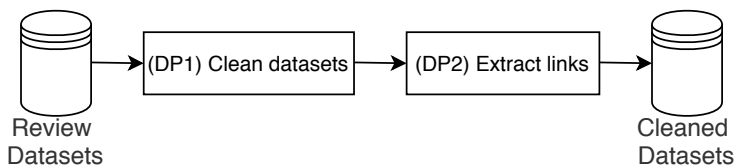


Figure 4.2: An overview of data preparation.

Table 4.1: Studied projects.

	OpenStack	Qt
Studied Period	11/2011-07/2019	05/2011-07/2019
# Reviews (#Merged/#Abandoned)	58,212 (45,439/12,773)	40,758 (35,284/5,474)
# Reviewers	4,568	1,123
# Reviews with Links	14,655 (25.2%)	4,613 (11.3%)
# Unique Links	26,746	7,518
# Total Links	31,698	7,988
# Links per Review (1st Qu./Median/3rd Qu.)	1/1/2	1/1/2
Percent of Links Shared by Reviewers	62.3%	44.0%
Percent of Links Shared by Authors	37.6%	56.0%

Chromium, as these fthe projects actively perform code reviews through Gerrit. However, we observe that a large proportion of LibreOffice reviews have only one reviewer. For the Chromium project, we find that it is not trivial to analyze the shared links based on their domains (i.e., most are under the google.com domain), since we want to investigate whether the link is external or not. To gain more insights and avoid potential errors, we exclude LibreOffice and Chromium from the study. Therefore, in this paper, we perform a case study on OpenStack and Qt projects.

OpenStack is an open sthce software project where many well-known organizations and companies, e.g., IBM, VMware, and NEC, collaboratively develop a platform for cloud computing. Qt project is developed for creating graphical user interfaces as well as cross-platform applications that run on various software and hardware platforms, such as Linux, and Windows.

Data Preparation

Figure 4.2 describes an overview of the data preparation. the data preparation process (DP) consists of two steps: (DP1) clean dataset and (DP2) extract links.

(DP1) Clean dataset. For two studied projects, we use the review datasets from the work of Thongtanunam and Hassan [146]. In order to study the correlation between review process duration and the link, we only include the reviews whose status are abandoned or merged. we exclude reviews with open status, since we can not calculate the review time of these reviews. Since we want to investigate the trend of links that are shared in review discussions, we exclude the comments that are posted by automated tools in the discussion thread. We refer to the documentation of the studied systems⁶ to identify the automated tools that are integrated with the code review tools. In addition, we exclude the reviews that do not have comments posted by the reviewers (or have only comments posted by the patch author). Table 1 shows the number of remaining reviews that are studied in this work. For OpenStack, 58,212 reviews are captured from November 2011 to July 2019. Qt owns 40,758 reviews from May 2011 to July 2019.

(DP2) Extract links. To identify the links in the review discussion, we apply regular expression (i.e., ‘`https?://\S+`’) to search for hyper links in review discussions. Finally, as shown in Table 4.1, we are able to collect 14,655 reviews with 31,698 links and 4,613 reviews including 7,988 links for OpenStack and Qt projects, respectively. Table 1 provides summary statistics of links per review and whether the links are shared by a patch author or a reviewer. More specifically, in the datasets, 37.6% of links in OpenStack are shared by the patch authors, while 62.3% of links are shared by the reviewers. For Qt, 44% of links are shared by the patch authors, while 56% of links are shared by the patch authors.

⁶<https://docs.openstack.org/infra/manual/developers.html>, and [https://wiki.qt.io/CI Overview](https://wiki.qt.io/CI%20Overview)

RQ1 Analysis

To answer RQ1: To what extent do developers share links in the review discussion?, we analyze to which extent of links are shared in three main aspects: (1) the trend of link sharing, (2) the common domains, and (3) the types of link targets. Below, we describe the analysis approach for each aspect.

Link Sharing Trend. To investigate the trend of link sharing, we examine how often reviews will have link sharing overtime. Similar to prior work [69], we measure the proportion of reviews that have at least one link shared in the review discussion in an interval of three months.

Common Domain in the Project. To analyze the domain popularity, we first determine whether the links are internal (i.e., the links that are directly related to the project), or external (i.e., the links that refer to the restheces outside the projects). To identify whether the links are internal or external, we manually examine the domain name and its homepage to determine the links are directly related to the projects (i.e., internal links). More specifically, for OpenStack, we determine the links with the domain names that have the ‘openstack’, ‘opendev’, keywords (e.g., <https://wiki.openstack.org/>) as internal links of the OpenStack project. We also consider the links under “<https://blueprints.launchpad.net/openstack>” and “<https://github.com/openstack>” as the internal links of the OpenStack project. Similarly, for Qt, we consider the links with the domain name that has the ‘qt’ keyword as internal links of the Qt project. We also consider the links under “<http://github.com/qt/>” as internal links of Qt. Links that are not identified as internal links will be identified as external links. Once we identify whether the links are internal or external, we examine the popular domain for internal and external links. To do so, we measure the frequency of links in each domain.

Link Target Types. To understand what kinds of link targets are referenced in review discussions, we perform a manual analysis on a statistically representative sample of the link dataset.

- (I) *Representative dataset construction.* As the full set of the constructed data is too large to manually examine their link targets, we then draw a statistically representative sample. The required sample size was calculated so that the conclusions about the ratio of links with a specific characteristic would generalize to all links in the same bucket with a confidence level of 95% and a confidence interval of 5.⁷ The calculation of statistically significant sample sizes based on population size, confidence interval, and confidence level is well established [85]. We randomly sample 379 internal links and 327 external links from the unique links of the OpenStack project, and 363 internal links and 309 external links of the Qt project. To remove the threat of links being inaccessible (404), the approach includes verifying each link until the sample size is reached. To do so, we first randomly select 500 internal candidate links and 500 external candidate links for each studied project. Then we automatically verified and filtered out links that are inaccessible. In the end, we filtered out 70 inaccessible internal links and 138 inaccessible external links for the OpenStack project, and 75 inaccessible internal links and 147 inaccessible external links for the Qt project.
- (II) *Manual coding.* To classify the type of link targets, we perform two iterations of manual coding. In the first iteration, the first two authors independently coded 50 internal and external links in the sample. The initial codes were based on the coding scheme of Hata et al. (2019) which provides the 14 types of link targets in sthce code comments. However, we found that their codes did not cover all link targets in the datasets. Hence, the following five codes emerged from the manual analysis in the first iteration:
 - *Communication channel:* links target for the mailing list, chat room.
 - *Github activity:* links target for pull requests, commits, and issues.
 - *Media:* links target for pictures and videos.
 - *Memo:* links target for the personally recorded documentation.
 - *Review:* links target for the code review.

⁷<https://www.surveysystem.com/sscalc.htm>

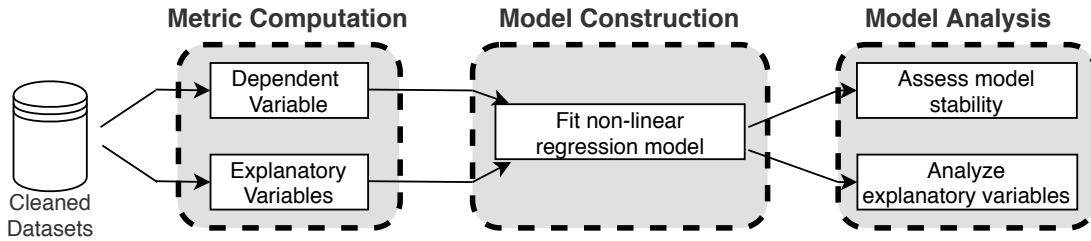


Figure 4.3: An overview of the RQ2 quantitative analysis.

To validate the codes, we perform a second iteration of manual coding. In this iteration, the three authors of this paper independently coded another 30 internal and external links in the sample. Then, we measure the inter-rater agreement using Cohen’s Kappa across the 19 types of link targets. The score of the Kappa agreement is 0.83, which is implied as nearly perfect [156]. Based on this encouraging result, we divided the remaining samples into two sets. Then, the first author independently coded the first set and the second author independently coded the second set.

RQ2 Analysis

To answer RQ2: Does the number of links shared in the review discussion correlate with review time?, we perform a statistical analysis using a non-linear regression model to investigate the correlation between the link sharing and the review process (i.e., review time), while consider several confounding variables. Similar to the prior studies [130, 150], the goal of the statistical analysis is not to predict the review time, but to understand the associations between the link sharing and the review time. In this section, we first describe the selected explanatory variables, then we describe the model construction, and finally, I explain how we analyze the model. Figure 4.3 presents an overview of the RQ2 quantitative analysis.

Explanatory Variables. Table 2 describes the 14 metrics that are used as explanatory variables in the model. Since we want to investigate the correlation between link sharing and the code review time, we count the number of internal links, external links, and the total links. As prior studies have shown that several

Table 4.2: The studied explanatory variables.

Confounding variables	Description
Add	The number of added lines by a patch.
Delete	The number of deleted lines by a patch.
Patch size	The total number of added and deleted lines by a patch.
Purpose	The purpose of a patch, i.e., bug, document, feature.
# Files	The number of files what were changed by a patch.
# Revisions	The number of review iterations.
Patch author experience	The number of prior patches that were submitted by the patch author.
# Comments	The number of messages posted in a review discussion by reviewers and the patch authors, excluding messages for change updates and the number of inline comments.
# Author comments	The number of messages posted in a review discussion by the patch author, excluding messages for change updates and the number of inline comments.
# Reviewer comments	The number of messages posted in a review discussion by reviewers, excluding messages for change updates and the number of inline comments.
# Reviewers	The number of developers who posted a comment to a review discussion.
Link sharing variables	Description
# External links	The number of external links shared in the general discussion.
# Internal links	The number of internal links shared in the general discussion.
# Total links	The number of internal and external links shared in the general discussion.

factors can have an impact on the review time [83, 149], we also include 11 variables shown in Table 2 into the model. Similar to the prior work [100, 150], we classify a patch where its description contains ‘doc’, ‘copyright’, or ‘license’ words as documentation, while a patch where its description contains ‘fix’, ‘bug’, or ‘defect’ words is classified as bug fixing. The remaining patches are classified as feature introduction.

For the dependent variable (i.e., review time), we measure the time interval in hthes from the time when the first comment was posted until the time when the last comment was posted. We did not include the time between the patch was submitted and the first comment was made because this period of time could be the review waiting time of a patch, which is not a key focus of this study.

Model Construction (MC). To investigate the association between link sharing and review time, we choose the Ordinary Least Squares (OLS) multiple regression model. This technique allows us to fit the nonlinear relationship between the explanatory variables and the dependent variable. We adopt the model construction approach of Harrell Jr. et al. (1984), which was also used by McIntosh et al. [101]. This construction approach also enables a more accurate and robust fit of the dataset, while carefully considering the potential for over-fitting. The model construction approach consists of five steps and we explain below:

(MC1) Estimate budget for degrees of freedom. As suggested by Harrell Jr. et al. [64], we estimate a budget for the model before fitting the regression model, i.e., the maximum number of degrees of freedom that we can spend. We spend no more than $\frac{n}{15}$ degrees of freedom in the OLS model, where n refers to the number of studied reviews in the dataset.

(MC2) Normality adjustment. OLS expects that the response variable (i.e., review time) is normally distributed. Since software engineering data is often skewed, we analyze the distribution of review time in each studied project before fitting the model. We use `skewness` and `kurtosis` function of the `moments` R package to check whether or not the modeled dataset is skewed. If the distribution of review time is skewed (i.e., `p_value` < 0.05), similar to prior work [101], we use a log transformation to lessen the skew in order to better fit the assumption of the OLS technique.

(MC3) Correlation and redundancy analysis. Highly correlated explanatory variables can interfere with each other when examining the significance of the relationship between each explanatory variable and the response variable, which potentially leads to spurious conclusions. Hence, we use the Spearman rank correlation (ρ) to assess the correlation between each pair of metrics. We repeat this process until the Spearman correlation coefficient values of all pairs of metrics are less than 0.7. Although correlation analysis reduces collinearity among the explanatory variables, it may not detect redundant variables (i.e., an explanatory variable that does not have a unique signal from other explanatory variables). To assure that studied variables provide a unique signal, we use the `redun` function of the `rms` R package to detect the redundant variables and remove them from the models.

(MC4) Allocating degrees of freedom. After removing highly correlated and redundant variables, we consider how to allocate degrees of freedom to the remaining variables most effectively. Similar to prior work [101], we use the `spearman2` function of the `rms` R package to calculate the Spearman multiple ρ^2 between the explanatory and response variables. The larger Spearman multiple ρ^2 denotes to the higher potential of sharing a nonlinear relationship. Thus, variables with larger ρ^2 values are allocated more degrees of freedom than variables with smaller ρ^2 values. To avoid the over-fitting issue, we only allocate three to five degrees of freedom to those variables with high ρ^2 values and allocate one degree of freedom (i.e., a linear relationship) to variables with low ρ^2 values.

(MC5) Fitting statistical models. Once we decide the allocation of freedom degrees to the variables, we construct a non-linear multiple regression model. Similar to prior work [101], we use restricted cubic splines which force the tails of the first and last degrees of freedom to be linear, to fit the modeled dataset. We use the `rcs` function of the `rms` R package to assign the allocated degree of freedom to each explanatory variable. Then, we use the `ols` function of the `rms` R package to construct the model.

Model Analysis (MA). After the model construction, we assess the goodness of fitting the models and examine the relationship between the review time and

the explanatory variables especially for the number of internal and external links shared in the review discussions. We analyze the model using the three steps: (1) assessing the goodness of fit and model stability, (2) estimating the power of explanatory variables, and (3) examining the relationship between the explanatory variables and the review time. We describe each step in detail below:

(MA1) Assessing model stability. We use the adjusted R^2 [65] to evaluate how well the model fits the dataset based on the studied metrics. However, the adjusted R^2 can be overestimated if the model is overfit to the dataset. Hence, we use the bootstrap validation approach to estimate the optimism of the adjusted R^2 . To do so, we first generate a bootstrap sample, i.e., a sample with replacement from the original dataset. Then, we construct a model using the bootstrap sample (i.e., a bootstrap model). The optimism is a difference in the adjusted R^2 values between the bootstrap model when applied to the original R^2 optimism. Finally, we subtract the average R^2 optimism from the initial adjusted R^2 value to obtain the optimism-reduced adjusted R^2 .

(MA2) Estimating the power of explanatory variables. To identify the variables that are highly correlated with the review time, we estimate the power of explanatory variables that contribute to the fit of the model. Similar to prior work [101], we use Wald χ^2 maximum likelihood tests to jointly test a set of model terms for each explanatory variable since these variables are allocated more than one degrees of freedom. The larger the χ^2 of an explanatory variable is, the larger the contribution that the variable made to the model. we use the `anova` function of the `rms` R package to report both the Wald χ^2 value and its corresponding p-value.

(MA3) Examining relationship. Finally, we examine the direction of the relationship between each explanatory variable and the review time. To do so, we use the `Predict` function of the `rms` package to plot the estimated review time while varying the value of a particular explanatory variable and hold the other explanatory variables at their median values.

RQ3 Analysis

To answer RQ3: What are the common intentions of links shared in the review discussion?, we conduct a qualitative analysis to investigate the intention of link sharing. In particular, we perform manual coding on a representative sample. Note that we use the same representative sample used in RQ1 (see 3). Below, we describe the coding scheme and manual coding process.

Coding scheme of intentions for link sharing: We hypothesis that links are shared to fulfill different information needs. Hence, we use the taxonomy of information needs of Pascarella et al. (2018) as the initial coding scheme. I rely on their taxonomy because their taxonomy is closely relevant to the study, i.e., what kinds of information that was requested by reviewers during code review in the MCR context, and the taxonomy has been validated based on a semi-structured interview.

To test how well the taxonomy of information needs of Pascarella et al. (2018) can be used to classify the intentions of link sharing, we randomly select 50 samples from the representative datasets and classify them into the taxonomy of information needs. More specifically, we identify the category of information needs which the share link aims to fulfill. This classification is conducted by the two authors of this paper. After the classification, the first fthe authors discuss whether the taxonomy of information needs can be used. We find that the links can be classified into the taxonomy of the information needs of Pascarella et al. However, we refine the taxonomy to focus on the intentions of link sharing since the taxonomy of Pascarella et al. focuses on the reviewers' questions. Table 4.3 shows the refined taxonomy of intentions of the work, which is derived from the taxonomy of information needs of Pascarella et al.

Manual coding process: After we refine the taxonomy of intentions for link sharing, we validate the coding schema by classifying another 30 links of the representative samples based on the taxonomy. This coding was conducted by the three authors of this paper. Then, we measure the inter-rater agreement using Cohen's Kappa. The average Cohen's Kappa score is 0.72 which indicates "substantial agreement" [156]. The somewhat lower agreement can be explained

Table 4.3: The taxonomy of intentions for sharing links.

Category	Description	Taxonomy of information needs [113]
Providing Context	The link is shared to provide the additional information related to the implementation.	Context —Reviewers ask about the information aimed at clarifying the context of a given implementation
Elaborating	The link is shared to complete the information or references related to the patch.	Rationale —Reviewers ask questions to get a rationale why the patch was implemented in a certain way.
Clarifying	The link is shared to clarify some doubts about the review process or to correct the reviewer’s understanding of the patch.	Correct Understanding —Reviewers ask questions to confirm the reviewer’s interpretation/understanding or to clarify doubts.
Explaining Necessity	The link is shared to inform more suitable solutions or explain the reasons why the patch is no longer needed.	Necessity —Reviewers need to know whether the patch (or a part of it) is necessary.
Proposing Improvement	The link is shared to point out an alternative solution or suggestion improvement.	Suitability of An Alternative Solution —Reviewers pose a question to discuss options and alternative solutions to the implementation of the patch.
Suggesting Experts	The link is shared to point out to an expert (other developers) who should be involved.	Specialized Expertise —Reviewers ask other reviewers to contribute with their specialized expertise.
Informing Splitted Patches	The link is shared to inform that the patch has been splitted.	Splittable —Reviewers ask questions to seek the possibility of splitting the patch into multiple, separated patches.

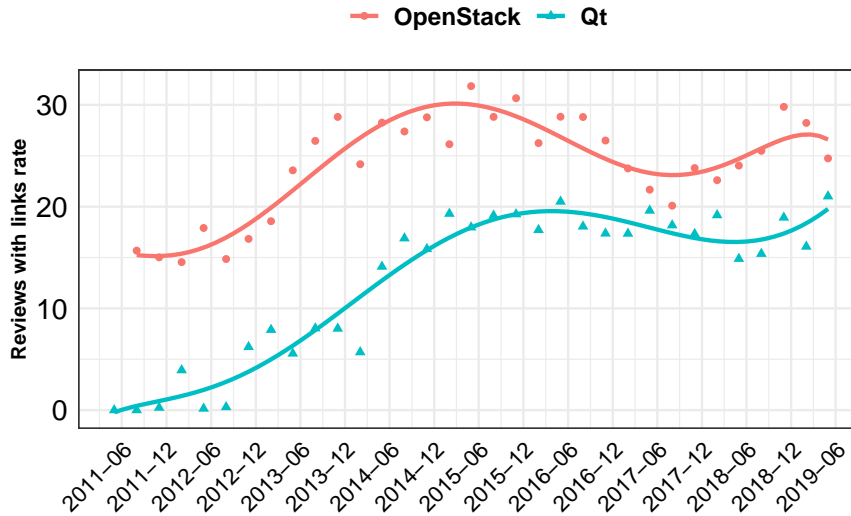


Figure 4.4: The proportion of reviews that have links in an interval of three months. In 2015-2019, 25% and 20% of the reviews have at least one link shared in a review discussion within the OpenStack and Qt.

by the need to extrapolate the intention behind a link from its context in the review discussion alone, without being able to interview the developer who added the link. After the validation, we splitted the remaining links into two sets. Then, the first author independently coded the first set and the second author independently coded the second set. In total, we manually classify 1,378 links. Note that when we classify the links, we also consider the textual content of the comments that contain links and the entire discussion thread to have a better understanding of the context.

4 Case Study Results

In this section, we present the results for each of the research questions.

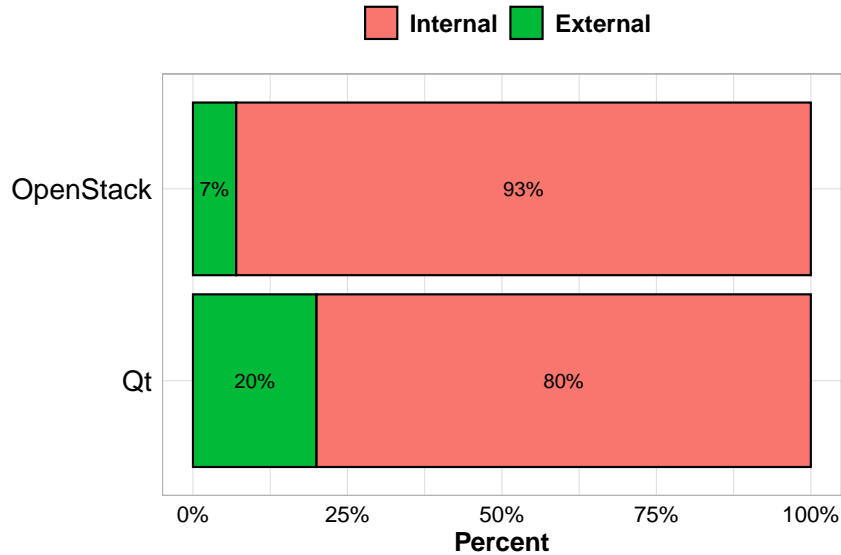


Figure 4.5: The proportion of internal and external links. 93% and 80% of links that are shared in code reviews are internal links within the OpenStack and Qt.

RQ1: To what extent do developers share links in the review discussion?

To answer RQ1, we analyze (1) the trend of link sharing (i.e., how often reviews have shared links overtime), (2) the common domains of the shared links, and (3) the types of link targets. Figure 4, 5, and Table 4 show the results of the analysis which is described in Section 3. We now discuss the results below.

Link Sharing Trend. In 2015–2019, 25% and 20% of the reviews have at least one link shared in a review discussion within the OpenStack and Qt, respectively. Figure 4.4 presents the proportion of reviews that have at least one link in the review discussion over time. We find that the proportion of reviews has an increasing trend from 2011 until 2014 for both OpenStack and Qt. Then the proportion of reviews remains at 20%–30% (OpenStack) and 15%–20% (Qt) from 2014 and onwards. This result suggests that links are commonly shared in a review discussion for code review.

Table 4.4: The five most common domains in OpenStack and Qt.

		Internal		External	
		OpenStack	Qt	OpenStack	Qt
Top 1	review.openstack.org (51%)	codereview.qt-project.org (79%)	github.com (15%)	paste.kde.org (14%)	
Top 2	github.com/openstack (13%)	bugreports.qt.io (6%)	docs.python.org (4%)	github.com (6%)	
Top 3	bugs.launchpad.net (7%)	testresults.qt.io (4%)	gist.github.com (4%)	msdn.microsoft.com (5%)	
Top 4	logs.openstack.org (6%)	doc.qt.io (2%)	bugzilla.redhat.com (3%)	pastebin.kde.org (3%)	
Top 5	wiki.openstack.org (4%)	wiki.qt.io (2%)	stackoverflow.com (2%)	gcc.gnu.org (3%)	

Common Domain in the Project. 93% and 80% of links that are shared in code reviews are internal links. Table 4.4 shows the ratio between internal and external links that are shared in OpenStack and Qt. We find the majority of links that are shared in the code reviews are internal links (i.e., links that are directly related to the projects). More specifically, Table 4.4 shows that 93% of links shared in OpenStack reviews are internal links, while only 7% of the links are external links (i.e., not directly related to OpenStack). Qt also has a similar ratio, where 80% of the links shared in Qt reviews are internal, and 20% of the links are external. These results indicate that links that are often shared in the review discussion are directly related to the project. In addition, Table 4.4 shows that the most common domains for the internal links are review.openstack.org and codereview.qt-project.org, which account for 51% and 79% of the internal links shared in OpenStack and Qt, respectively. The other common domains of the internal links shared in OpenStack reviews are github.com/openstack (OpenStack mirror projects in Github), bugs.launchpad.net, log.openstack.org, and wiki.openstack.org, which account for 30% of the internal links. For Qt, the other four common domains of the internal links are bugreports.qt.io, testresult.qt.io, doc.qt.io, and wiki.qt.io, which account for 14% of the internal links. On the other hand, the most common domain of external links is github.com for OpenStack and paste.kde.org for Qt.

Table 4.5: Frequency of link target types in our representative samples. The bold target categories are complemented from the work by Hata et al. [66].

	Internal		External	
	OpenStack	Qt	OpenStack	Qt
Licence	-	-	0.3%	-
Software homepage	1.1%	0.6%	9.2%	3.6%
Specification	2.6%	-	2.8%	0.6%
Organization homepage	-	-	0.6%	0.3%
Tutorial or article	7.1%	5.8%	18.7%	14.6%
API documentation	-	2.5%	15.3%	16.5%
Blog post	-	-	2.8%	1.9%
Bug report	9.2%	9.9%	8.3%	8.4%
Research paper	-	-	0.3%	-
Code	13.5%	4.1%	13.5%	10.4%
Forum thread	-	0.8%	0.3%	0.6%
Book content	-	-	0.6%	1.0%
Q&A thread	-	-	1.2%	0.6%
Stack Overflow	-	-	3.1%	1.9%
Communication channel	2.6%	0.3%	4.9%	3.6%
Github activity	0.3%	-	4.9%	3.6%
Media	-	-	2.1%	5.5%
Memo	5.8%	1.1%	5.8%	6.5%
Review	55.9%	73.6%	-	0.3%
Others	1.8%	1.4%	5.5%	20.1%

Link Target Types. We now further examine what kinds of information to which the shared links are referenced. Based on a manual coding on a statistical representative sample, Table 4.5 shows that reviews (a set of code changes) are the most frequently referenced by internal links, which account for 55.9% and 73.6% of the internal links for OpenStack and Qt, respectively. The other kinds of information that are frequently referenced by the internal links are bug report (9.2% for OpenStack; 9.9% for Qt), source code (13.5% for OpenStack; 4.1% for Qt), and tutorial or article (7.1% for OpenStack; 5.8% for Qt). On the other hands, we find that tutorial or article and API documentation are the two most frequent targets referenced by external links, which account for 18.7% and 15.3%, and 14.6% and 16.5% the external links shared in OpenStack and Qt, respectively. The other kinds of information that are frequently referenced by the external links are source code (13.5% for OpenStack; 10.4% for Qt), and bug report (8.3% for OpenStack; 8.4% for Qt). This might suggest that the external links often reference to temporary information. Specifically, within the Qt, we observe that 20.1% of external links are classified as Others. Through the manual analysis, these links are mostly referred to the links that can be accessible but requiring authentication. For example, when we click on the external link <https://paste.kde.org/pgjaet12z>, the web page shows that we need to sign in or sign up before continuing.

RQ1 Summary: In the past five years, 25% and 20% of the reviews have at least one link shared in a review discussion within the OpenStack and Qt. 93% and 80% of shared links are the internal links that are directly related to the project. Importantly, although the majority of the internal links are referencing to reviews, we find that the links referencing to bug reports and source code are also shared in review discussions. In addition, we find that the common target types of external links are tutorial and API documentation.

RQ2: Does the number of links shared in the review discussion correlate with review time?

To answer RQ2, we analyze the correlation between link sharing variables and the review time, using a non-linear regression model. Figure 6 and Table 6 show the results of our model construction and model analysis which is described in Section 3. We now discuss our results below.

Model Construction. We first describe our model construction. Table 4.6 shows the surviving explanatory variables that are used in our models. Based on the hierarchical clustering analysis, we remove those explanatory variables that are highly correlated with one another, i.e., Patch size, # Comments, # Reviewers, and # Total links. For the surviving explanatory variables, we do not find a redundant variable, i.e., the variable that has a fit with an R^2 greater than 0.9 during the redundancy analysis. The budgeted degrees of freedom are then carefully allocated to the surviving explanatory variables based on their potential for sharing a nonlinear relationship with the response variable as described in Section 3.4. We spent 24 degrees of freedom on OpenStack and Qt models.

Model Analysis. We now analyze the goodness of fit of our models. Table 4.6 shows that our non-linear regression model achieve an adjusted R^2 of 0.3737 (OpenStack) and 0.4580 (Qt). The adjusted R^2 scores are acceptable as our models are supposed to be explanatory not for the predictive purpose [104]. Taking overestimation into account, after applying the bootstrap techniques with 1,000 iterations, we find that the optimism of an adjusted R^2 is 0.0004 and 0.0007 for OpenStack and Qt models, respectively. The result indicates that our constructed models are stable and can provide a meaningful and robust amount of explanatory power.

We now discuss the explanatory power of the variables of interests (i.e., # External links, # Internal links) and their relationship with the review time. Table 4.6 shows the explanatory power (Wald χ^2 value) of our explanatory variables that contribute to the fit of our models. In the table, the ‘Overall’ column shows the Wald χ^2 value of the entire model fit that the explanatory variable contributes to the fit of the model, while the ‘Nonlinear’ column shows the Wald χ^2 value

Table 4.6: Review time model statistics.

Adjusted R^2		OpenStack		Qt	
		0.3737		0.4580	
Optimism-reduced adjusted R^2		0.3733		0.4573	
Overall Wald χ^2		34,649		34,358	
Budgeted Degrees of Freedom		3,873		2,711	
Spent Degrees of Freedom		24		24	
Confounding variables		Overall	Nonlinear	Overall	Nonlinear
Patch size	D.F. χ^2	†		†	
Add	D.F. χ^2	2 154***	1 153***	2 337***	1 337***
Delete	D.F. χ^2	1 1 ^o	-	1 0.53 ^o	-
Purpose	D.F. χ^2	2 276***	-	2 131***	-
# Files	D.F. χ^2	2 31***	-	1 0.34 ^o	-
Patch author Exp.	D.F. χ^2	1 220***	-	1 98***	-
# Comments	D.F. χ^2	†		†	
# Author comments	D.F. χ^2	3 1936***	2 1679***	3 2216***	2 1549***
# Reviewer comments	D.F. χ^2	4 1360***	3 1066***	3 4908***	2 4036***
# Reviewers	D.F. χ^2	†		†	
# Revisions	D.F. χ^2	3 3237***	2 1847***	3 2038***	2 1687***
Link sharing variables		Overall	Nonlinear	Overall	Nonlinear
# External links	D.F. χ^2	1 3 ^o	-	1 0.22 ^o	-
# Internal links	D.F. χ^2	1 119***	-	1 78***	-
# Total links	D.F. χ^2	†		†	

†: This explanatory variable is discarded during variable clustering analysis $|\rho| \geq 0.7$

-: Nonlinear degrees of freedom not allocated

Statistical significance of explanatory power according to Wald χ^2 likelihood ratio test:

o $p \geq 0.05$; * $p < 0.05$; ** $p < 0.01$; *** $p < 0.001$

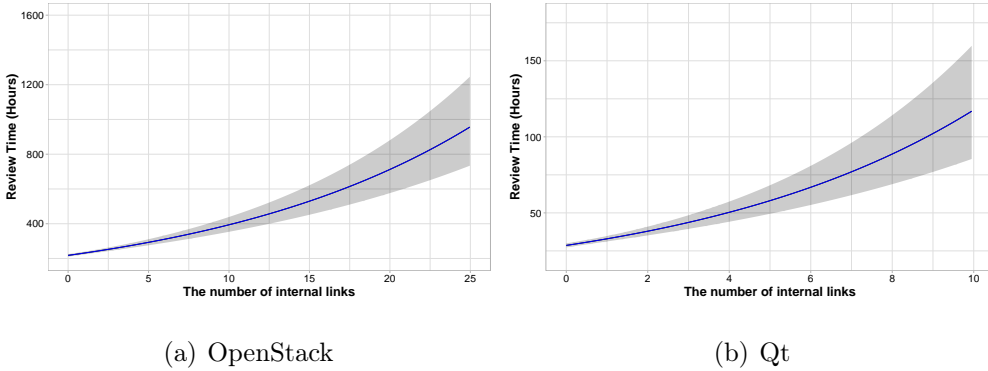


Figure 4.6: The direction of the relationships between the number of internal links and the code review time. The light grey area shows the 95% confidence interval. It shows that the more internal links are shared during the discussion, the longer review time will be taken.

that the nonlinear component of the explanatory variable contributes to the fit of the model. Taking a look into link sharing variables, the statistics show that there is no significant correlation between the number of external links and the review time ($p\text{-value} > 0.05$) for both studied projects. On the other hand, we observe that the number of internal links has a significant correlation with the review time ($p\text{-value} < 0.001$). However, the explanatory power of the number of internal links is not as large as the explanatory powers of the confounding variables. More specifically, the Wald χ^2 values of the number of internal links account for only 0.4% ($\frac{119}{34,649}$; OpenStack) and 0.3% ($\frac{78}{34,358}$; Qt), while the Wald χ^2 values of the variables range from 0.4%–10% (OpenStack) and 0.3%–13% (Qt). These results suggest that the internal link has a relatively weak correlation with the review time when compared to other variables.

Figure 4.6 shows the relationship between the number of internal links and the review time. We find that for both studied projects, the number of internal links has an increasing relationship with the review time. In other words, the more internal links shared in the review discussions, it tends to take a longer time for the patch to be reviewed. There are two possible conjectures that how internal links may contribute in a longer review process. One potential reason is that developers may spend time on finding the related internal links. Another

possible reason is that since the internal links are closely related to the project resources, it will take time for developers to pay attention to and understand the project environment. Although the model shows an increasing relationship, we can not explain the causality between the internal link count and the review time, since before the internal links occur, the long review time could have already been taken.

RQ2 Summary: Our non-linear regression models show that the internal link has a significant correlation (but relatively weak) with the review time. However, the external link is not significantly correlated with the review time. Furthermore, we observe that the number of internal links has an increasing relationship with the review time.

RQ3: What are the common intentions of links shared in the review discussion?

To answer RQ3, we analyze (1) the kinds of common intentions of sharing links, and (2) the frequency of these intentions within our studied projects. Below, we first provide representative examples for each intention type, and then we discuss the result of the frequency of intentions (Figure 7).

Taxonomy of Common Intentions. Seven intentions of sharing links are classified through our qualitative analysis, which is described in Section 3:

(I) Providing Context. This category emerges by grouping discussions in which the links are shared to provide additional information related to the implementation. The Ex 1⁸ shows that the reviewer shared an internal review link for the author to inform the review team of the dependent patch. During the classification process, we observe that apart from sharing review links, the developers also share specific log results or screenshots for review teams to better understand the code change implementation. For instance, in the Ex 2⁹, the reviewer self came across a test failure. To reduce the confusion, the reviewer attached an external memo link recorded with the log result for the review team to check. Note that

⁸<https://codereview.qt-project.org/c/qt/qtbase/+/-194731>

⁹<https://review.opendev.org/#/c/25515/>

although the log result is from the CI tool of the studied project, the link that is shared is not directly related to the project.

Ex 1

Reviewer: Depends on [\[internal review link\]](#).

Ex 2

Reviewer: Hmm, test failures from Jenkins look real, if confusing: [\[external memo link\]](#).

(II) Elaborating. The category refers to the links that are shared to complete the information provided in the review comment or references related to the patch. In this case, the keywords are usually left on the comments such as “refer to”, “for example”, “for your information”. We show the following two representative examples to describe the intention of elaborating regarding internal and external links. In the Ex 3¹⁰, the reviewer presented his review suggestion, and to complete the opinion, the reviewer as well shared an internal code link as a reference for the author. For the external link, as shown in the Ex 4¹¹, the author shared a Q&A thread link to explain what is the needed HZ value.

Ex 3

Reviewer: I would prefer that you didn't merge this. Like mentioned in previous review, if 'import' is removed and there is no code/comment/docstring, the license header should be removed as well. Please refer to: [\[internal doc link\]](#)

Ex 4

Author: For example on page [\[external Q&A thread link\]](#) it is well explained for what is this HZ value needed there.

(III) Clarifying. In this category, the links are shared to clarify some doubts of review process or to correct the reviewer's understanding of the patch. We find the clarification can be claimed from either the reviewer or the patch author aspect. The Ex 5¹² illustrates the case where the patch author used an internal code link to address the reviewer's doubts about the undefined behaviour with the

¹⁰<https://review.opendev.org/#/c/75839/>

¹¹<https://review.opendev.org/#/c/236210/>

¹²<https://codereview.qt-project.org/c/qt/qtbase/+/160883>

code change. In the Ex 6¹³, the patch author thought that signed types implicitly converting to unsigned ones was not a problem. However, the reviewer shared external doc links to explain that it is a true problem that should be focused on.

Ex 5

Reviewer: What is this fixing? Is there an undefined behaviour i am not seeing?

Author: I probably need to add this to the commit message, but: 1. by fixing it to QObject, we fix it to a specified size, independent of which class type the member belonged to (see long explanation in [\[internal review link\]](#).) 2. since it no longer depends on the class type, the impl() functions are generated based only on the signal and slot's arguments, not on the class they are from. This reduces template bloat. [\[internal code link\]](#).

Ex 6

Author: Signed types implicitly convert to unsigned ones. That's not a problem.

Reviewer: I think that **is** a problem: going from signed to unsigned is not an integral promotion [\[external doc link\]](#), but an integral conversion [\[external doc link\]](#)

(IV) Explaining Necessity. This category refers to the links that are shared to inform more suitable solutions or explain the reasons why the patch is no longer needed. We observe this category can happen in both internal and external links. For example, as shown in the Ex 7¹⁴, the reviewer shared a review link and pointed out that the linked review had already changed `network_type` validation using a simple approach. And further suggested that the current patch was not really needed. The Ex 8¹⁵ is related to the shared external link with the intention of explaining necessity. One developer considered the submitted change was not sufficient since `libproxy` already had a fix. At the end of the comment, the developer shared an external GitHub link to indeed prove that the fix has been done through this link. Note that the external GitHub link is not directly

¹³<https://codereview.qt-project.org/c/qt/qtbase/+/186305>

¹⁴<https://review.opendev.org/#/c/22928/>

¹⁵<https://codereview.qt-project.org/c/qt/qtbase/+/200463>

related to the GitHub of the studied project.

Ex 7

Reviewer: I would prefer that you didn't merge this
The WIP ml2 patch at [\[internal review link\]](#) changes the network_type validation in the core to simply validate a string. Is anything more really needed?

Ex 8

Author: This change is insufficient. libproxy already has a mutex, so adding another one won't solve anything.
* [\[external GitHub link\]](#)

(V) Proposing Improvement. In this category, we group discussions in which the links are shared to point out an alternative solution or suggestion improvement. As shown in the Ex 9¹⁶, the reviewer provided the author with an internal GitHub code link and suggested the author should follow the proposed method. Similarly, the reviewer asked the author to do something like what the shared long-term memory link did in the Ex 10.¹⁷

Ex 9

Reviewer: The compute api loading code and the new hostapi loading code (and network api loading code for that matter) should follow the method that the volume api loading code uses here: [\[internal GitHub code link\]](#)

Ex 10

Reviewer: could we do something like that: [\[external long-term memory link\]](#)
... a bit rough but you should get the idea ... get all (without specifying any attr) layer

(VI) Suggesting Experts. We define this category as the links are shared to point out to an expert (other developers) who should be involved. In Ex 11¹⁸, the author shared an internal review link related to the spec change and invited the reviewer named John to have a review as well. In the Ex 12¹⁹, in order to

¹⁶<https://review.opendev.org/#/c/12311>

¹⁷<https://codereview.qt-project.org/c/qt/qtbase/+/126975>

¹⁸<https://review.opendev.org/#/c/219248/>

¹⁹<https://review.opendev.org/#/c/219248/>

address the reviewer’s question, the author used @ to suggest an expert along with an issue link to point out he was an experienced maintainer regarding such a situation.

Ex 11
Author: Hi John, spec is on review: [\[internal review link\]](#). Could you please review it?

Ex 12
Reviewer: Or is it a bug in the protocol?
Author: @psychon (a.k.a Uli Schlachter) who is a maintainer of Awesome WM has a good comment regarding the situation with the standard [\[external GitHub issue link\]](#).

(VII) Informing Splitted Patches. In this category, the links are shared to inform that the patch has been splitted. We only find this intention existing behind the internal link sharing. As shown in the Ex 13²⁰, the reviewer was trying to abandon the current patch and split it into several small chunks to fulfill the request, along with internal review links.

Ex 13
Reviewer: I’m abandoning this patch as a result of the request to split this review into smaller chunks - these reviews are [\[internal review link\]](#).

Frequency of intentions for link sharing. We now examine what are the common intentions for sharing links in the review discussion. Figure 4.7 shows the distribution of each intention category within internal and external types for OpenStack and Qt. The figure clearly reveals that not all the intentions are equally distributed and highlights the presence of a particular type. Specifically, for internal links (i.e., directly related to the project), we observe that *Providing Context* category is the most frequent intention in OpenStack and Qt (128 and 134 links, respectively). The result indicates that internal links are commonly shared in the review discussion to provide additional information related to the implementation of a patch. For the external links (i.e., not directly related to the project), we find that the most common intention is *Elaborating* in both

²⁰<https://review.opendev.org/#/c/103167>

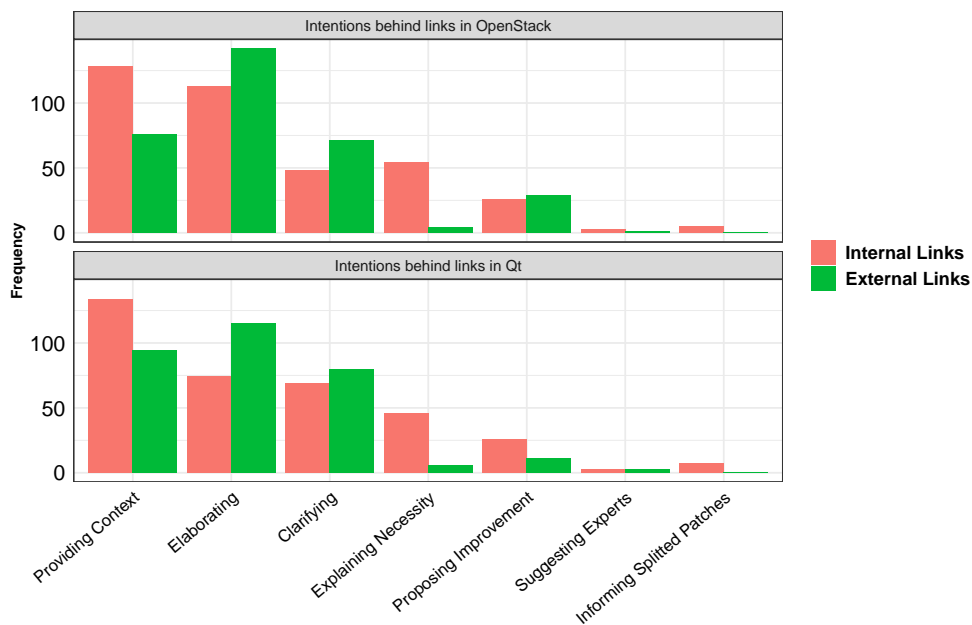


Figure 4.7: Distribution of seven intentions behind sharing links across the studied projects. The results show that *Providing Context* and *Elaborating* are the most common intentions for internal and external links, respectively.

projects (145 and 115 links, respectively). This finding suggests that external links are commonly shared to complement the information provided in the review comment.

Table 4.7: The three most frequent intentions of sharing review links.

Intention	OpenStack	Qt
Providing Context	40%	40%
Explaining Necessity	23%	18%
Elaborating	19%	16%

Upon closer inspection on the review links (i.e., the most common shared link target in RQ1), as shown in Table 4.7, we find that the most frequent intention is to provide context, accounting for 40% for both OpenStack and Qt project. Through our manual coding, we observe that such context is usually concerning

the patch dependency²¹ and integration test environment.²² The second most frequent intention of sharing review links is to explain necessity, accounting for 23% and 18% for the OpenStack and Qt project, respectively. The third most frequent intention is to elaborate, i.e., 19% and 16% for the OpenStack and Qt project, respectively.

RQ3 Summary: We identify seven intentions of sharing links: (1) Providing Context, (2) Elaborating, (3) Clarifying, (4) Explaining Necessity, (5) Proposing Improvement, (6) Suggesting Experts, and (7) Informing Split-tered Request. We find that providing context is the most common intention for sharing internal links and elaborating (i.e., providing a reference or related information) to complement review comments is the most common intention for sharing external links.

5 Discussions

In this section, we discuss the implications of our analysis results. To gain a further insight of the perception of sharing links by developers, we conduct a survey study with OpenStack and Qt developers. Below, we first present the results of our survey. Then we provide suggestions to the patch author, review teams, and researchers.

5.1 Developer Feedback

To gain insights into developer perception of link sharing, we sent out a survey²³ to OpenStack and Qt developers, with the goal to (i) receive feedback on the three study findings, (ii) solicit developers' opinions, and (iii) collect insights into the developer experience with existing functionalities (i.e., related changes, same topic). The survey consists of two likert scale questions and five open-ended questions. We sent our online survey invitation to 1,871 developers who have shared links in the past based on our studied dataset. The survey was open from

²¹<https://review.opendev.org/#/c/612393/>

²²<https://review.opendev.org/#/c/453537/>

²³<https://forms.gle/hiBameBdGFMhnxNSA>

January 22 to February 13, 2021. We received responses from 53 developers in total. To analyze the responses of the open-ended questions, we use the card sorting method. Below, we present our survey questions and discuss the survey results.

Feedback on Findings of RQ1 and RQ2. Table 4.8 shows the feedback on our findings of RQ1 and RQ2. For the finding of RQ1, forty-eight respondents (90% = 48/53) agreed that developers often share internal links to reference reviews, bug reports, and source code, while external links often reference tutorials and API documentation. In the open-ended questions, six respondents reported that the external link target is also to point out to bug reports outside the projects. This response is consistent with our analysis results of RQ1 as well, i.e., bug report is the fourth frequent external link target (see Table 5). Interestingly, one respondent stated that *“This (links) is useful for example for newcomers that may have missed guidelines or did not found the corresponding bug report when doing submitting a fix.”*









We found that eleven respondents partially agreed with the finding of our RQ2, while fifteen respondents did not have opinion and twenty-seven respondents disagreed with our findings. One of the respondents who agree cited that *“Sometime we are not familiar wit the exact context of internal item.”* Eleven respondents who did not agree reported that the information brought with the shared links is useful and could aid the review process. For instance, one respondent cited that *“Links usually provide a concise and clear answer compared to trying to explain it in prose.”* Such perception of developers is consistent with our intuition as stated under the RQ2 Motivation in the Introduction. However, the analysis result shows an inverse relationship. There could be other confounding factors that play a role, and future work should further investigate the causality of this relationship.

Survey on Intentions of Sharing Links. In the survey questions, we also asked the respondents to select the intention(s) that they usually use when sharing link. Table 4.9 shows the most selected intention of sharing link is to provide context (48/53 = 90% of respondents). The second most frequent intention from developers was to explain necessity using links (forty respondents voting),

Table 4.8: Feedback on findings of RQ1 and RQ2, using the Likert-scale scale below: 1 = Strongly disagree, 2 = Partially disagree, 3 = No opinion, 4 = Partially agree, 5 = Strongly agree.

	1	2	3	4	5
Finding of RQ1-“Developers often share internal links to reference reviews, bug reports and source code, while external links often reference tutorials and API documentation.”	0	3	2	31	17
Finding of RQ2-“A review that has an internal link shared during its review discussion is likely to take reviewing time longer than other reviews.”	3	24	15	11	0

Table 4.9: Respondents feedback on the intentions for sharing links.

Intention	Respondent Count	
Providing Context	48	
Explaining Necessity	40	
Elaborating	35	
Clarifying	33	
Proposing Improvement	26	
Informing Splitted Patches	17	
Suggesting Experts	6	
Others	3	

followed by the intention to elaborate (thirty-five respondents voting). These frequent intentions are overall consistent with our findings of our RQ3, especially for internal links (see Fig. 7). The least frequent intentions from respondents are to inform splitted patches (17 respondents) and suggest experts (6 respondents). In regards to the open-ended feedback, one respondent stated that “*Usually when doing a code review, if an expert should take a look at the review, he will likely be added by one of the reviewer if not already by the submitter.*”.

Survey on Perception of Existing Functionalities of Sharing Related Patch . Finally, we asked the respondents about their experience and perception of the Gerrit functionalities, which provide review links of related changes and the same topic. Out of thirty-five responses, twenty-four respondents claimed experience in using the functionalities. On the one hand, fourteen respondents acknowledged the usefulness, especially to find related patches or track dependencies. On the other hand, respondents did express limitations, e.g., “ *It seems useful for seeing what changes are submitted at a similar point in the change history, but doesn’t seem useful for finding patches that are related by content (e.g., changing the same feature) but separated by longer periods of time.*”.

When we asked about the differences between existing functionalities and the practice of sharing links in the review discussion, twelve respondents acknowledged how both approaches can complement each other. For example, one of the respondents commented: “*The tool can help you find information you were already looking for, but the posting of a link is a communication option to help convey the idea to other people.*”.

5.2 Suggestions

Based on the our results of RQs 1-3 and the survey results, we now present our suggestions of the study.

Suggestions for Patch Authors. Our RQ1 shows that the majority of shared links are directly related to the project. Figure 5 shows that 93% and 80% of shared links are internal links for OpenStack and Qt, respectively. The link targets are diverse from different locations with the complexity of projects. As shown in Table 5 from RQ1, we found eleven different target types. We suggest that in the case of well-documented projects (i.e., OpenStack and Qt), patch authors (especially for newcomers or novice developers) should read the project related guidelines to be familiar with the environment before their submission. For instance, in the review #37843²⁴, one reviewer shared a link related to the Gerrit Workflow (i.e., https://wiki.openstack.org/wiki/Gerrit_Workflow) with the patch author who was newly to the project, to avoid the broken conflict. This is also supported by the responses of developers: “*This (links) is useful*

²⁴<https://review.opendev.org/#/c/37843>

for example for newcomers that may have missed guidelines or did not found the corresponding bug report when doing submitting a fix.”

Through our qualitative analysis in RQ3, the observation suggests that the information brought by the shared links is helpful as an indicator for review teams to clearly understand the patch implementation context. For one example of clarifying intention (Ex 5) shown in Section 4.3, during the review process, the reviewer got confused about the behaviour of the fix. To reduce the confusion, the author decided to improve the patch content in the commit message, along with shared links to supplement further explanation. Another example of elaborating intention (Ex 4) illustrates that to help the review team better understand what is HZ value, the patch author shared a Q&A thread link as a reference. One surveyed respondent also suggested that *“It could also decrease the review time by making more clear the intent of the code change.”*. Inspired by these examples, we encourage patch authors to provide more information such as implementation related information via shared links during their submission, in turn to receive efficient feedback from review teams quicker and reduce the discussion confusion.

Suggestions for Review Teams. Our study indicates that the practice of sharing links can fulfill various information needs. Seven intentions are classified as shown in Table 3. Our frequency analysis in RQ3 shows that the links are commonly shared to provide the context, further elaborate, and clarify doubts. Moreover, the information that is shared through links is also diverse. The example of providing context intention (Ex 2) in Section 4.3 shows that one reviewer shared an external memo link which was recorded with CI tool test failure results, in order to solve the confusion among the review team and the patch author. On the other hand, as shown in the example of explaining necessity intention (Ex 7), the reviewer noticed that a related patch already changed the `network_type` validation and shared this review link in the review discussions to suggest that the current patch was no longer needed. These findings show that sharing links can help developers fulfill the information needs (the challenges discovered by Pascarella et al.), which potentially saves the reviewer effort. Thus, we suggest that during the future review process, the review teams should share links to transfer the needed information for guiding the patch author and the review process, especially for the review team that does not adopt such practice. One respondent

from our survey commented that *“Links usually provide a concise and clear answer compared to trying to explain it in prose.”* Our suggestion also extends the suggestion of Pascarrella et al. that in addition to automatic change summarization, sharing links could be another method to meet the information needs. We believe that such information may help to conduct a more efficient review and also assist with mentoring new members to the review team.

Suggestions for Researchers. Our RQ1 results indicate that link sharing is becoming a popular practice during review discussions in the MCR. Furthermore, Figure 4 shows that in the last five years, around 25% and 20% of the reviews have at least one link within the OpenStack and Qt project. As the practice of linking sharing increases, new opportunities arise for researchers to develop tool support, especially to recommend related and useful links for both the patch author and review teams in order to facilitate the review process. An intelligent tool may reduce the time for developers to find the needed links. We propose the following three potential features that could be embedded in the future code review tool: First, the mechanism to automatically recommend related patches can be improved not only based on the similar change history, but also considering the patch contents. Such limitation is also pointed by the responses *“It seems useful for seeing what changes are submitted at a similar point in the change history, but doesn’t seem useful for finding patches that are related by content (e.g., changing the same feature) but separated by longer periods of time.”*. Second, a functionality to detect alternative solution patches (i.e., patches aim to achieve the same objective) is needed, since our empirical study shows that the second most frequent intention of sharing review links is to explain necessity (See Table 7). Third, a tool to recommend guideline and tutorial related link would be especially useful for novice developers and help them to be familiar with the project environment. This study also lays the groundwork for future research on the links shared in the review process to generate the structural and dynamic properties of the emergent knowledge network, aiming to enable more effective knowledge sharing within the project.

6 Threats to Validity

We now discuss threats to the validity of our empirical study.

External Validity. We perform an empirical study on two projects relying on Gerrit review tools. Although OpenStack and Qt commonly used in the prior research, the observations based on this case study may not generalize to other projects or peer review settings such as the pull-based review process. However, our goal is not to build a theory that can be fit to all projects, but rather to shed light in some large open-source projects, the links being often shared in the code reviews to provide the context, elaborate to complement review comments. We only focus on the large open-source projects with distributed teams, since most of the code review activities are performed through the code review tool (the data is available). The data or communication recorded in the small or medium team may be incomplete as they can have in-person communication or using other channels to discuss, like slack. Nonetheless, additional replication studies would help to generalize our observations. Thus, in order to encourage future replication studies, our replication package is available online including the raw review datasets, manually labeled link targets and their intentions, and the script to construct the non-linear regression model.

Construct Validity. We summarize two threats regarding construct validity. First, in the identification of external and internal links, we apply the keyword search to automatically split domains into external and internal types. However, cases might occur where some domains not including any indicated keywords can still belong to internal links. To reduce such bias, we manually click the domains to validate the correctness carefully.

Second, in our qualitative analysis, especially for intention classification, intentions may be miscoded due to the subjective nature of our coding approach. To mitigate this threat, we took a systematic approach to first test our comprehension with 30 samples using Kappa agreement scores by three separate individuals. Only until the Kappa score reaches more than 0.7 (i.e., 0.83 for link targets and 0.72 for link intentions), indicating that the agreement is substantial (0.61–0.80)

or almost perfect (0.81–1.00), we were able to complete the rest of the sample dataset.

Internal Validity. Four related threats are summarized. The first threat is concerning the link extraction. In this study, we only consider the links which are posted in the general comments. We understand that links can also be shared in the inline comments. However, the prior work [69] pointed out that the proportions of links in the inline comments are relatively low, accounting for 18% and 10% for OpenStack and Qt project, respectively. The analysis of links shared in inline comments may provide further insights, but may not have a large impact on our findings in this paper.

The second threat is related to the results derived from the statistical models that we fitted to our data. Though we can observe the correlation between explanatory and dependent variables, the causal effects of link sharing on the review time cannot be represented. Thus, future in-depth qualitative analysis or experimental studies are needed so as to better understand the reasons and effects of link sharing impact.

The third threat is regarding our factor selection to fit the statistical models. Other factors might also influence the review time. For instance, the prior study showed that the code ownership has an impact on the review process [149]. Yet, we take commonly used metrics into account similar to the work conducted by Kononenko et al. [83]. We are confident that these selected explanatory factors are appropriate to be considered and measured.

The last threat is concerning the model performance overestimation. An overfit model may exaggerate spurious relationships between explanatory and response variables. To mitigate this concern, we validate our model results using the bootstrap-calculated optimism with 1,000 iterations.

7 Summary

In this paper, we perform an empirical study on two open source projects, i.e., OpenStack and Qt, to (1) analyze to what extent do developers share links, (2) analyze the correlation between link sharing and the review time using the sta-

tistical model, and (3) investigate the common intentions of sharing links. Our results show that the majority of shared links are internal links (directly related to the project), i.e., 93% and 80% for OpenStack and Qt. We find that although the majority of the internal links are referencing to reviews, the links referencing to bug reports and source code are also shared in review discussions. Through the statistical models, our results show that the number of internal links has an increasing relationship with the review time. Regarding the intention classification, we identify seven intentions behind link sharing, with providing context and elaborating being the most common intentions for internal and external links.

Our study highlights the role that shared links play in the review discussion and the link is served as an important resource to fulfill various information needs for patch authors and review teams. The next logical step would be a deeper study of investigating the causality of these factors and understanding the reasons why it takes a longer time to complete the review. Future research directions also include the extension of a more exhaustive study that investigate the small and medium open-source projects, the in-depth analysis of link sharing practices (e.g., an impact of links shared by a patch author and reviewers on the review process), the potential for tool support, and the management of the collective knowledge within projects.

5 | An Exploration of Cross-Patch Collaborations via Patch Linkage

Empirical results in Chapter 4 show that the most common links shared by developers are review links (patch related links). As highlighted in the recent work, collaboration is crucial and becomes a challenge during code review process. This chapter hypothesizes that patch linkage may have association with collaborations. Specifically, this chapter investigates to what extent do developers collaborate across linked patches after the patch related links are shared.

1 Introduction

Software development teams nowadays benefit from online code review tools (e.g., Gerrit, Codestriker, and ReviewBoard) to effectively inspect patches and improve the code quality of their projects, while enabling the teams to perform synchronized code reviews that are more lightweight and flexible [132]. On the other hand, a large number of code reviews are being performed by software teams as new patches (i.e., a set of code changes) frequently occur in a contemporary code review setting [126]. For example, the 2018 OpenStack User Survey report¹ showed that about 70,000 patches were reviewed, with an average of 182 code reviews changes per day. Such a large number of code reviews potentially poses a

¹<https://www.openstack.org/user-survey/2018-user-survey-report/>

new challenge of collaboration (e.g., improving the patch, fixing defects) during code reviews and development tasks.

More specifically, recent studies highlight evidence of why developers should collaborate across code review tasks. Zhang et al. [178] found that redundant patches (i.e., patches that address the same task or problem) are often submitted for a review in software projects hosted in GitHub. Ebert et al. [46] observed that the inclusion of more people in the code review increases their awareness of the code change, i.e., confusion resolution contributes to knowledge sharing. Recently, Wang et al. [158] observed that developers are likely to share links during review discussions with seven intentions to fulfill information needs. Meanwhile, Hirao et al. [69] shed light that the patch linkage (i.e., posting a patch link to another patch) is used to indicate patch dependency, competing solutions, or provide broader context. As the recent work has shown that patch linkage can increase the awareness of the related patches, we further investigate to what extent do developers collaborate across these linked patches.

Figure 5.1 provides an illustrative scenario where collaboration occurs after the patch linkage. As shown in the figure, a reviewer *Pink* in Patch A posted a patch link to Patch B in the review discussion. In this patch linkage, we consider Patch A as a source patch and Patch B as a target patch. After the patch link is posted, a developer *Green* who participated in the Patch A discussion votes and leaves review comments in Patch B. At the same time, a developer *Blue* who participated in the Patch B discussion before the linking time could also provide comments in Patch A discussion. We consider either of these two cases as a collaboration occurrence.

In a realistic scenario (i.e., review at <https://review.openstack.org/#/c/211019>), we observed that a reviewer posted a comment with a collaboration request to the patch author:

‘Could you please sync your efforts with another patch [https://review.openstack.org/#/c/209612/]?’

After the patch link is posted, we observe that the author and one of the reviewers from Patch 211019 made the specific review comments related to the code changes in Patch 209612. We hypothesize that there exist collaborations (**cross-patch collaborations**) that occur after the patch linkage.

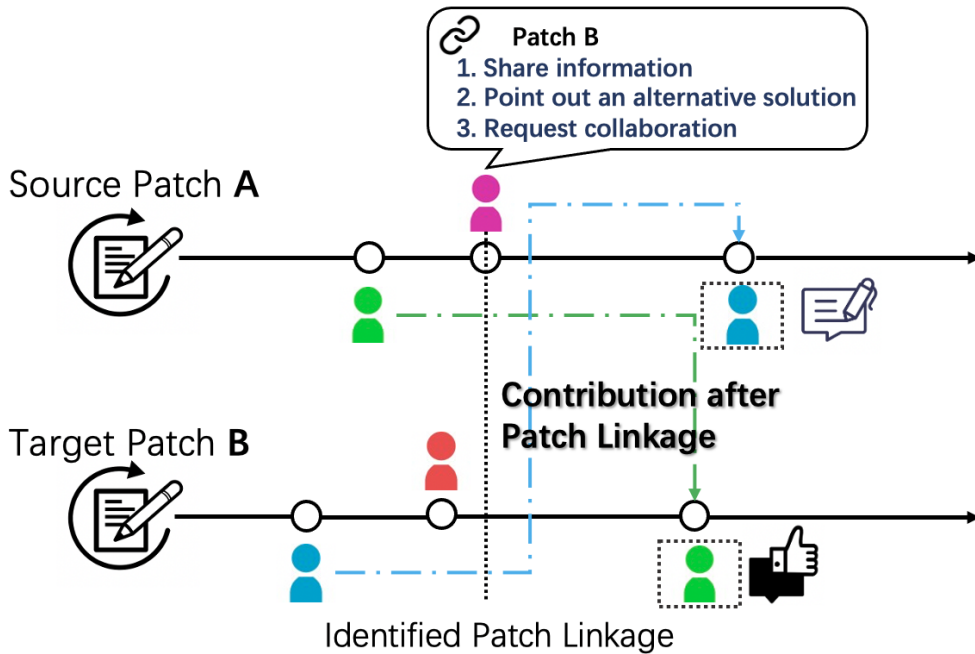


Figure 5.1: A conceptual illustration that describes (1) a linkage between two patches is identified and (2) a collaboration activity happens where a developer on one patch contributes to the review of the other patch.




In this work, we conduct an empirical exploration study of 368 patch linkages from a total of 8,612 linked patches to better understand the intentions of the patch linkage (e.g., requesting a collaboration) and the degree to which collaboration occurs after the patches are linked. Furthermore, we investigate how different types of linkage sharing lead to collaboration opportunities and characterize the contributions that follow after the link is identified.

2 Data Collection

For our experiments, we used the OpenStack review dataset provided by Thongtanunam and Hassan [146]. The dataset includes 58,212 patches dated from November 2011 to July 2019.

Since we focus on the collaboration and contributions done by patch authors or reviewers, we exclude the comments that are posted by automated tools in

Table 5.1: The prevalence of link types and their timing nature.

Link Type	Count	Patch-linked Time (# days)				Patch-closed Time (# days)			
		1st Qu.	Median	Mean	3rd Qu.	1st Qu.	Median	Mean	3rd Qu.
Requesting collaboration	57 	1.2	14.1	48.8	56.5	3.3	20.6	92.1	67.2
Sharing information	211 	0.9	11.6	38.3	48.2	1.0	10.8	55.1	49.2
Pointing out an alternative solution	100 	0.4	4.0	31.7	32.8	0.0	0.9	31.2	19.2

the discussion threads. To do so, we refer to the documentation of the studied system² to identify the automated tools that are integrated with the code review tools.

To identify the patch links, we applied the regular expression to search all messages in the review discussions that include a patch URL in the following format: `https?://review.openstack|opendev.org/##/c/[1-9]+[0-9]*`. A total of 8,944 pairs of patches are retrieved. Then we exclude the case where the source and target patches are the same. In our study, we keep the cases where (i) the patch linkages are written by the same patch authors and (ii) the patch authors post links by themselves, as we assume that collaboration could occur between the reviewers of both patches. Finally, we obtain 8,612 pair of patches that met our experiment criteria.

3 Preliminary Study

Although Hirao et al. [69] have shown that patch linkage is mainly for team awareness (i.e., indicating dependency, providing broader context, and pointing out an alternative solution), we hypothesize that the sharing of patch linkage has association with developer collaboration across the patches. The goal of our preliminary study is to first analyze the requests made for collaboration and then identify collaboration after the patch linkage.

3.1 Requesting Collaboration

We perform a manual analysis to investigate the intention behind the patch linkage. More specifically, our analysis mainly focuses on how often the patches are

²<https://docs.openstack.org/infra/manual/developers.html>

linked to request collaboration. Below, we describe our manual coding based on a statistically representative sample of our patch linkage dataset:

- *Representative dataset construction.* As the full set of our constructed data is too large to manually examine their collaboration intention, we then draw a statistically representative sample. The calculation of statistically significant sample sizes based on population size, confidence interval, and confidence level is well established [85], with a confidence level of 95% and a confidence interval of 5. We randomly sample 368 patch linkages.³
- *Manual coding.* In this step, we classify whether the patch linkage is for requesting collaboration or not. Based on the finding of prior work, patch linkage can be also for sharing information or pointing out an alternative solution. Hence, we classify the intention of patch linkages into three main types:
 - *Requesting collaboration:* Patch linkage for requesting collaboration is the linkage where a developer (either a patch author or reviewer) posts a link with a message that explicitly requests other developers to collaborate in the target patch. In this case, developers often write the message which includes words such as ‘help’, ‘collaborate’, ‘integrate’ or ‘rebase on’. For example, “*Patch Set 1: Code-Review-1 Can we please rebase this on <https://review.openstack.org/#/c/93842/> that review ensures specific values is present in the string for the flag to be switched on. thanks, dims*”.
 - *Sharing information:* Patch linkage for sharing information is the linkage where a developer posts a link to increase team awareness (e.g., indicating patch dependency, providing broader context)
 - *Pointing out an alternative solution:* Patch linkage for pointing out an alternative solution is the linkage where a developer posts a link to mention that the target patch attempts to explicitly address the same or similar objective as the source patch.

³<https://www.surveysystem.com/sscalc.htm>









To classify the patch linkages into a category, we consider the whole textual message that comes with the link. In some cases, we also read the whole review discussion to understand the context. To test the comprehensive understanding of the constructed schema, we randomly select 30 samples from our representative dataset, and the three authors of this paper independently coded these samples. Among the three coders, we obtain a Kappa agreement score of 0.77 (i.e., substantial). The three coders then discussed the samples with inconsistent codes to reach a consensus. Finally, the remaining data was then coded by one coder.

Analysis I - Timeline of patch linkage. To understand the timeline of patch linkage, we measure patch-linked time and patch-closed time. In terms of the timeline, we hypothesize that the time differs between linkage categories. The patch-linked time is the duration from when reviews start on a patch to the time when the patch link is posted into the review discussion. The patch-closed time is the duration from when a patch link is posted to the time when the review is closed. Then, we perform a statistical analysis to examine if the time differs between linkage categories (i.e., requesting collaboration, sharing information, and pointing out an alternative solution) is different. To do so, we use a Kruskal-Wallis test, i.e., a non-parametric test, to compute the statistical significance.

Results: We observe two main findings. **First, patch linkage for requesting collaboration is relatively less frequent than others.** Table 5.1 shows that only 57 patch linkages (15%) where developers post a patch link with an explicit request for collaboration. Most patch linkages (i.e., 211 patch linkages) are posted for sharing information such as patch dependency and broader context, while the other 100 patch linkages that are for pointing out an alternative solution.

Second, we observe around 4 to 14 days (median) before a review member posts a patch linkage. Regarding the patch-linked time, we find that it takes a relatively long time for review teams to post patch linkage. The median of 11.6, 4.0, and 14.1 days are taken for each linkage type, as shown in Table 5.1. Related to the patch-closed time, we find that the patch with the linkage indicating an alternative solution is more likely to be closed quicker than the other categories. Interestingly, we find that the linkage for requesting

Table 5.2: The collaboration between the source patch and target patch.

Collaboration Direction	Link Type	Occurrence Percent
Source \rightarrow Target	Requesting collaboration	0.72 
	Sharing information	0.57 
	Pointing out an alternative solution	0.47 
	Average	0.57 
Source \leftarrow Target	Requesting collaboration	0.62 
	Sharing information	0.49 
	Pointing out an alternative solution	0.34 
	Average	0.47 

collaboration takes a longer time to be closed compared with other categories. The Kruskal-Wallis test confirms that there is a significant difference (p -value < 0.001) in the patch-closed time between different linkage types.

Takeaway I: Requesting for collaboration when posting a patch linkage is less frequent. Furthermore, we observed a delay from 4 to 14 days before a patch linkage is posted and the linkage for requesting collaboration statistically takes a longer time.

3.2 Collaboration after Patch Linkage

Prior work sheds light that patch linkage can increase the awareness [69]. Yet, little is known if patch linkage can promote collaboration. To better understand this, we investigate whether collaboration occurs after a patch link is posted, and what kinds of collaboration contribution types are made.

Analysis II - Collaboration occurrence. To investigate the collaboration occurrence after the patch linkage, we analyze the set of additional developers who newly join and contribute to the patch after the patch link is posted. We consider both directions of collaboration, i.e., developers who participate in the source patch contribute to the target patch (Source \rightarrow Target) and developers who participate in the target patch contribute to the source patch (Source \leftarrow Target).

To identify the additional developers and direction of collaboration, we first identify the set of developers who contribute (e.g., provide a comment, voting) to the source patch before the patch link is posted (S) and the set of other developers who *only* contribute after the link is posted (S'). Note that S includes the developer who posted the patch link. Similarly, we identify the set of developers who contribute to the target patch based on the time point when the patch link is posted (T and T'). Then, we identify the set of developers in the source patch who contribute to the target patch after the patch link is posted (i.e., Source \rightarrow Target = $S \cap T'$) and the number of developers in the target patch who contribute to the source patch after the patch link is posted (i.e., Source \leftarrow Target = $T \cap S'$).

For example, in Figure 5.1, we will identify the following sets of developers: $S = \{\text{Green, Pink}\}$, $S' = \{\text{Blue}\}$, $T = \{\text{Blue, Orange}\}$, and $T' = \{\text{Green}\}$. Therefore, in this example, the developer *Green* is considered as the one who is from the source patch and contributes to the target patch. Similarly, the developer *Blue* is considered as the one who is from the target patch and contributes to the source patch. Note that since we will analyze the collaboration occurrence across the three link types, we perform this analysis based on the labeled 368 patch linkages.

Results: **Patch linkage with requesting collaboration has a relatively higher percentage of collaboration than the other two types.** Table 5.2 shows the percentage of patch linkages that have at least one developer from the source patch who contributes to the target patch or vice versa. We find that on average, 72% of the patch linkages for requesting collaboration have at least one developer from the source patch who contributes to the target patch (Source \rightarrow Target). Similarly, 62% of the patch linkages for requesting collaboration have at least one developer from the target patch contributes to the source patch (Source \leftarrow Target). On the other hand, the percentages of collaboration in the other two link types are relatively lower than the percentage of the patch linkages for requesting collaboration (i.e., 34%–57%).

Takeaway II: A cross-patch collaboration is more likely to occur when the patch linkage comment is accompanied with a request for collaboration.

Table 5.3: The definition of contribution types and their distribution across the link types. Note that one review message can be labeled with more than one contribution type.

Contribution after Linkage	Definition	Link Type	Percent
Vote	Collaborator votes whether to merge or abandon the patch, i.e., “Code-Review +1”.	Requesting collaboration	0.56
		Sharing information	0.59
		Pointing out an alternative solution	0.61
Specific Comments	Collaborator posts a comment that is directly related to patch change, i.e., typically an inline comment.	Requesting collaboration	0.29
		Sharing information	0.37
		Pointing out an alternative solution	0.31
General Comments	Collaborator posts a generic comment that does not directly relate to or reference any line of code in the patch.	Requesting collaboration	0.43
		Sharing information	0.47
		Pointing out an alternative solution	0.52
Revise	Collaborator uploads revised patches, i.e., “Uploaded patch set 3”.	Requesting collaboration	0.15
		Sharing information	0.09
		Pointing out an alternative solution	0.05

Analysis III - Collaboration contribution type. In addition to the occurrence analysis, we examine what collaboration contributions were made by the additional developers (i.e., $S \cap T'$ and $S' \cap T$). In this work, based on an open discussion with ten random samples and the OpenStack documentation by the first three authors of this paper, we focus on four types of contribution: 1) *Vote*, 2) *Specific Comments*, 3) *General Comments*, and 4) *Revise*. Table 5.3 describes the definition of four contribution types.

To identify the contribution, we extract the contribution information recorded in the review message, i.e., 1,898 contributions are retrieved. In particular, we first use a regular expression to identify each type of contribution. Then, we manually validate the extraction. In addition, we highlight those general comments that are not trivial. For instance, a not trivial general comment is left with “*Patch Set 2: Code-Review-1 i think you should update this file <https://github.com/openstack/neutron/blob/master/doc/requirements.txt> because after the new PTI, doc requirements are moved here.*”

Results: While vote is the most common collaboration contribution type, the revise contribution type in the patch linkage for requesting collaboration is relatively higher than other types of patch linkage. Table 5.3 shows the distribution of contribution types across the link types. We

find that among four contribution types, *Vote* is the most common type (i.e., 56% for requesting collaboration, 59% for sharing information, and 61% for pointing out an alternative solution). The following common contribution type is *General Comments*. Based on our manual validation, we observe that around 20% of these comments are left with not trivial information. For instance, one comment provides advice to fix up the eventlet change, i.e., “*Patch Set 4: OK, fix up the docstring on run_vios_command_as_root and I think the commit message should mention the eventlet change, and then I’m +1.*”. Interestingly, we find that *Revise* contribution type is relatively more frequent in the patch linkage for requesting collaboration (15%) than other two link types (9% and 5%, respectively).

Takeaway III: The cross-patch collaboration via the patch linkage includes voting, writing specific and general comments, and a revision of patches.

4 Threats to Validity

We summarize two key threats. The first threat is regarding the generalizability of the results. Our study only focuses on the ecosystem level using a tool-based code review. We understand that there are not many multi-project review ecosystems similar to OpenStack. However, as open source adoption has grown significantly in the last decade, and numerous companies have built business models around OSS ecosystems [168], we believe it is important to study the ecosystem level.

The second threat is related to the internal threat of our approach. We employ manual analysis for classifying linkage types. The label might be miscoded due to the subjective nature of understanding. To eliminate such a threat, we use the Kappa agreement to measure inter-rater reliability.

5 Challenges and Opportunities

The preliminary results show the potential for this new kind of collaboration that is triggered by a patch linkage. Hence, the study calls for new avenues for research

into this kind of collaboration. In fact, we show that cross-patch collaboration contributions via the patch linkage are non-trivial, with key contributions like voting which affects the review outcome of the target patch, or revising which improves the patch.

There are still open challenges that remain. For instance, the current approach has the threat to include collaborations that may have not been triggered by the patch linkage. Hence, future work needs to address the soundness of our approach. Another challenge may include capturing cross-patch collaborations that do not have patch linkage. This can also be addressed in a bigger study. Furthermore, we would need a developer study to validate the practical implications of the study.

Our work lays out future opportunities for directions on how patch linkage sharing can lead to these new kinds of collaboration. We highlight three below to name a few:

- *Identify heuristics and the information required for a reviewer to contribute to a linked patch.* To gain more practical insights, a survey or interview of the reviewer who posts the link could reveal collaboration barriers and opportunities
- *Investigate the impact of the collaboration on patch quality and code review quality.* To further understand the impact of the collaboration, one promising direction is to explore if the patch involved with contribution via the linkage is likely to decrease the probability of defects.
- *Automatic recovery of links (especially for Duplicate/Alternative Solution Detection).* Provide tool support to early detect or recommend patches to reduce the time taken to identify the link, especially since we find that pointing out an alternative solutions earlier leads to a shorter review time compared to the other link types.

Part III

Automatic Patch Linkage
Detection

6 | Patch Linkage Detection Using Textual Content and File Location Features

Chapter 4 and Chapter 5 provide evidence that patch related links are most frequently shared and the contributions across these links are not trivial. Inspired by these findings, this chapter further explores the feasibility of patch linkage detection, with the goal to support existing code review tools. Patch linkage detection could facilitate developers to identify the needed information efficiently.

1 Introduction

Contemporary software development teams widely adopt code review tools for their software quality assurance [13, 126]. Over the last ten years, review tools have been utilized by well-known Open Source projects such as Android, OpenStack [8, 108], and industry giants such as Google and Microsoft [13, 132]. In contrast to the traditional vigorous face-to-face meetings between small team members, larger teams can adopt a review tool that integrates review discussions [25, 129]. Tools such as Gerrit, Codestriker, and ReviewBoard allow for patches to be submitted, and later assigned to a review team (i.e., patch author and assigned reviewers) [41, 52, 124].

On the downside, massive projects like OpenStack (i.e., which attracts more than 100,000 contributors that spread over 600 repositories [181]) are susceptible

to having their contributors submit patches that may be similar to an already existing patch. This happens due to the parallel and distributed nature of a review-then-commit model [128], where submitted patches achieve a similar goal (i.e., with duplication being the most typical case). This lack of awareness occurs because review teams (i) do not possess the knowledge of other review teams or (ii) do not notice when other authors submit similar work.

To raise awareness of similar patches during review discussions, review teams may post a *linkage* between two patches to notify developers [69]. Existing work shows that the late duplicate patch identification leads to additional maintenance costs and redundant efforts [58]. Specifically, Yu et al. [171] manually studied pull requests from 26 popular projects on Github, observed that on average, 2.5 reviewers participated in the review discussions of redundant pull requests, and 5.2 review comments were generated before the duplicate relation is identified.

Apart from raising awareness of duplication, Hirao et al. [69] claimed that the linkage that is posted in a review discussion can be used to point out three common patch linkage types: *Dependency* (a patch linkage to another patch that it is dependent upon), *Broader Context* (a patch linkage to another patch that provides related resources), and *Alternative Solution* (a patch linkage to another patch that implements similar functionality). Their results suggest that reviewer recommendation can be improved by incorporating information from linked reviews and these linkages could be exploited by code review analytics. Inspired by their work, we argue that the early detection of the patch linkage would help contributors to avoid unnecessary efforts, strengthen collaboration between reviewers, and encourage an effective review process.

In this paper, we extend from the idea of duplication detection to investigate the potential of detecting three different types of patch linkages (i.e., Dependency, Broader Context, and Alternative Solution). Different from patch duplication, we do not look at the similarity of the patch source code, but instead focus on linkages between two patches. Through a case study of three large Open Source projects, i.e., Qt, OpenStack, and Android Open Source Project (AOSP), we are able to extract 11,353 patch linkages in total. We formulate two research questions to guide our study:

- (RQ1) **What is the impact of the patch linkage on the review**

process?

Motivation. Hirao et al. [69] investigated the different types of linkage by classifying their various purposes in the review discussion. However, the impact of the linkage on the review process is unknown. Specifically, we would like to investigate the impact in terms of when the linkage is first notified (i.e., submission time to notification time) and the time taken for that review (i.e., notification time to decision time). Moreover, we would like to investigate whether or not the entire review process (i.e., submission time to decision time) of patches with patch linkages is different from those patches with no patch linkages. To address this RQ, we conduct an exploratory study.

- **(RQ2) *What is the performance of detecting patch linkages?***

Motivation. Motivated by the exploratory study (RQ1), we would like to evaluate whether or not it is promising to detect these patch linkages. Although recent work [88, 123, 162] investigate the feasibility of detecting pull request in the code review setting, it is still unclear that how linkage detection performs in a realistic setting, other than duplication. We assume that patches that are linked tend to share similar textual content and modify similar code locations. Thus, we use these two patch features to construct our detection model borrowed from the information retrieval-based and file location-based recommendation. Taking realistic evaluation into account, we evaluate our models based on four time intervals (i.e., 2, 7, 14, and 30 days). This RQ is split into three sub-questions:

- **(RQ2.1) *What is the performance of using textual content to detect patch linkage?***
- **(RQ2.1) *What is the performance of using file location to detect patch linkage?***
- **(RQ2.3) *How does using more than one feature (textual content and file location) improve the performance of linkage detection?***

The key results of each RQ are: For RQ1, results show that there exists latency in the notification of linked patches (i.e., a median of 1.2 days, 3.1 days,

and 2.2 days for Qt, OpenStack, and AOSP). Results also show that patches with linkages are likely to take a longer time to review when compared to a control group (i.e., patches without linkages), as patches without linkages seem to not require additional reviewing efforts (23 out of 28 patches). Earlier patch linkage notification could make for a more efficient review, especially in detecting the Alternative Solution. For RQ2, results show that combining two features (i.e., textual content and file location) performs better than two separate models. In experiments that span four time intervals, the model performs with promising recall rates (i.e., 24%–68% for Qt, 25%–61% for OpenStack, and 28%–81% for AOSP). The Alternative Solution linkage detection is also promising with relatively high recall rates (i.e., 74%–95% for Qt, 71%–87% for OpenStack, and 77%–94% for AOSP in the Top-10). Reasonable Alternative Solution linkage detection also means that the precision rates are feasible, with 60%–74% for Qt, 43%–67% for OpenStack in the textual content model, and 56%–67% for AOSP in the file location model.

We suggest that improving the awareness between the patches may also increase the likelihood that the linked patches will be identified. To improve the textual content model, developers could be encouraged to increase the natural language or generate a project specific corpus. We also see that linkage detection is promising in a realistic setting, especially for the Alternative Solution linkage. The main contributions of this paper are three-fold: First, our exploratory study provides evidence that the latency of the linkage notification exists. Second, we propose and evaluate our detection models for three different types of patch linkages. Finally, we provide a replication package which includes (a) manually coded patch linkages with various properties and (b) experiment datasets and scripts that can be used to reproduce our detection model.

1.1 Chapter Organization

The remainder of this chapter is structured as follows: Section 2 describes the motivating example. Section 3 presents an exploratory study on the impact of the patch linkage on the review process. Section 4 introduces the patch linkage detection. Section 5 discusses the implications from our findings. Section 6 discloses threats to the validity of our study. Finally, we draw our conclusions in

2 Motivating Example

Figure 6.1 is a real-world example to illustrate how a reviewer posts a linkage to notify the review team there is a similar patch. In the figure, one author (Ken’ichi Ohmichi) submitted a patch # 86771 to the review tool. The patch # 86771 aims to add API tests in a new Nova API called “server-group”. During a review discussion, the reviewer (Zhi Kun Liu) pointed out that this patch addressed an issue similar to another patch: ‘*similar patch <https://review.openstack.org/#/c/84977/>*’ and provided an explicit link of that patch. Once notified, the author (Ken’ichi Ohmichi) proceeded to abandon the patch without any revisions on that same day. From the description of the patch #84977, we observe that its goal is as well to add several tests for “server-group” Nova APIs. From the activity log, three days passed before the reviewer identified and posted the linkage into the patch # 86771. The example provides evidence that latency exists before the team is notified to become aware of the linkage, i.e., around three days are taken in the motivating example.

In terms of the feature similarity between two linked patches, we find that the linked patches from the example share similar textual content and modify similar file locations. As shown in the figure, the two patches use the same keywords (i.e., `server-group`, `Nova`, `v2`, `tests`). Apart from highly similar textual content, both patches touch the same file path (i.e., `tempest/api/compute/base.py`).

3 Impact of Patch Linkage on the Review Process

To answer RQ1, we conduct an exploratory study to investigate the impact of patch linkage on the review process. In this section, we first describe the studied projects (Section 3.1), then we describe the data preparation (Section 3.2), and present the analysis approach (Section 3.3). Finally, we discuss our results (Section 3.4).

Change 86771 – Abandoned

Add server-group v2 API tests

A new Nova API "server-group" has been implemented since Icehouse. This patch adds some API tests for it.

Change-Id: I909294ff7e53446557a8366a9866267711281de5

Author	Ken'ichi Ohmichi	Apr 11, 2014 11:55 AM
Committer	Ken'ichi Ohmichi	Apr 11, 2014 1:06 PM
Commit	211dc2edb865c84fe9ce8f51b5d89850a428186	(gitweb)
Parent(s)	9e9b394b98d8125d47a4cbc26aadbbd33cbffcbb	(gitweb)
Change-Id	I909294ff7e53446557a8366a9866267711281de5	

File Path

Commit Message

tempest/api/compute/base.py

tempest/api/compute/test_server_groups.py

tempest/clients.py

tempest/services/compute/json/server_groups_client.py

History

Zhi Kun Liu
Patch Set 1: Apr 14, 2014

similar patch <https://review.openstack.org/#/c/84977/>

Ken'ichi Ohmichi
Patch Set 1: Abandoned

@Zhi Kun Liu Apr 14, 2014

Thank you very much for point it up!
I will review it later.

(a) Submitted patch with textual content and file location. #86771

Change 84977 - Merged

Add tests for server-group Nova V2 APIs

This patch adds the JSON tests for following server-group Nova V2 APIs

1. server-group create
2. server-group delete
3. server-group list
4. server-group get

It Also adds the clients for JSON

Closes-Bug: 1297933

Change-Id: Iee2717872d58a715f9e012218a67bf040d2abe80

File Path

Commit Message

tempest/api/compute/base.py

tempest/api/compute/servers/test_server_group.py

tempest/services/compute/json/servers_client.py

(b) Target patch with textual content and file location. #84977

Figure 6.1: A real world example to motivate the Alternative Solution linkage between patch #86771 and patch #84977 in OpenStack. The example suggests that the linked patches share similar textual content and modify similar set of file paths.

Table 6.1: Collected dataset including three open source projects: Qt, OpenStack, and AOSP. In total, 11,353 patch linkages are retrieved from these projects.

Time Period	Qt	OpenStack	AOSP
	Aug.2011 ~ Apr.2015	Aug.2011 ~ Nov.2016	Oct.2008 ~ Apr.2015
# Sub-Projects	111	1,528	567
# Files	176,898	158,168	166,931
# Patches	107,858	215,725	59,490
# Revisions	258,066	766,017	113,501
# Lines of Code (LOC) Δ	273,395	8,753,713	35,202,362
# Patch linkages (Hyperlinks)	705	8,048	1,079
# Patch linkages (Review #, Changeid)	253	1,002	266
Total	(0.8%) 958	(4.2%) 9,050	(2.2%) 1,345

Studied Projects

We select studied projects with the two criteria: (1) large software projects actively use review tools (e.g., Gerrit) and (2) projects are representative and have been commonly analyzed in prior work. From the range of open source projects, we select three projects: Qt, OpenStack, and Android Open Source Project (AOSP) as these three projects actively perform code reviews through Gerrit. Qt is a cross-platform application for creating graphical user interfaces. OpenStack is a collaborative platform for cloud computing, which is used by many well-known organizations and companies (e.g., IBM, VMware, and NEC). Finally, AOSP is a mobile operating system developed by Google.

Data Preparation

Our data preparation process consists of three parts: (DP1) patch linkage recovery and filtering, (DP2) ground-truth construction, and (DP3) control group construction. We define the patch linkage as any unique linkage from a patch to another. Let two patches be p_a and p_b , where p_a refers to a patch where the review team posts a linkage into the review discussion and p_b denotes the target patch. Therefore, a linkage from p_a to target p_b and a linkage from p_b to target p_a are counted separately.

- (DP1) Patch linkage recovery and filtering: For studied projects, we adopted the Yang et al. [169] dataset as our original dataset. Table 6.1 shows

our dataset summary (i.e., 107,858 patches for Qt, 215,725 patches for OpenStack, and 59,490 patches for AOSP). To recover the patch linkage, we make sure that all linkage formats should be considered including Changeid, Review # as well as the hyperlinks. For the hyperlink format, we use regular expression patterns to search all discussions (e.g., <https://review.openstack.org/#/c/84977/>). Additionally, we conduct manual checks on each review carefully to ensure the number is correctly related to the Changeid and Review #, but not for other information such as bug id. For example, a detected Changeid would be “*would conflict with 9afb02412eadc567e82a0aca10c6401937d213e9*”, while an example of a detected Review # is “*Replaced by 22724, 22725*”.

Furthermore, we apply three filters to ensure an unbiased dataset. The first filter is to remove cases where the patch author is aware of the linkage. This is done by using two conditions. The first condition is to detect the case where linked patches are written by the same author. The second condition is to detect the case where a patch author cherry-picks or reverts an already known patch. The second filter is to remove patches with incomplete author information (i.e., where the authors could not be retrieved from the REST API). The third filter involves the removal of duplicate patch linkages. We only detect the first instance of a linkage, as the same patch linkage can be used several times by different reviewers during review discussions. After DP1, we are left with 1,345, 9,050, and 958 distinct linkages posted by different authors as shown in Table 6.1.

- (DP2) Ground-truth construction: To construct our ground-truth, we manually classify the three different types of patch linkages (i.e., Alternative Solution, Broader Context, and Dependency) using the linkage taxonomy defined by Hirao et al. [69]. Since our collected data from DP1 is too large to manually examine, we statistically generate a representative sample using a statistical calculator [39] with a 95% confidence level and a margin error of no more than 5%, which is similar to previous empirical studies in the SE domain [9, 66]. We end up with 369 patch linkages, 299 patch linkages, and 274 patch linkages as shown in Table 6.2.

Table 6.2: Ground-Truth based on Hirao et al. [69] and Control Group (patches with no patch linkages).

Project	Ground-Truth					Control Group
	Alternative Solution	Broader Context	Dependency	Others	Sample Size	Non-PatchLink
Qt	93	77	74	30	274	244
OpenStack	97	147	101	24	369	345
AOSP	127	87	58	27	299	272
Total	317	311	233	81	942	861

In the coding process, we first test our comprehension with a statistical agreement. Three authors of this article independently coded a random sample of 30 comments that contain the patch linkages based on the constructed coding scheme. We then measured the Kappa agreement for the coding results. The Kappa statistic is used frequently to measure inter-rater reliability for qualitative (categorical) items [156]. We ended up with a Kappa score of 0.83, which indicates that our agreement is nearly perfect. Two authors then completed the coding for the remaining sample dataset. We classify those patch linkages which do not indicate Broader Context, Dependency, and Alternative Solution into Others Category. After DP2, 317 patches, 311 patches, and 233 patches are labeled as either Alternative Solution, Broader Context, and Dependency.

- *(DP3) Control group construction:* To construct a balanced control group, we then randomly select an equal 861 patches from the ground-truth in DP2, as shown in Table 2. We carefully and manually checked each patch to ensure that there was no indication of a linkage to another patch.

Analysis for RQ1

To analyze the impact of patch linkage on the review process, we focus on two main aspects: (1) comparison among linkage types and (2) comparison against a control group (patches with no patch linkages). Below, we describe our analysis approach for each aspect.

Comparison among linkage types. We now investigate the three different patch linkage types, in terms of their review process (i.e., review time, patch revisions

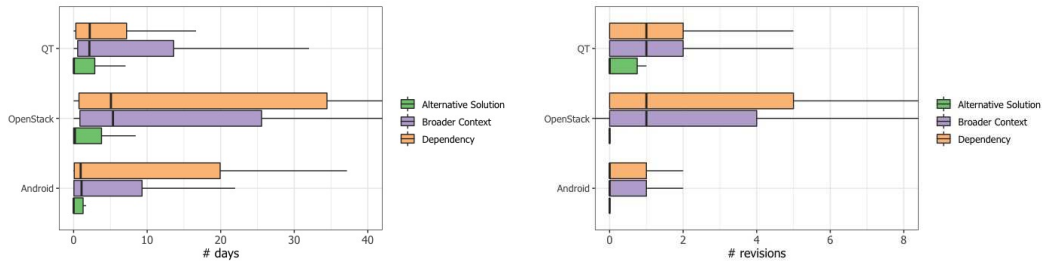
before and after the patch linkage notification). We define four metrics to conduct our statistic analysis as shown below:

- *First-Notify-Time (# days)* is the duration from when the author submits ι_a until when the link to ι_b first appears in the review discussion.
- *First-Notify-Revisions* is the number of patch revisions that occur after the author submits a ι_a until the link to ι_b first appears in the review discussion. Note that the linkage is always mapped to the current patch revision, thus the first submission is counted as the first patch revision.
- *Notify-to-Decision-Time (# days)* is the duration from when the link of ι_b first appears until there is a decision to either abandon or merge ι_a into the codebase.
- *Notify-to-Decision-Revisions* is the number of patch revisions that occur after the link to ι_b first appears until there is a decision to either abandon or merge ι_a into the codebase.

After the metric computation, we then test the hypothesis ‘*a patch with the Alternative Solution linkage is reviewed quicker after notification when compared with other two linkages.*’ We use a Mann-Whitney U test ($\alpha= 0.05$) [94] to validate our hypothesis and investigate the effect size using Cliff’s Delta [40]. Effect size is analyzed as follows: (1) $|\delta| < 0.147$ as Negligible, (2) $0.147 \leq |\delta| < 0.33$ as Small, (3) $0.33 \leq |\delta| < 0.474$ as Medium, or (4) $0.474 \leq |\delta|$ as Large.

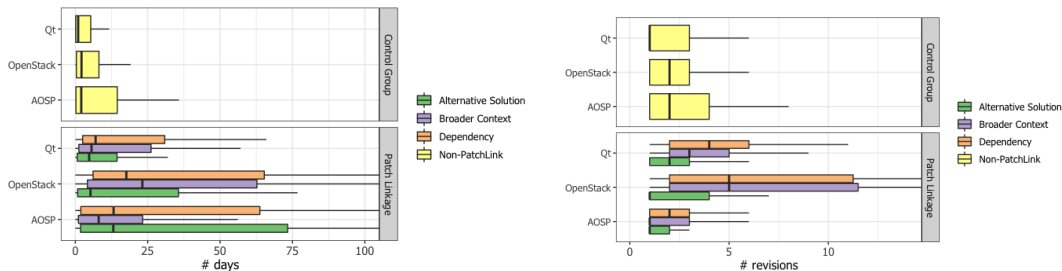
Comparison against control group (patches with no patch linkages). In this aspect, we compare the review process between patches with linkages and patches that do not have any patch linkages with two additional metrics. Two metrics related to the entire review process are defined as follows:

- *Submit-to-Decision-Time (# days)* is the duration from when the author submits a patch until there is a decision to either abandon or merge the patch into the codebase.
- *Submit-to-Decision-Revisions* is the number of patch revisions from when the author submits a patch until there is a decision to either abandon or merge the patch into the codebase.



(a) *Notify-to-Decision-Time* Distribution. (b) *Notify-to-Decision-Revisions* Distribution.

Figure 6.2: Box-plots showing comparison among linkage types (*Notify-to-Decision-Time* and *Notify-to-Decision-Revisions*). The results show that the patch with an Alternative Solution linkage tends to have a quicker review process after the notification, compared with other patch linkages.



(a) *Submit-to-Decision-Time* Distribution. (b) *Submit-to-Decision-Revisions* Distribution.

Figure 6.3: Box-plots showing comparison against a control group (*Submit-to-Decision-Time* and *Submit-to-Decision-Revisions*). The results show that compared to patches having no linkages, patches with linkages tend to take a longer time to complete the review process.

Table 6.3: Statistics showing comparison among linkage types (*First-Notify-Revisions* and *First-Notify-Time*). The results suggest that latency exists in the notification of a patch linkage (i.e., the median of 1.2, 3.1, and 2.2 days for Qt, OpenStack, and AOSP).

Project	Linkage Type	First-Notify-Revisions			First-Notify-Time (# days)		
		Median	Mean	Max	Median	Mean	Max
Qt	Alternative Solution	1	2	18	1.1	16.4	261.6
	Broader Context	2	3	21	1.1	21.4	627.8
	Dependency	2	5	56	2.6	20.4	400.5
	All	2	3	56	1.2	19.2	627.8
OpenStack	Alternative Solution	1	3	29	1.1	26.5	448.0
	Broader Context	2	5	28	3.5	22.6	418.0
	Dependency	3	6	43	4.3	26.8	574.8
	All	2	4	43	3.1	25.0	574.8
AOSP	Alternative Solution	1	2	8	6.2	48.7	669.0
	Broader Context	1	2	9	0.6	13.2	258.7
	Dependency	1	2	14	0.8	32.3	771.0
	All	1	2	14	2.2	33.5	771.0

After the metric computation, we test our hypothesis that ‘*there exists a difference between patches with linkages and those that do not have any patch linkages with regard to the reviewing time*’. We use Kruskal-Wallis non-parametric statistical test [37] to validate our hypothesis.

Results for RQ1

We analyze the impact of the patch linkage on the review process in terms of the comparison among patch linkage types and the comparison against the control group (patches with no patch linkages). Table 3 and Figure 3 show the related results. We now discuss our results below.

Comparison among linkage types. Latency exists in the notification of a patch linkage. Table 6.3 shows the statistics of *First-Notify-Time* and *First-Notify-Revisions*. Using *First-Notify-Time* metric, we find that it takes a median of 1.2, 3.1, and 2.2 days for the linkage to be notified for Qt, OpenStack, and AOSP. Comparing the projects, we find that OpenStack tends to have more linkage

latency compared with the other two projects. We find that the time latency is up to around 3.5 and 4.3 days (i.e., median) for Broader Context and Dependency linkages, while in AOSP, the Alternative Solution linkage takes almost 6.2 days before the patch is made known to the review team. On the other hand, using the *First-Notify-Revisions* metric, we observe that the author already revised the patch before a patch linkage was notified, especially for Qt and OpenStack.

The patch with an Alternative Solution linkage tends to have a quicker process after the notification. Figure 6.2 shows the comparison from the time when the review team is notified until the decision for the patch has been made. Figure 6.2(a) shows evidence that a patch with the Alternative Solution linkage tends to be reviewed quicker when compared to the other patch linkages. For a patch with the Broader Context or Dependency linkage, it takes a longer time to complete the review. Based on our experience with manual coding, a potential reason is due to the nature of both patch linkages. Furthermore, Figure 6.2(b) shows that a patch with the Alternative Solution linkage will be resolved without patch revisions after the notification. For instance, the median of *Notify-to-Decision-Revisions* for the Alternative Solution linkage is 0 for all projects, while for Broader Context and Dependency linkages, patches with them tend to have additional revisions, i.e., one more patch revision in Qt and OpenStack. For the statistical test, Mann-Whitney U tests confirm that ‘*a patch with the Alternative Solution linkage is quicker to complete the review after the notification compared with other two linkages*’ with p -value < 0.001 for the three studied projects. Additionally, the Cliff’s Delta scores indeed prove our hypothesis, i.e., $|\delta|=0.43$ (medium) for Qt, $|\delta|=0.48$ (large) for OpenStack, and $|\delta|=0.36$ (medium) for AOSP.

Comparison against control group (patches with no patch linkages). Compared to those patches with no linkages, patches with linkages tend to take a relatively longer time to complete the review process. Figure 6.3(b) presents the distribution of *Submit-to-Decision-Revisions*. The results show that a patch with the Broader Context or Dependency linkage is likely to have more patch revisions than our control group. On the other hand, Figure 6.3(a) shows that patches with linkages tend to have longer reviewing time compared to those that do not have any patch linkages. It may mean that patches including linkages require more review discussion [46, 68], although our results suggest that the Alternative Solu-

tion linkage does not require as much discussion or revisions. For the statistical test, the values of Kruskal-Wallis tests reveal that *‘there exists a difference between patches with linkages and those that do not have any patch linkages with regard to the reviewing time’*.

Inspired by the results, we would like to further explore why the review process of the patch with no linkages is quicker than the patch with an Alternative Solution linkage. To do so, we randomly select 30 samples (i.e., the median of *Submit-to-Decision-Time* is smaller than the median time for the Alternative Solution linkage), covering three studied projects from the control group. Then we conduct a qualitative analysis to investigate these 30 samples in the aspect of the reviewer participation [150], review divergence [68], and review confusion [46]. The analysis results show that within these 28 patches (ignoring 2 self-approved patches), the reviewer response is timely while few comments are left (i.e., 20 out of 28 patches have less than 3 comments), review divergence rarely exists (i.e., all 28 patches do not have the case where reviewers have divergence), and the confused reviews seldom occur (i.e., 5 out of 28 patches have confusion). The qualitative findings suggest that the review process of the patch with no linkages is relatively straightforward, which could explain why the patch with no linkages is likely to take a shorter time compared to the patch with linkages.

Answering RQ1: Our exploratory results show latency in the notification of linked patches (i.e., a median of 1.2 days, 3.1 days, and 2.2 days for Qt, OpenStack, and AOSP). Results also show that patches with linkages are likely to take a longer time to review when compared to a control group (i.e., patches without linkages), as patches without linkages seem to not require additional reviewing efforts (23 out of 28 patches). Earlier patch linkage notification could make for a more efficient review, especially in detecting the Alternative Solution.

4 Patch Linkage Detection

To answer RQ2, we propose techniques using two patch features (i.e., textual content and file location) to detect the patch linkage. Particularly, we aim to

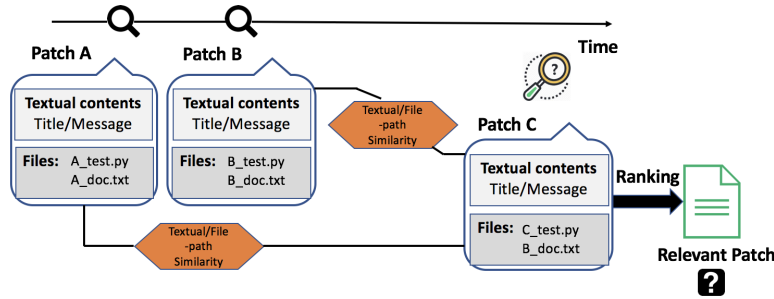


Figure 6.4: The overview of our linkage detection process. To calculate the similarity between the two patches, we focus on the following features: textual content feature (the concatenation of title and description text in a patch) and file location feature (a set of file paths that the patch modifies).

detect the Alternative Solution linkage, which has potential for an efficient review. In this section, we first provide a technique overview (Section 4.1), then introduce our experiment dataset (Section 4.2), and describe the evaluation metrics (Section 4.3). Finally, we discuss our results (Section 4.4).

Technique Overview

Figure 6.4 presents an overview of our proposed process to detect patch linkage, focusing on the patch itself. Our key assumption is that linked patches share similar textual contents and modify similar file locations. We define the textual content as the concatenation of title and description text in a patch. The file location refers to a set of file paths that the patch modifies. For instance, in the figure, the file location of the Patch A is [A_test.py, A_doc.py].

Detection Using Textual Content - The first technique is to compute the textual content similarity. Since the length of the textual content is relatively short, it is appropriate to use the vector space model (VSM), which works well and is widely applied in a similar context such as duplicate bug reports [151]. The first step involves text processing, which includes tokenization, stop word removal, and stemming. Next, in the representation step, our pre-processed text is converted as a vector weight representation. Then we employ a term frequency-inverse document frequency (*tf-idf*) [120] as weighting scheme and calculate the

cosine similarity [95] between two linked patches.

Algorithm 1: Detect linkage based on textual content similarity	
Input	: A new patch (p_n); A set of patches ($Patches$)
Output	: A list of candidate patches
1	$T_n \leftarrow \text{Tokenize}(p_n.\text{textualcontent});$
2	Remove stop words from T_n ;
3	Stem each word in T_n ;
4	$v_n \leftarrow \text{ConstructVSM}(T_n);$
5	for $p_i \in Patches$ do
6	$T_i \leftarrow \text{Tokenize}(p_i.\text{title}, p_i.\text{message});$
7	Remove stop words from T_i ;
8	Stem each word in T_i ;
9	$v_i \leftarrow \text{ConstructVSM}(T_i);$
10	$sim_i \leftarrow \text{CosineSimilarity}(v_n, v_i);$
11	end
12	return A patch list in $Patches$ in the descending order of sim_i ;

Based on Runeson et al. [131], Algorithm 1 details our procedure to rank patches based on textual content similarity. Inputs are new patch p_n and a set of patches $Patches$. In detail, we first perform the text processing for p_n (i.e., the concatenation of title and description text) using NLTK¹ package and well known Porter stemming [166] (lines 1-3). Then we apply gensim² package to create a VSM representation defined as v_n weighted by *tf-idf* (line 4). To process the textual content for each $p_i \in Patches$, we construct a VSM representation defined as v_i (lines 5-9). We compute the similarity between v_n and $v_i \in Patches$ through *cosine similarity* (line 10) and then sort the patches in $Patches$ based on their cosine scores (line 12), returning a resultant list of candidate patches in descending similarity order.

Detection Using File Location - The second technique is to compute the file location similarity. We apply the file location-based model which performs well in reviewer recommendation task [147, 172]. In this model, the key step is to

¹<https://www.nltk.org/>

²<https://radimrehurek.com/gensim/>

Table 6.4: File path comparison technique descriptions. Similar to the work of Thongtanunam et al. [147] four comparison techniques are included: LCP, LCS, LCSustr, and LCSubseq.

Functions	Description	Example
Longest Common Prefix (LCP)	Longest consecutive path components that appears in the beginning of both file paths.	f1 = "src/com/android/settings/LocationSettings.java" f2 = "src/com/android/settings/Utils.java" LCP(f1, f2) = length([src, com, android, settings]) = 4
Longest Common Suffix (LCS)	Longest consecutive path components that appears in the end of both file paths	f1 = "src/imports/undo/undo.pro" f2 = "tests/auto/undo/undo.pro" LCS(f1, f2) = length([undo, undo.pro]) = 2
Longest Common Substring (LCSustr)	Longest consecutive path components that appears in both file paths	f1 = "res/layout/bluetooth_pin_entry.xml" f2 = "tests/res/layout/operator_main.xml" LCSustr(f1, f2) = length([res, layout]) = 2
Longest Common Subsequence (LCSubseq)	Longest path components that appear in both file paths in relative order but not necessarily contiguous	f1 = "apps/CtsVerifier/src/com/android/cts/verifier/sensors/MagnetometerTestActivity.java" f2 = "tests/tests/hardware/src/android/hardware/cts/SensorTest.java" LCSubseq(f1, f2) = length([src, android, cts]) = 3

calculate the file path similarity, with file path \mathcal{U}_a and \mathcal{U}_b computation as follows:

$$FilePathSimilarity_{LCx}(f_a, f_b) = \frac{LCx(f_a, f_b)}{\max(\text{Length}(f_a), \text{Length}(f_b))} \quad (6.1)$$

where $LCx(f_a, f_b)$ function is a parameter specifying how to compare file path components f_a and f_b . Table 6.4 presents the definitions and example calculation for four file path comparison techniques. The four comparison techniques, i.e. Longest Common Prefix (LCP), Longest Common Suffix (LCS), Longest Common Substring (LCSustr), and Longest Common Subsequence (LCSubseq) [60], are used in the LCx function. The comparison function value is normalized by the maximum length of f_a and f_b , i.e., the number of file path components.

Algorithm 2 describes the procedure used to calculate file location similarity. Inputs include a new patch p_n , a set of patches $Patches$, and a file path comparison function LCx . $Files(p)$ represents extracting a modified file set in a patch p (line 1 and 3). Then for each patch p_i in the $Patches$, the file location similarity score between p_n and p_i is computed using the $FilePathSimilarity_{LCx}$ function (lines 4-9). Finally, (sim_i) measures the average value of the file path similarity for every file path in p_n and p_i (line 10). After the patches are sorted based on their similarity scores, the algorithm returns a list of candidate patches in descending similarity order (line 12).

Algorithm 2: Detect linkage based on file location similarity

Input : A new patch (p_n); A set of patches ($Patches$); A file path comparison function LCx

Output : A list of candidate patches

```

1  $F_n \leftarrow Files(p_n)$ ;
2 for  $p_i \in Patches$  do
3    $F_i \leftarrow Files(p_i)$ ;
4    $SimilaritySum \leftarrow 0$ ;
5   for  $f_n \in F_n$  do
6     for  $f_i \in F_i$  do
7        $SimilaritySum \leftarrow SimilaritySum + FilePathSimilarity_{LCx}(f_n, f_i)$ ;
8     end
9   end
10   $sim_i \leftarrow \frac{1}{|F_n| \cdot |F_i|} SimilaritySum$ ;
11 end
12 return A patch list in  $Patches$  in the descending order of  $sim_i$ ;

```

Combination Technique - Prior work [78] successfully shows that the model performance can be improved when combining individual techniques. Similar to Thongtanunam et al. [147], our algorithm is based on the Borda count method [121] for scoring the ranks. The Borda count is a voting technique that simply combines the recommendation lists based on the rank.

- *Combined file location model (fl)*. First, we combine the four string comparison techniques (i.e., LCP, LCS, LCSustr, and LCSubseq) used in the file location model. For each patch candidate p_k , we assign scores based on the rank of p_k in each recommendation list generated from four comparison techniques, with the candidate with the highest ranks receiving the highest scores. For example, if a recommendation list of C_{LCP} votes a patch candidate p_1 as the first rank and the number of total candidates are S , then this patch candidate will get the score of S . The candidate p_1 will get a score of $S - 10$. Given a set of recommendation lists $C \in \{C_{LCP}, C_{LCS}, C_{LCSustr}, C_{LCSubseq}\}$, the score for a patch candidate p_k is defined as follows:

$$Combination(p_k) = \sum_{C_i \in C} S_i - rank(p_k|C_i) \quad (6.2)$$

Table 6.5: Dataset used in experiment based on time intervals. Time intervals are divided into 2 days, 7 days, 14 days, and 30 days.

Project	Ground-truth from RQ1			Experiment Dataset			
	Interval (Days)	#Patches	#File Paths	#Avg. Patches	#Total Patches	#Avg. File Paths	#Total File Paths
Qt	2	40	198	174	6,953	1,868	74,714
	7	74	455	528	39,033	5,948	440,115
	14	96	528	1,018	97,723	10,718	1,028,905
	30	119	632	2,169	258,070	23,193	2,760,000
OpenStack	2	87	259	605	52,637	1,755	152,644
	7	147	689	1,659	243,791	5,199	764,209
	14	196	811	3,398	666,040	10,681	2,093,308
	30	233	1,055	7,267	1,693,125	23,143	5,392,245
AOSP	2	62	708	117	7,259	2,493	154,539
	7	106	971	327	34,629	8,755	928,019
	14	137	2,759	654	89,574	17,017	2,331,345
	30	170	2,997	1,393	236,714	31,659	5,381,939

where S_i is the total number of the recommended patch candidates, rank $(p_k | C_i)$ represents the rank of patch candidate p_k in C_i . The patch linkage recommendation is a list of patch candidates that are ranked according to their scores. To resolve tie-breakers, we reorder candidate patches whose Borda scores are same based on their created time, bubbling up recent patches to the top.

- *Combined File location and Textual Contents.* Second, we combine our candidate lists from the textual content model (tc) and the combined file location model (fl). Following Equation 6.2, we assign scores based on the rank of p_k in each recommendation list from tc and fl ($C \in \{C_{tc}, C_{fl}\}$) and rank the patch candidates according to their Borda scores. Like the combined file location algorithm, we use the patch created time to reorder candidate patches that return the same scores.

Data Preparation

Table 6.5 shows our dataset that is adopted in the model evaluation experiment, using the same three projects from the exploratory study. To construct a more realistic experiment setting, we now use time intervals. For the ground-truth, we collect patches that are created in the same time interval (i.e., 2, 7, 14, and 30 days.) with the ground-truth patch as our experiment dataset (i.e., document set

for the textual similarity analysis). Note that within a patch pair, we always treat the patches whose created time are later as our ground truth. Our ground truth includes patch features (i.e., textual content and file location). Our assumption is that a closer time interval should have a higher textual and file location similarity for that patch linkage. For instance, the OpenStack ground truth includes 87 labeled patches (2 days), 147 labeled patches (7 days), 196 labeled patches (14 days), and 233 labeled patches (30 days).

Evaluation Metrics

To evaluate our approach for patch linkage detection, we use recommendation metrics that are commonly used in software engineering domains [5, 107, 147]. Since each patch only allows one target patch, other evaluation metrics (i.e., Mean Average Precision) are not suitable for this study. The metrics are defined below:

- *Recall@k* calculates the relevant items proportion found in the candidate list. A high recall means that an algorithm returned more relevant results.

$$Recall@k_{all} = \frac{|\mathbb{D}|}{|\mathbb{G}|} \quad (6.3a)$$

$$Recall@k_{type} = \frac{|\mathbb{D}_{type}|}{|\mathbb{G}_{type}|} \quad (6.3b)$$

The Equation 6.3a defines how the Recall@k is calculated for all patch linkage types. In this formula, \mathbb{D} refers to a set of ground truth (patches) whose linked patches are retrieved in the candidate list, while \mathbb{G} refers to a set of ground truth (patches) used in the experiment. Equation 6.3b computes the Recall@k for specific linkage type. \mathbb{D}_{type} is a set of ground truth (patches) labeled with one linkage type whose linked patches are retrieved in the candidate list, while \mathbb{G}_{type} refers to a set of ground truth (patches) labeled with one linkage type used in the experiment. Inspired by the previous study [15, 90], we set k to range from 1 to 10.

- *Precision@k_{type}* calculates the ratio of correctly predicted positive observations to the total predicted positive observations. A high precision relates to the low false-positive rate.

$$Precision@k_{type} = \frac{|\mathbb{D}_{type}|}{|\mathbb{D}|} \quad (6.4)$$

Equation 6.4 defines our precision calculation for specific linkage type, where \mathbb{D}_{type} refers to a set of ground truth (patches) labeled with one linkage type whose linked patches are retrieved in the candidate list, while \mathbb{D} refers to a set of ground truth (patches) whose linked patches are retrieved in the candidate list.

- *Mean Reciprocal Rank (MRR@k)* calculates an average of the reciprocal ranks for correctly detected patches in a candidate list. A high MRR score indicates that the first true positive is being returned closer to the top list.

$$MRR@k = \frac{1}{|\mathbb{G}|} \sum_{\mathfrak{d} \in \mathbb{G}} \frac{1}{rank(candidates(\mathfrak{d}))} \quad (6.5)$$

Equation 6.5 explains the MRR@k calculation. Given a set of ground truth \mathbb{G} (patches) used in the experiment, the $rank(candidates(\mathfrak{d}))$ refers to the rank position value of a ground truth \mathfrak{d} (a patch) whose linked patch is retrieved in the candidate list. If there is no correctly retrieved patch in the candidate list, the value of $\frac{1}{rank(candidates(\mathfrak{d}))}$ will be 0.

Results for RQ2

Results for RQ2.1 – To evaluate the performance of the textual content model, we compute the *Recall@k_{all}* for each project, the *Recall@k_{type}* for each patch linkage type, and the *MeanReciprocalRank (MRR)* scores. Figure 6.5 and Table 6.6 show the evaluation results of the textual content model. We summarize two main findings from this table.

The recall rates for the textual content model decrease when the time intervals get larger. Figure 6.5 shows the recall rates of the textual content model covering all patch linkage types from Top-1 to Top-10 for Qt, OpenStack, and AOSP. We observe that when the time interval is set as 2 days, the recall rates can range from 40% to 53%, 28% to 55%, and 40% to 76% for these projects. While when the time intervals are set as 7, 14, 30 days, the recall rates keep increasing

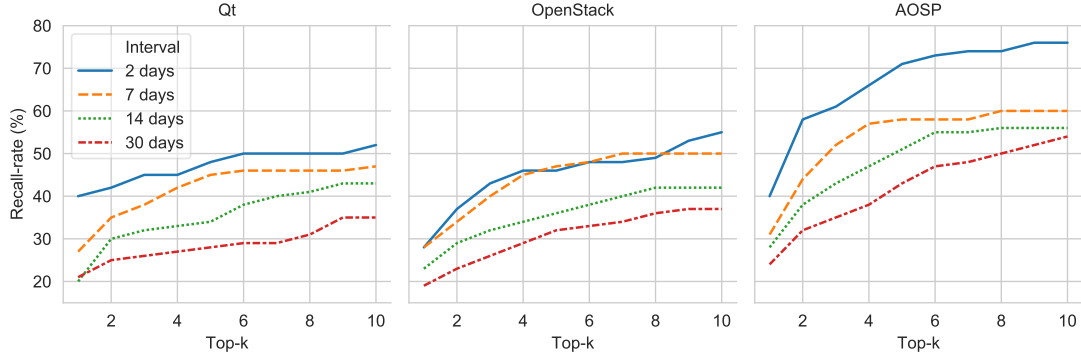


Figure 6.5: $Recall@k_{all}$ for the detection based on textual content (the concatenation of title and description text in a patch). The results suggest that the recall rates for the textual content model decrease when the time intervals get larger.

Table 6.6: Evaluation results ($Recall@k_{type}$ and $MRR@10$) for the textual content model. The recall rates for detecting the Alternative Solution linkage range from 34% to 69%, 34% to 80%, and 31% to 82% for Qt, OpenStack, and AOSP.

Project	Interval (Days)	Alternative Solution Recall				Broader Context Recall				Dependency Recall				MRR
		Top-1	Top-3	Top-5	Top-10	Top-1	Top-3	Top-5	Top-10	Top-1	Top-3	Top-5	Top-10	Top-10
Qt	2	58%	63%	69%	69%	34%	45%	45%	56%	17%	17%	17%	25%	43%
	7	45%	62%	65%	68%	20%	25%	40%	45%	9%	18%	22%	22%	34%
	14	34%	54%	54%	61%	16%	24%	32%	44%	3%	10%	10%	16%	27%
	30	35%	45%	47%	55%	19%	23%	23%	32%	5%	5%	8%	13%	25%
OpenStack	2	54%	67%	70%	80%	13%	38%	41%	47%	17%	21%	25%	38%	36%
	7	56%	70%	73%	79%	13%	28%	40%	44%	18%	25%	29%	29%	36%
	14	46%	62%	62%	63%	15%	20%	29%	40%	14%	19%	22%	26%	29%
	30	34%	49%	56%	60%	15%	18%	22%	30%	12%	14%	20%	24%	24%
AOSP	2	43%	68%	75%	82%	30%	50%	65%	65%	50%	64%	71%	79%	53%
	7	42%	61%	70%	74%	14%	42%	45%	47%	38%	50%	54%	54%	42%
	14	43%	56%	67%	71%	10%	30%	37%	40%	26%	36%	39%	48%	38%
	30	31%	48%	57%	64%	13%	22%	31%	46%	22%	27%	33%	43%	32%

compared to the interval of 2 days. For instance, in the interval of 30 days, the recall rates vary from 21% to 35%, 19% to 38%, and 24% to 54% for the three projects. These results indicate that the selection of the time interval affects the textual content model performance, as more submitted patches could potentially increase the complexity of the textual corpus.

The detection of the Alternative Solution linkage outperforms the other patch linkage types with relatively high recall rates. Table 6.6 reports the recall rates for the Top-1, Top-3, Top-5, and Top-10 of three studied linkage types. We observe that the recall rates of the Alternative Solution linkages can range from 34% to 69%, 34% to 80%, and 31% to 82% for Qt, OpenStack, and AOSP. On the other hand, the recall rates of the Broader Context linkage vary from 16% to 56%, 13% to 47%, and 10% to 65%. For the Dependency linkage, the recall rates range from 3% to 25%, 12% to 38%, and 22% to 79%. These results suggest that the linked patches with an Alternative Solution linkage share more similar textual contents than the other two linkage types.

Results for RQ2.2 – We use the combination technique as described in Section 4.1 to combine recommendation lists from four string comparison techniques (i.e., LCP, LCS, LCSubstr, and LCSubseq). To evaluate the performance of the file location model, similarly, we compute the $Recall@k_{all}$ for each project, the $Recall@k_{type}$ for each patch linkage type, and the *MeanReciprocalRank (MRR)* scores. Figure 6.6 and Table 6.7 show the evaluation results of the file location model. We now discuss our findings below.

The detection using file location feature overall can not perform as well as the model using the textual content feature. Figure 6.6 shows the recall rates of the file location model from Top-1 to Top-10 for Qt, OpenStack, and AOSP. As we can see, the overall performance is visually lower than the model using textual content (except for the cases in the intervals of 14 days and 30 days for Qt). For example, when the interval is set as 2 days, the recall rates range from 30% to 50%, 24% to 32%, and 29% to 48% for the three projects. Table 6.7 shows the recall rates for each patch linkage type using file location. We observe that the Alternative Solution linkage still outperforms the other linkage types, i.e., the $Recall@10$ rates of 69%, 50%, and 72% in the interval of 2 days for three projects. Such result indicates that the linked patches with an Alternative Solution linkage

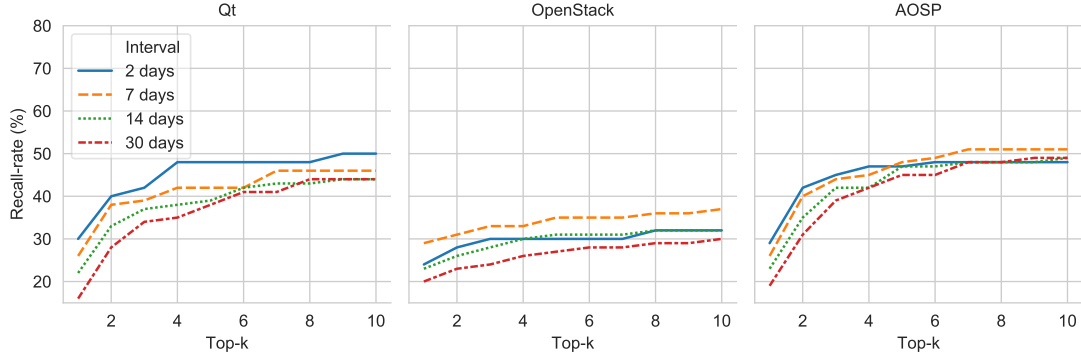


Figure 6.6: $Recall@k_{all}$ for the detection based on file location (a set of file paths that the patch modifies). The results show that the file location as a patch feature overall does not perform as well as the textual content feature with relatively lower recall rates (16%–50% for Qt, 19%–37% for OpenStack, and 19%–51% for AOSP).

Table 6.7: Evaluation results ($Recall@k_{type}$ and $MRR@10$) for the file location model. The recall rates for detecting the Alternative Solution linkage range from 23% to 69%, 34% to 60%, and 26% to 72% for Qt, OpenStack, and AOSP.

Project	Interval (days)	Alternative Solution Recall				Broader Context Recall				Dependency Recall				MRR
		Top-1	Top-3	Top-5	Top-10	Top-1	Top-3	Top-5	Top-10	Top-1	Top-3	Top-5	Top-10	
Qt	2	37%	53%	63%	69%	22%	45%	45%	45%	25%	25%	25%	25%	50%
	7	29%	42%	45%	55%	20%	45%	50%	50%	26%	31%	31%	31%	34%
	14	24%	46%	49%	56%	24%	44%	48%	48%	16%	19%	26%	26%	30%
	30	23%	41%	47%	57%	13%	39%	42%	45%	10%	21%	21%	26%	26%
OpenStack	2	34%	47%	47%	50%	24%	24%	24%	27%	13%	17%	17%	17%	27%
	7	47%	53%	55%	60%	22%	27%	29%	29%	18%	18%	20%	22%	31%
	14	40%	51%	53%	56%	19%	23%	28%	28%	12%	14%	15%	15%	30%
	30	35%	40%	46%	52%	17%	20%	23%	25%	9%	11%	14%	16%	23%
AOSP	2	43%	64%	68%	72%	20%	30%	30%	30%	14%	29%	29%	29%	37%
	7	37%	59%	63%	68%	17%	36%	36%	36%	21%	29%	38%	42%	36%
	14	32%	57%	59%	61%	9%	30%	37%	37%	23%	26%	32%	39%	33%
	30	26%	49%	55%	60%	9%	29%	36%	40%	19%	33%	35%	41%	30%

Table 6.8: Evaluation results ($Recall@k_{type}$ and $MRR@10$) for the feature combination model. The higher $MRR@10$ scores show that the model can detect more patch linkages in higher ranks (i.e., 33%–44% for Qt, 33%–43% for OpenStack, and 40%–53% for AOSP).

Project	Interval (days)	Alternative Solution Recall				Broader Context Recall				Dependency Recall				MRR
		Top-1	Top-3	Top-5	Top-10	Top-1	Top-3	Top-5	Top-10	Top-1	Top-3	Top-5	Top-10	Top-10
Qt	2	32%	74%	84%	95%	33%	67%	67%	67%	17%	25%	25%	25%	44%
	7	35%	68%	78%	81%	30%	45%	55%	60%	17%	35%	35%	39%	41%
	14	37%	59%	66%	76%	28%	52%	56%	56%	10%	23%	26%	26%	36%
	30	39%	55%	63%	76%	23%	48%	52%	55%	5%	21%	23%	23%	33%
OpenStack	2	50%	77%	84%	84%	21%	39%	46%	55%	29%	34%	34%	38%	43%
	7	60%	83%	83%	85%	20%	42%	47%	60%	18%	34%	34%	36%	43%
	14	47%	70%	74%	79%	21%	34%	39%	50%	12%	25%	27%	31%	35%
	30	39%	59%	64%	71%	22%	31%	34%	41%	14%	23%	24%	29%	33%
AOSP	2	50%	72%	79%	89%	25%	45%	65%	75%	29%	64%	71%	71%	53%
	7	44%	70%	87%	94%	17%	47%	58%	58%	29%	46%	54%	63%	46%
	14	49%	68%	79%	90%	14%	40%	49%	56%	26%	39%	42%	48%	44%
	30	41%	64%	68%	77%	16%	35%	44%	56%	24%	35%	46%	54%	40%

are more likely to modify similar file locations.

Results for RQ2.3 – We use the combination technique as described in Section 4.1 to combine two features, based on two Top-10 candidate lists from the separate models (textual content model and file location model). To evaluate the performance of the model combined with two features, we compute the $Recall@k_{all}$ for the three projects, the $Recall@k_{type}$ for linkage types, and the *Mean Reciprocal Rank* (MRR) scores. In addition, we would like to investigate the $Precision@k_{type}$ of the Alternative Solution linkage, since the separate models were able to detect the Alternative Solution linkage best. Figure 6.7, Table 8, and Table 9 show the evaluation results related to the feature combination model. We now discuss two main findings below.

The model combined with two features performs better than two separate models using a single feature. Figure 6.2 shows the combination model performance for three projects at different time intervals. From the interval of 2 days to the interval of 30 days, the feature combination model performs with a recall rate, ranging from 24% to 68% for Qt, 25% to 61% for OpenStack, and 28% to 81% for AOSP. Moreover, Table 8 records the $MRR@10$ scores for each studied project. Compared to the $MRR@10$ scores in separate models, we find that the $MRR@10$ scores as well increase in the feature combination model. For instance, in the OpenStack project, the $MRR@10$ scores are between 33% and 43%

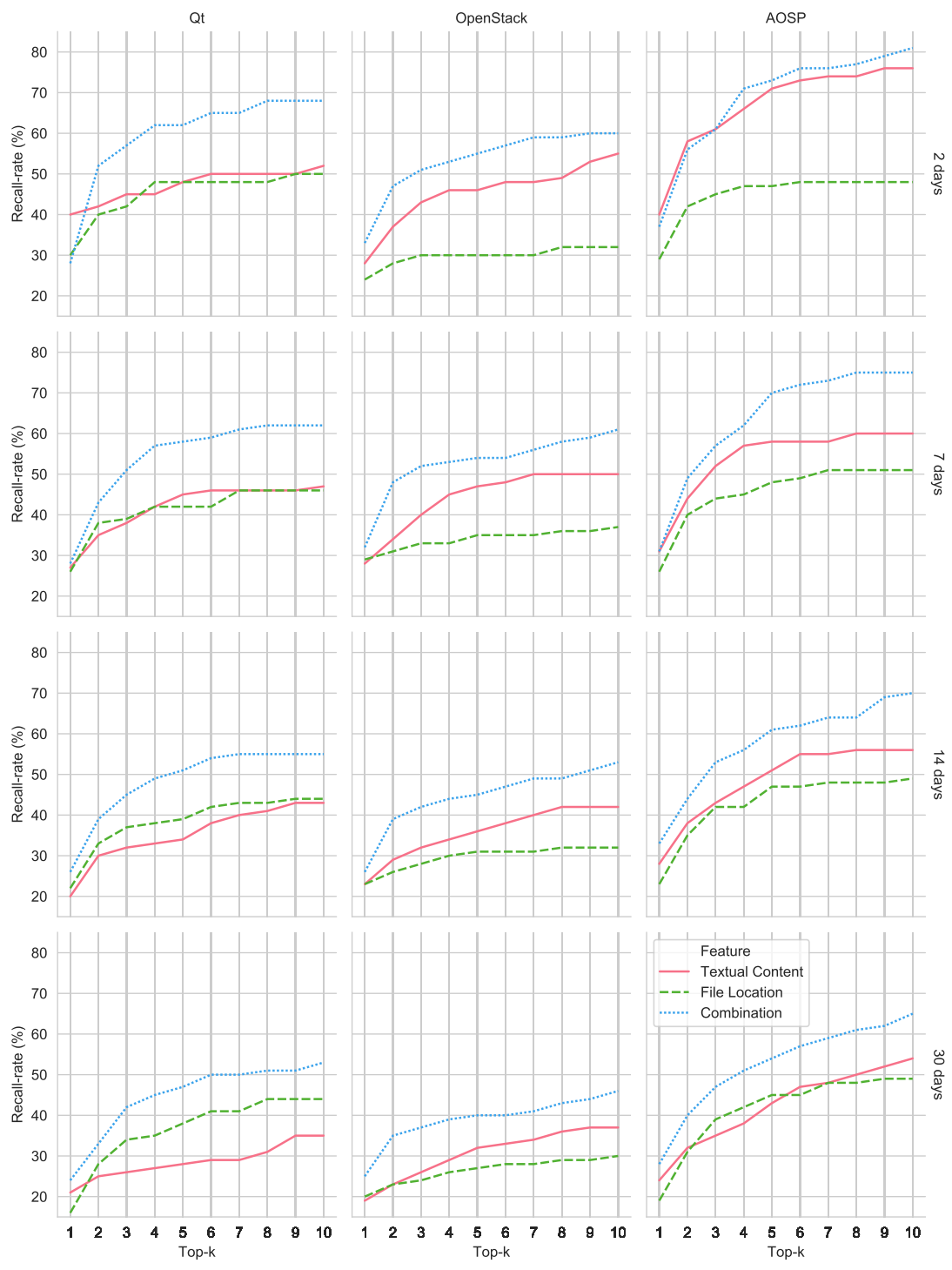


Figure 6.7: $Recall@k_{all}$ for the detection based on file location and textual content for studied projects.

Table 6.9: Evaluation results ($Precision@k_{type}$) for the Alternative Solution linkage. The results show that precision is relatively higher in the separate models than in the feature combination models (i.e., 60%–74% and 43%–67% in the textual content model for Qt and OpenStack; 56%–67% in the file location model for AOSP).

Project	Interval (Days)	Textual Content				File Location				Combination			
		Top-1	Top-3	Top-5	Top-10	Top-1	Top-3	Top-5	Top-10	Top-1	Top-3	Top-5	Top-10
Qt	2	69%	67%	68%	62%	58%	59%	68%	70%	55%	61%	64%	67%
	7	70%	68%	63%	60%	47%	45%	45%	50%	52%	55%	56%	54%
	14	74%	71%	67%	61%	47%	53%	53%	54%	60%	55%	55%	59%
	30	68%	71%	72%	64%	58%	50%	53%	54%	68%	54%	55%	59%
OpenStack	2	67%	54%	53%	50%	48%	54%	54%	54%	52%	52%	52%	48%
	7	64%	56%	49%	50%	52%	52%	51%	52%	60%	51%	49%	45%
	14	57%	56%	50%	43%	51%	51%	48%	48%	53%	49%	47%	44%
	30	53%	57%	53%	49%	54%	53%	52%	52%	48%	49%	49%	47%
AOSP	2	48%	53%	50%	49%	67%	64%	66%	67%	61%	53%	49%	52%
	7	58%	51%	53%	53%	61%	57%	57%	58%	61%	53%	54%	54%
	14	69%	59%	60%	59%	65%	63%	59%	58%	69%	60%	60%	59%
	30	63%	63%	61%	55%	63%	58%	57%	56%	64%	61%	56%	54%

while the scores are between 24% and 36%, 23% and 31% in the textual content model and file location model. The higher MRR scores indicate that the feature combination model can detect more patch linkages with higher ranks.

The detection of the Alternative Solution linkage is promising with high recall rates and possible precision rates. Table 6.8 shows the recall rates of three linkage types detected at different intervals. We observe that with the combination of two features, the recall rates of the Alternative Solution linkage increase to 76%–95%, 71%–85%, and 77%–94% in the Top-10 for Qt, OpenStack, and Android, which is much higher than the other two patch linkages (i.e., 23%–39%, 29%–38%, and 48%–71% for the Dependency linkage). Table 6.9 specifically calculates the $Precision@k_{type}$ for the Alternative Solution linkage. The table shows that overall the precision does not increase in the combination model compared to those separate models. For example, for Qt and OpenStack projects, the textual content model generally performs better with the precision rates being from 60% to 74% and 43% to 67%. On the other hand, for AOSP project, the file location model achieves higher precision rates, ranging from 56% to 67%.

Answering RQ2: Results show that combining two features (i.e., textual content and file location) performs better than two separate models. In experiments that span four time intervals (i.e., 2, 7, 14, and 30 days), the model performs with promising recall rates (i.e., 24%–68% for Qt, 25%–61% for OpenStack, and 28%–81% for AOSP). The Alternative Solution linkage detection is also promising with relatively high recall rates (i.e., 74%–95% for Qt, 71%–87% for OpenStack, and 77%–94% for AOSP in the Top-10). Reasonable Alternative Solution linkage detection also means that the precision rates are feasible, with 60%–74% for Qt, 43%–67% for OpenStack in the textual content model, and 56%–67% for AOSP in the file location model.

5 Discussion

We now discuss four insights from our empirical experiments and the interpretation of the results.

1. *Latency of patch linkage notification.* The exploratory results provide evidence that there is latency until the review team is notified with the patch linkage. Yu et al. [171] similarly recognized this detection latency in the case of duplicate pull requests. For researchers, we believe that this study can be used to motivate the need for early detection of potential patch linkages, which could be in the form of automatic tool support. Furthermore, we are unsure if the latency is due to team awareness or that the review is not yet fully understood by the team that is reviewing the patch. For the review team, maybe part of the workflow should include the search for similar patches that could be linked, especially those patches that introduce alternative solutions. Potential future work would be studying reasons for the latency and its impact on the review process. A possible example includes latency caused by waiting for a reviewer.
2. *Team awareness enhancement.* Active awareness can improve teams' trust, relationships, and efficiency [87] as review discussions particularly serve as

an important mechanism for coordination and collaboration between team members. Bacchelli and Bird [13] reported that practitioners at Microsoft view team awareness as an important motivation for conducting code review. According to the results in our exploratory study, we notice that the patch with an Alternative solution linkage tends to finish the review process quicker after the linkage has been established. For researchers, understanding the role that linkages play in team awareness might be a potential future research avenue. For the practitioners, we suggest that improving the awareness between the patches may also increase the likelihood that the linked patches will be identified. This can be done by making sure there are overlaps in the review teams.

3. *Detection improvements.* Results show that using more than one feature (i.e., textual content and file location) can improve performance. We also find that the textual information in a patch is complex and very difficult to standardize English. As such, using the typical information retrieval methods might not be ideal. To improve the textual content model, developers could be encouraged to increase the natural language or generate a project specific corpus. For researchers, other more sophisticated textual similarity could be calculated. Furthermore, looking at the similarity of the patch contents themselves (i.e., source code) might be interesting for improvement of the model. For developers, our research shows the potential for related patches, which is already adopted in practice. For example, the Gerrit OpenStack web interface is able to show (i) related patches and (ii) same topic patches for a patch under review.
4. *Detection in Practice.* The linkage detection is promising in a realistic setting, especially for the Alternative Solution linkage. Our results in Table 6.8 and Table 6.9 from RQ2 show that the detection of the Alternative Solution linkage can reach the recall rates of 74%–95%, 71%–87%, and 77%–94% in the Top-10 for Qt, OpenStack, and AOSP. Although the precision rates are possible (60%–74%, Qt; 43%–67%, OpenStack; 56%–67%, AOSP), we acknowledge that our model still has room for improvement. One possible clue is to set the threshold for the textual similarity, as the similarity shared

Table 6.10: Evaluation results using sample-based datasets. The statistics show that in the sample-based evaluation, the file location model performs better than the textual content model.

Project	Feature	All				Alternative Solution				Broader Context				Dependency			
		Top-1	Top-3	Top-5	Top-10	Top-1	Top-3	Top-5	Top-10	Top-1	Top-3	Top-5	Top-10	Top-1	Top-3	Top-5	Top-10
Qt	Textual Content	24%	32%	36%	43%	39%	45%	53%	61%	14%	23%	27%	35%	15%	23%	23%	27%
	File Location	31%	44%	48%	56%	44%	56%	61%	71%	23%	36%	39%	45%	22%	37%	42%	47%
OpenStack	Textual Content	24%	32%	37%	44%	45%	58%	64%	68%	16%	24%	28%	34%	13%	19%	25%	25%
	File Location	29%	39%	44%	52%	53%	61%	66%	75%	20%	31%	35%	44%	19%	29%	35%	42%
Android	Textual Content	29%	41%	47%	56%	35%	52%	58%	64%	18%	30%	36%	49%	32%	37%	40%	47%
	File Location	40%	51%	57%	65%	54%	65%	72%	79%	28%	38%	46%	53%	26%	40%	40%	53%

by different linkage types could be different.

At the same time, we would like to investigate whether there is a difference between the realistic experiment and the sample-based experiment. To do so, following the technique employed by previous work, we apply the 20% and 80% ratio to construct the sample-based datasets. In detail, 20% are ground truth patches with labeled linkage types, while 80% are randomly selected patches with no linkages. Table 6.10 shows the model evaluation results using sample-based datasets. Two observations are summarized from the table. The first observation is that the file location models perform better than the textual content models. In contrast, in our time interval based experiment, the results suggest that the textual content models outperform. One potential reason leading to the difference could be that the file location is not as flexible as the textual content in the reality. The one same file location can be edited by more than one developer with different purposes. The second observation is that in all, the experiment based on the time intervals performs better than the one using the sample-based datasets. This observation indicates that patches that are closer in specific time intervals are more likely to share similar textual contents and file locations than patches that are relatively longer time apart. Inspired by the existing results from the time interval based experiment, the immediate future work is to find out whether or not the implementation of our proposed models would be useful by studying real users.

6 Threats to Validity

In this section, we describe the threats to the validity of both exploratory study and linkage detection study.

Construct validity. We summarize the construct validity into two parts according to our exploratory study and patch linkage detection study. In the exploratory study, the threat exists in our approach to recover patch linkages. We use regular expressions to identify the patch linkages in the form of Changeid, Review #, and hyperlinks. However, it is prone to generate false positives which means these formats cannot assure the target is a patch, i.e., Review # could be bug id. To mitigate the risk of such false positives, we conducted a careful manual check as described in Section 3.2 (DP1) Patch linkage recovery and filtering. With regards to the patch linkage detection, one threat is the time interval selection in our experiment dataset, as we cannot cover all patch linkages using the current time intervals. However, we believe that such an experiment setting is closed to reality and could provide insight into whether or not the time interval has influence on the linkage detection. Another threat related to the detection experiment is patch feature selection as we do not include other features such as source code. Our assumption is that the file location provides some heuristic of the source code. We will list this as an immediate future study to improve our models.

Internal validity. In our exploratory study, a potential threat exists in our qualitative method to manually classify types of patch linkages. Patch linkages may be mis-coded due to the subjective nature of understanding the coding schema by the co-authors. To mitigate this threat, we took a systematic approach to first test our comprehension with 30 samples using Kappa agreement scores by three separate individuals. Only until the Kappa score reaches more than 0.8 (nearly perfect), we were able to complete the rest of the sample dataset. Another threat is related to the tool selection for our study, as they may change the results of the study. In our study, we rely on various tools such as gensim for the tf-idf calculation and the Mann-Whitney U test for the statistical significance testing. We are however confident, as these tools have been widely used in other studies.

External validity. External validity is related to the result generalization. Our empirical findings are based on three open source projects using the Gerrit code review tool, i.e., Qt, OpenStack, and AOSP. However, it is unknown whether our results can be generalized to the other tool-based code review such as pull-based code review. For those smaller commercial projects or projects that adopt the commit-then-review style, unawareness of patch linkage could not be an issue. Thus, the patch linkage issue might only affect large open source projects or projects that adopt the review-then-commit style. Due to the popular rise of tools (i.e., Gerrit and Pull-Requests) that support the review-then-commit style, I believe that the problem still does affect many software development teams. In order to encourage future replication studies, our replication package is available at <https://github.com/dong-w/Replication-Patch-Linkage>.

7 Summary

Our study provides evidence that latency exists in the notification of a patch linkage, and confirms that patch linkage detection is promising, with likely improvements if the practice of posting linkages becomes more prevalent. This study provides many open avenues for future work, which includes (i) studying the role that linkages play in a review team awareness, (ii) improvement of the detection using alternative approaches with additional features (i.e., source code), (iii) improving our feature metrics, and (iv) studying real users through the implementation of our proposed models. From our discussions, this paper lays the groundwork for future research on how to increase patch linkage awareness, which may facilitate a more efficient review.

7 | Conclusion

Code review plays a vital role in software quality assurance. Recently, the lightweight variant of code review process, known as Modern Code Review (MCR) has been widely adopted in both open source projects and industrial projects. MCR is a collaborative process, where developers and authors relying on review tools conduct an online discussion asynchronously at the distributed teams. Different from the traditional code inspection, the motivation of MCR is not only limited to defect finding, but also includes knowledge transfer, context understanding, and team transparency.

An effective review requires a proper understanding. However, it is challenging to identify and acquire the needed information to conduct a review. This thesis continues on this line of better supporting the code review process and presume that the practice of link sharing can fulfill the information needs. To address this, three empirical studies are performed around understanding the nature of link sharing and facilitating the traceability of review links. To the end, a mapping study is presented to figure out the existent CR research gap and provide the future direction for researchers. In the remainder of this chapter, we outline the implications and suggestions of this thesis and lay out opportunities for future research.

1 Contributions

The outcomes drawn from this thesis could be beneficial for both practitioners and researchers. Below, the contribution along with suggestions are summarized for each part as follows:

Part I Map for Future Code Review Research

- Mapping results show that 65% of CR researches published in premium SE venues use sound evaluation methodology, targeting particularly socio-technical and understanding of CR processes. However, there is a lack of papers that report the experience and propose solutions to deal with CR problems.

Suggestion. Practitioners are encouraged to more emphasize and share the experience with CR. At the same time, future research could propose more solution tool support to facilitate the developers to make the CR process more efficient.

- CR research not only relies on the data sources from the CR process but also largely uses the data sources from the software development process (i.e., issue tracking system and GitHub). Only 50% of CR papers that use the quantitative method or mixed-method provide the public datasets (i.e., the replication links are provided in the papers).

Suggestion. To promote the validity of scientific findings, future researches are encouraged to strive for a replicable dataset.

- The SE topic of quality assurance is more likely to use CR metrics to conduct the research. In addition, experience and code are the two most frequently used metric sets. From the mapping between metric sets and research topics, results show that it has the potential to benchmark the metric based CR research, as different research topics tend to use particular metric sets.

Suggestion. The researchers are encouraged to take the existing metrics into account when conducting a certain topic.

Part II Link Sharing in Code Review

- The link targets are diverse from different locations with the complexity of projects. One survey respondent cited that *“This (links) is useful for example for newcomers that may have missed guidelines or did not found the corresponding bug report when doing submitting a fix.”*

Suggestion. Patch authors (especially for newcomers or novice developers) should read the project related guidelines to be familiar with the environment before their submission.

- The study indicates that the practice of sharing links can fulfill various information needs, i.e., seven intentions are classified. One respondent from our survey commented that *“Links usually provide a concise and clear answer compared to trying to explain it in prose.”*

Suggestion. Reviewers are encouraged to share links as such information may help to conduct a more efficient review and also assist with mentoring new members to the review team.

- Results indicate that link sharing is becoming a popular practice during review discussions in the MCR. In addition, the existing functionality to automatically recommend related patches falls short. Such limitation is also pointed by the responses *“It seems useful for seeing what changes are submitted at a similar point in the change history, but doesn’t seem useful for finding patches that are related by content (e.g., changing the same feature) but separated by longer periods of time.”*

Suggestion. New opportunities arise for researchers to develop tool support, especially to recommend related and useful links for both the patch author and review teams in order to facilitate the review process.

Part III Automatic Patch Linkage Detection

- Results show latency in the notification of linked patches. Results also show that patches with linkages are likely to take a longer time to review when compared to a control group (i.e., patches without linkages), as patches without linkages seem to not require additional reviewing efforts.

Suggestion. Earlier patch linkage notification could make for a more efficient review, especially in detecting the Alternative Solution.

- Results show that combining two features (i.e., textual content and file location) performs better than two separate models. The Alternative Solution linkage detection is also promising with relatively high recall rates.

Suggestion. The implementation of the proposed models using textual content and file location features is possible, while there is still room for precision improvement.

2 Opportunities for Future Work

We believe that this thesis makes a major contribution to improving the code review process. However, there are many open challenges for future work. Below, we outline a list of potential opportunities.

(I) Studying the causality between the link sharing and the review outcomes. In Chapter 3, statistical model results show that the internal link has a significant correlation (but relatively weak) with the review time. However, the external link is not significantly correlated with the review time. Furthermore, the number of internal links has an increasing relationship with the review time. Though we can observe the correlation between explanatory and dependent variables, the causal effects of link sharing on the review time cannot be represented. Thus, future in-depth qualitative analysis or experimental studies are needed so as to better understand the reasons and effects of link sharing impact. Apart from the review time, future research could investigate whether the link sharing has effect on the code change quality.

(II) Understanding the challenges of cross-patch collaboration. In Chapter 4, empirical results show that after the patch linkage is provided, developers are likely to collaborate with each other. Moreover, collaboration contributions are non-trivial, with key contributions like voting which affects the review outcome of the target patch, or revising which improves the patch. Future research should investigate if there exist any challenges when developers tend to collaborate once the patch linkage is provided (i.e., human factors including workload, familiarly with codes, social distance, etc.). The effective collaboration could make the review process more efficient.

(III) Automatic recovery of links to recommend the needed information. This thesis shows evidence that the links is useful and it has potential to automatically identify the specific links (i.e., patch linkage). In Chapter 3, empirical results show that links regarding project guidelines is often shared as will, which indicates that newcomers may be not familiar with the environment before their submission. Such observation is also supported by our survey results. Thus, it is a necessity to develop a functionality or chat-bot that can systematically manage project related and automatically recommend project related links to mentor newcomers. In Chapter 5, model evaluation shows that detecting patch linkage using textual content and file location features is promising, with high recall rates. Thus, future work can adopt this model and implement a tool into the practice by studying real users. Moreover, improvement of the detection using alternative approaches with additional feature is also an immediate future work, as there is still room for precision.

(III) Establish a Common Benchmark of Dataset and Metric for CR research. In Chapter 6, results from the mapping study indicates that at this stage, we cannot benchmark CR studies. However, the existing datasets and metrics do show potential for creating a benchmark. For instance, results show that there exists a regular group of metric set combinations commonly used for papers that are addressing a specific SE topic. On the other hand, observations show that researchers from different groups prefer to construct their own datasets to conduct their study. With the rise of machine learning and AI techniques, CR researchers will soon need to agree on the common set of metrics that should be included to accurately compare such techniques against each other. Having a benchmark will facilitate new researchers, including experts from other fields, to innovate new techniques and build on top of already established methodologies.

References

- [1] B A. Kitchenham. Guidelines for performing systematic literature reviews in software engineering. Technical Report EBSE 2007-001, Keele University and Durham University Joint Report, 2007.
- [2] Ulrike Abelein and Barbara Paech. Understanding the influence of user participation and involvement on system success - a systematic mapping study. In Software Engineering 2016, pages 85–86, 2016.
- [3] A. Frank Ackerman, Priscilla J. Fowler, and Robert G. Ebenau. Software inspections and the industrial production of software. In Proc. of a Symposium on Software Validation: Inspection-Testing-Verification-Alternatives, page 13–40, USA, 1984. Elsevier North-Holland, Inc. ISBN 044487593X.
- [4] A. Frank Ackerman, Lynne S. Buchwald, and Frank H. Lewski. Software inspections: An effective verification process. IEEE Softw., 6(3):31–36, May 1989. ISSN 0740-7459. doi: 10.1109/52.28121. URL <https://doi.org/10.1109/52.28121>.
- [5] Muhammad Ahasanuzzaman, Muhammad Asaduzzaman, Chanchal K. Roy, and Kevin A. Schneider. Mining duplicate questions in stack overflow. In Proceedings of the 13th International Conference on Mining Software Repositories, MSR '16, pages 402–412, 2016.
- [6] A. Alami, M. Leavitt Cohn, and A. Wąsowski. Why does code review work for open source software communities? In 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), pages 1073–1083, 2019.
- [7] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. Learning natural coding conventions. In Proceedings of the 22Nd ACM SIGSOFT

International Symposium on Foundations of Software Engineering, FSE 2014, pages 281–293, 2014.

- [8] Android, 2019. URL <http://android-review.googlesource.com>.
- [9] Maurício Aniche, Christoph Treude, Igor Steinmacher, Igor Wiese, Gustavo Pinto, Margaret-Anne Storey, and Marco Aurélio Gerosa. How modern news aggregators help development communities shape and share knowledge. In Proceedings of the 40th International Conference on Software Engineering, page 499–510, 2018.
- [10] F. Armstrong, F. Khomh, and B. Adams. Broadcast vs. unicast review technology: Does it matter? In 2017 IEEE International Conference on Software Testing, Verification and Validation (ICST), pages 219–229, 2017.
- [11] Ikram El Asri, Nouredine Kerzazi, Gias Uddin, Foutse Khomh, and M.A. Janati Idrissi. An empirical study of sentiments in code reviews. Information and Software Technology, pages 37 – 54, 2019.
- [12] Aybüke Aurum, Håkan Petersson, and Claes Wohlin. State-of-the-art: software inspections after 25 years. Softw. Test., Verif. Reliab., 12:133–154, 2002.
- [13] Alberto Bacchelli and Christian Bird. Expectations, outcomes, and challenges of modern code review. In Proceedings of the 2013 International Conference on Software Engineering, ICSE '13, pages 712–721, 2013.
- [14] Deepika Badampudi, Ricardo Britto, and Michael Unterkalmsteiner. Modern code reviews - preliminary results of a systematic mapping study. In Proceedings of the Evaluation and Assessment on Software Engineering, EASE '19, pages 340–345. ACM, 2019.
- [15] Vipin Balachandran. Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation. In Proceedings of the 2013 International Conference on Software Engineering, ICSE '13, pages 931–940, 2013.
- [16] Sebastian Baltes and Stephan Diehl. Usage and attribution of stack overflow code snippets in github projects. Empirical Software Engineering, page 1259–1295, 2019.

- [17] Mike Barnett, Christian Bird, João Brunet, and Shuvendu K. Lahiri. Helping developers help themselves: Automatic decomposition of code review changesets. In Proceedings of the 37th International Conference on Software Engineering - Volume 1, ICSE '15, pages 134–144, 2015.
- [18] T. Baum, K. Schneider, and A. Bacchelli. On the optimal order of reading source code changes for review. In 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME), pages 329–340, 2017.
- [19] Tobias Baum, Olga Liskin, Kai Niklas, and Kurt Schneider. Factors influencing code review processes in industry. In Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, pages 85–96, 2016.
- [20] Tobias Baum, Fabian Kortum, Kurt Schneider, Arthur Brack, and Jens Schauder. Comparing pre-commit reviews and post-commit reviews using process simulation. Journal of Software: Evolution and Process, page e1865, 2017.
- [21] Tobias Baum, Hendrik Leßmann, and Kurt Schneider. The choice of code review process: A survey on the state of the practice. In Michael Felderer, Daniel Méndez Fernández, Burak Turhan, Marcos Kalinowski, Federica Sarro, and Dietmar Winkler, editors, Product-Focused Software Process Improvement, pages 111–127, 2017.
- [22] G. Bavota and B. Russo. Four eyes are better than two: On the impact of code reviews on software quality. In 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME), pages 81–90, 2015.
- [23] O. Baysal, O. Kononenko, R. Holmes, and M. W. Godfrey. The secret life of patches: A firefox case study. In 2012 19th Working Conference on Reverse Engineering, pages 447–455, 2012.
- [24] O. Baysal, O. Kononenko, R. Holmes, and M. W. Godfrey. The influence of non-technical factors on code review. In 2013 20th Working Conference on Reverse Engineering (WCRE), pages 122–131, 2013.
- [25] Olga Baysal, Oleksii Kononenko, Reid Holmes, and Michael W. Godfrey. Investigating technical and non-technical factors influencing modern code review. Empirical Software Engineering, 21:932–959, 2015.

- [26] 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, MSR 2014, pages 202–211, 2014.
- [27] H. Bernard. Research Methods in Anthropology: Qualitative and Quantitative Approaches. Rowman & Littlefield, 2011.
- [28] João Helis Bernardo, Daniel Alencar da Costa, and Uirá Kulesza. Studying the impact of adopting continuous integration on the delivery time of pull requests. In Proceedings of the 15th International Conference on Mining Software Repositories, MSR '18, pages 131–141, 2018.
- [29] N. Bettenburg, R. Premraj, T. Zimmermann, and 3. Sunghun Kim. Duplicate bug reports considered harmful . . . really? In 2008 IEEE International Conference on Software Maintenance, pages 337–345, 2008.
- [30] Christian Bird, Trevor Carnahan, and Michaela Greiler. Lessons learned from building and deploying a code review analytics platform. In Proceedings of the 12th Working Conference on Mining Software Repositories, MSR '15, pages 191–201, 2015.
- [31] V. Boisselle and B. Adams. The impact of cross-distribution bug duplicates, empirical study on debian and ubuntu. In 2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM), pages 131–140, 2015.
- [32] A. Bosu and J. C. Carver. Impact of peer code review on peer impression formation: A survey. In 2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement, pages 133–142, 2013.
- [33] A. Bosu, J. C. Carver, C. Bird, J. Orbeck, and C. Chockley. Process aspects and social dynamics of contemporary code review: Insights from open source development and industrial practice at microsoft. IEEE Transactions on Software Engineering, pages 56–75, 2017.
- [34] Amiangshu Bosu and Jeffrey C. Carver. Impact of developer reputation on code review outcomes in oss projects: An empirical investigation. In Proceedings of

the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM '14, pages 33:1–33:10, 2014.

- [35] 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 ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2014, pages 257–268, 2014.
- [36] Amiangshu Bosu, Michaela Greiler, and Christian Bird. Characteristics of useful code reviews: An empirical study at microsoft. In Proceedings of the 12th Working Conference on Mining Software Repositories, MSR '15, pages 146–156, 2015.
- [37] NORMAN BRESLOW. A generalized Kruskal-Wallis test for comparing K samples subject to unequal patterns of censorship. Biometrika, pages 579–594, 1970.
- [38] L. Brothers, V. Sembugamoorthy, and M. Muller. Icycle: Groupware for code inspection. In Proceedings of the 1990 ACM Conference on Computer-Supported Cooperative Work, page 169–181, 1990.
- [39] Sample Size Calculator, 2020. URL <https://www.surveysystem.com/sscalc.htm>.
- [40] Norman Cliff. Answering Ordinal Questions with Ordinal Data Using Ordinal Statistics. Multivariate Behavioral Research, (3):331–350, 1996.
- [41] Codestriker, 2019. URL <http://codestriker.sourceforge.net>.
- [42] F. Coelho, T. Massoni, and E. L.G. Alves. Refactoring-aware code review: A systematic mapping study. In 2019 IEEE/ACM 3rd International Workshop on Refactoring (IWor), pages 63–66, 2019.
- [43] Manoel Limeira de Lima Júnior, Daricélio Moreira Soares, Alexandre Plastino, and Leonardo Murta. Automatic assignment of integrators to pull requests: The importance of selecting appropriate attributes. Journal of Systems and Software, pages 181 – 196, 2018.
- [44] R. M. de Mello, R. Oliveira, and A. Garcia. On the influence of human factors for identifying code smells: A multi-trial empirical study. In 2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), pages 68–77, 2017.

- [45] M. di Biase, M. Bruntink, and A. Bacchelli. A security perspective on code review: The case of chromium. In 2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM), pages 21–30, 2016.
- [46] F. Ebert, F. Castor, N. Novielli, and A. Serebrenik. Confusion in code reviews: Reasons, impacts, and coping strategies. In 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), pages 49–60, 2019.
- [47] M. E. Fagan. Design and code inspections to reduce errors in program development. IBM Systems Journal, 15(3):182–211, 1976. doi: 10.1147/sj.153.0182.
- [48] Michael Fagan. A History of Software Inspections, pages 562–573. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002. ISBN 978-3-642-59412-0. doi: 10.1007/978-3-642-59412-0_34. URL https://doi.org/10.1007/978-3-642-59412-0_34.
- [49] Yuanrui Fan, Xin Xia, David Lo, and Shanping Li. Early prediction of merged code changes to prioritize reviewing tasks. Empirical Softw. Engg., pages 3346–3393, 2018.
- [50] Benjamin Floyd, Tyler Santander, and Westley Weimer. Decoding the representation of code in the brain: An fmri study of code review and expertise. In Proceedings of the 39th International Conference on Software Engineering, ICSE '17, pages 175–186. IEEE Press, 2017.
- [51] 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, ICSE '18, pages 523–534. ACM, 2018.
- [52] Gerrit, 2019. URL <http://www.gerritcodereview.com>.
- [53] John Gintell, John Arnold, Michael Houde, Jacek Kruszelnicki, Roland McKenney, and Gérard Memmi. Scrutiny: A collaborative inspection and review system. In Ian Sommerville and Manfred Paul, editors, Proceedings of the 4th European Software Engineering Conference, pages 344–360, 1993.

- [54] Carlos Gomez, Brendan Cleary, and Leif Singer. A study of innovation diffusion through link sharing on stack overflow. In IEEE International Working Conference on Mining Software Repositories, pages 81–84, 05 2013.
- [55] Anurag Goswami, Gursimran Walia, and Abhinav Singh. Using learning styles of software professionals to improve their inspection team performance. International Journal of Software Engineering and Knowledge Engineering, pages 1721–1726, 2015.
- [56] 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, ICSE 2014, pages 345–355, 2014.
- [57] Georgios Gousios, Andy Zaidman, Margaret-Anne Storey, and Arie van Deursen. Work practices and challenges in pull-based development: The integrator’s perspective. In Proceedings of the 37th International Conference on Software Engineering - Volume 1, ICSE ’15, pages 358–368. IEEE Press, 2015.
- [58] Georgios Gousios, Margaret-Anne Storey, and Alberto Bacchelli. Work practices and challenges in pull-based development: The contributor’s perspective. In Proceedings of the 38th International Conference on Software Engineering, ICSE ’16, pages 285–296, 2016.
- [59] Monika Gupta, Ashish Sureka, and Srinivas Padmanabhuni. Process mining multiple repositories for software defect resolution from control and organizational perspective. In Proceedings of the 11th Working Conference on Mining Software Repositories, MSR 2014, pages 122–131, 2014.
- [60] Dan Gusfield. Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology. Cambridge University Press, 1997.
- [61] 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, MSR ’13, pages 49–52, 2013.
- [62] Q. Hanam, A. Mesbah, and R. Holmes. Aiding code change understanding with semantic change impact analysis. In 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME), pages 202–212, 2019.

- [63] Christoph Hannebauer, Michael Patalas, Sebastian Stünkel, and Volker Gruhn. Automatically recommending code reviewers based on their expertise: An empirical comparison. In Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, pages 99–110, 2016.
- [64] Frank E. Harrell Jr., Kerry L. Lee, Robert M. Califf, David B. Pryor, and Robert A. Rosati. Regression modelling strategies for improved prognostic prediction. Statistics in Medicine, pages 143–152, 1984.
- [65] T. Hastie, R. Tibshirani, and J.H. Friedman. The Elements of Statistical Learning: Data Mining, Inference, and Prediction. Springer, 2009.
- [66] H. Hata, C. Treude, R. G. Kula, and T. Ishio. 9.6 Million Links in Source Code Comments: Purpose, Evolution, and Decay. In Proceedings of the 41st International Conference on Software Engineering, page 1211–1221, 2019.
- [67] Vincent J. Hellendoorn, Premkumar T. Devanbu, and Alberto Bacchelli. Will they like this?: Evaluating code contributions with language models. In Proceedings of the 12th Working Conference on Mining Software Repositories, MSR '15, pages 157–167, 2015.
- [68] T. Hirao, S. McIntosh, A. Ihara, and K. Matsumoto. Code reviews with divergent review scores: An empirical study of the openstack and qt communities. IEEE Transactions on Software Engineering, pages 1–1, 03 2020.
- [69] Toshiaki 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 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2019, page 578–589, 2019.
- [70] Dorota Huizinga and Adam Kolawa. Automated Defect Prevention: Best Practices in Software Management. John Wiley & Sons, 2007. ISBN 0470042125.
- [71] Jing Jiang, David Lo, Xinyu Ma, Fuli Feng, and Li Zhang. Understanding inactive yet available assignees in github. Inf. Softw. Technol., pages 44–55, 2017.
- [72] Jing Jiang, Yun Yang, Jiahuan He, Xavier Blanc, and Li Zhang. Who should comment on this pull request? analyzing attributes for more accurate commenter

- recommendation in pull-based development. Inf. Softw. Technol., pages 48–62, 2017.
- [73] Jing Jiang, Jin Cao, and Li Zhang. An empirical study of link sharing in review comments. In Zheng Li, He Jiang, Ge Li, Minghui Zhou, and Ming Li, editors, Software Engineering and Methodology for Emerging Domains, pages 101–114, 2019.
- [74] Y. Jiang, B. Adams, and D. M. German. Will my patch make it? and how fast? case study on the linux kernel. In 2013 10th Working Conference on Mining Software Repositories (MSR), pages 101–110, 2013.
- [75] 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, pages 757–773, 2013.
- [76] D. Kavaler, P. Devanbu, and V. Filkov. Whom are you going to call?: Determinants of @-mentions in github discussions. Empirical Software Engineering, page 3904–3932, 2019.
- [77] Barbara Kitchenham. Procedures for performing systematic reviews. Keele, UK, Keele Univ., 33:1–26, 2004.
- [78] Josef Kittler, Mohamad Hatef, Robert P. W. Duin, and Jiri Matas. On combining classifiers. IEEE Trans. Pattern Anal. Mach. Intell., pages 226–239, March 1998. ISSN 0162-8828.
- [79] Sami Kollanus. Icm— a maturity model for software inspections. J. Softw. Maint. Evol., page 327–341, 2011.
- [80] Sami Kollanus and Jussi Koskinen. Survey of software inspection research. The Open Software Engineering Journal, 3:15–34, 05 2009.
- [81] O. Kononenko, O. Baysal, L. Guerrouj, Y. Cao, and M. W. Godfrey. Investigating code review quality: Do people and participation matter? In 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME), pages 111–120, 2015.

- [82] O. Kononenko, O. Baysal, and M. W. Godfrey. Code review quality: How developers see it. In 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE), pages 1028–1038, 2016.
- [83] Oleksii Kononenko, Tresa Rose, Olga Baysal, Michael Godfrey, Dennis Theisen, and Bart de Water. Studying pull request merges: A case study of shopify’s active merchant. In Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice, ICSE-SEIP ’18, pages 124–133. ACM, 2018.
- [84] Vladimir Kovalenko, Nava Tintarev, Evgeny Pasyukov, Christian Bird, and Alberto Bacchelli. Does reviewer recommendation help developers? IEEE Transactions on Software Engineering, pages 710–731, August 2018.
- [85] Robert V. Krejcie and Daryle W. Morgan. Determining sample size for research activities. Educational and Psychological Measurement, 30(3):607–610, 1970.
- [86] Stephan Krusche, Mjellma Berisha, and Bernd Bruegge. Teaching code review management using branch based workflows. In Proceedings of the 38th International Conference on Software Engineering Companion, ICSE ’16, pages 384–393. ACM, 2016.
- [87] F. Lanubile, F. Calefato, and C. Ebert. Group awareness in global software engineering. IEEE Software, (2):18–23, March 2013.
- [88] Zhixing Li, Gang Yin, Yue Yu, Tao Wang, and Huaimin Wang. Detecting duplicate pull-requests in github. In Proceedings of the 9th Asia-Pacific Symposium on Internetware, Internetware’17, pages 20:1–20:6, 2017.
- [89] Zhongxin Liu, Xin Xia, Christoph Treude, David Lo, and Shanping Li. Automatic generation of pull request descriptions. In Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering, ASE ’19, page 176–188, 2019.
- [90] D. Ma, D. Schuler, T. Zimmermann, and J. Sillito. Expert recommendation with usage expertise. In 2009 IEEE International Conference on Software Maintenance, pages 535–538, 2009.
- [91] L. MacLeod, M. Greiler, M. Storey, C. Bird, and J. Czerwonka. Code reviewing in the trenches: Challenges and best practices. IEEE Software, pages 34–42, 2018.

- [92] Chandra Maddila, Chetan Bansal, and Nachiappan Nagappan. Predicting pull request completion time: A case study on large scale cloud services. In Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ES-EC/FSE 2019, page 874–882, 2019.
- [93] I. Malavolta, R. Verdecchia, B. Filipovic, M. Bruntink, and P. Lago. How maintainability issues of android apps evolve. In 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), pages 334–344, Sep. 2018.
- [94] H. B. Mann and D. R. Whitney. On a test of whether one of two random variables is stochastically larger than the other. Annals of Mathematical Statistics, pages 50–60, 1947.
- [95] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. Introduction to Information Retrieval. Cambridge University Press, 2008.
- [96] Vadim Markovtsev, Waren Long, Hugo Mougard, Konstantin Slavnov, and Egor Bulychev. Style-analyzer: Fixing code style inconsistencies with interpretable unsupervised algorithms. In Proceedings of the 16th International Conference on Mining Software Repositories, MSR '19, page 468–478, 2019.
- [97] V. Mashayekhi, J. M. Drake, W. . Tsai, and J. Riedl. Distributed, collaborative software inspection. IEEE Software, pages 66–75, 1993.
- [98] G. Mathew, A. Agrawal, and T. Menzies. Finding trends in software research. IEEE Transactions on Software Engineering, pages 1–1, 2018.
- [99] S. McIntosh and Y. Kamei. Are fix-inducing changes a moving target? a longitudinal case study of just-in-time defect prediction. IEEE Transactions on Software Engineering, pages 412–428, 2018.
- [100] 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, MSR 2014, pages 192–201, 2014.
- [101] 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 Softw. Engg., 21:2146–2189, 2016. ISSN 1382-3256.

- [102] Massimiliano Menarini, Yan Yan, and William G. Griswold. Semantics-assisted code review: An efficient toolchain and a user study. In Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, pages 554–565, 2017.
- [103] Samim Mirhosseini and Chris Parnin. Can automated pull requests encourage software developers to upgrade out-of-date dependencies? In Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, pages 84–94, 2017.
- [104] Ferenc Moksony. Small is beautiful: The use and interpretation of r2 in social research. Szociologiai Szemle, pages 130–138, 01 1999.
- [105] R. Morales, S. McIntosh, and F. Khomh. Do code review practices impact design quality? a case study of the qt, vtk, and itk projects. In 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER), pages 171–180, 2015.
- [106] Murtuza Mukadam, Christian Bird, and Peter C. Rigby. Gerrit software code review data from android. In Proceedings of the 10th Working Conference on Mining Software Repositories, MSR '13, page 45–48, 2013.
- [107] Anh Tuan Nguyen, Tung Thanh Nguyen, Tien N. Nguyen, David Lo, and Chengnian Sun. Duplicate Bug Report Detection with a Combination of Information Retrieval and Topic Modeling. In Proceedings of the 27th International Conference on Automated Software Engineering, pages 70–79, 2012.
- [108] OpenStack, 2019. URL <http://review.openstack.org>.
- [109] A. Ouni, R. G. Kula, and K. Inoue. Search-based peer reviewers recommendation in modern code review. In 2016 IEEE International Conference on Software Maintenance and Evolution (ICSME), pages 367–377, 2016.
- [110] M. Paixao and P. H. Maia. Rebasing in code review considered harmful: A large-scale empirical investigation. In 2019 19th International Working Conference on Source Code Analysis and Manipulation (SCAM), pages 45–55, 2019.
- [111] Matheus Paixao, Jens Krinke, DongGyun Han, Chaiyong Ragkhitwetsagul, and Mark Harman. Are developers aware of the architectural impact of their changes?

In Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, pages 95–105, 2017.

- [112] 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 (SANER), pages 161–170, 2015.
- [113] Luca Pascarella, Davide Spadini, Fabio Palomba, Magiel Bruntink, and Alberto Bacchelli. Information Needs in Contemporary Code Review. Proceedings of the ACM Conference on Computer Supported Cooperative Work, 2:135:1–135:27, 2018.
- [114] Rajshakhar Paul, Amiangshu Bosu, and Kazi Zakia Sultana. Expressions of sentiments during code reviews: Male vs. female. In 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), pages 26–37, 2019.
- [115] J. M. Perpich, D. E. Perry, A. A. Porter, L. G. Votta, and M. W. Wade. Anywhere, anytime code inspections: Using the web to remove inspection bottlenecks in large-scale software development. In Proceedings of the 19th International Conference on Software Engineering, page 14–21, 1997.
- [116] Kai Petersen, Robert Feldt, Shahid Mujtaba, and Michael Mattsson. Systematic mapping studies in software engineering. In Proceedings of the 12th International Conference on Evaluation and Assessment in Software Engineering, EASE’08, pages 68–77, 2008.
- [117] Mohammad Masudur 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 Companion, ICSE ’16, pages 222–231. ACM, 2016.
- [118] 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, MSR ’17, pages 215–226, 2017.

- [119] Achyudh Ram, Anand Ashok Sawant, Marco Castelluccio, and Alberto Bacchelli. What makes a code change easier to review: An empirical investigation on code change reviewability. In Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2018, pages 201–212. ACM, 2018.
- [120] J. Ramos. Using tf-idf to determine word relevance in document queries. In Proceedings of the first Instructional Conference on Machine Learning, pages 133–142, 2003.
- [121] Romesh Ranawana and Vasile Palade. Multi-classifier systems: Review and a roadmap for developers. Int. J. Hybrid Intell. Syst., page 35–61, 2006.
- [122] 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, ICSE '18, page 834–845, 2018.
- [123] L. Ren, S. Zhou, C. Kästner, and A. Wąsowski. Identifying redundancies in fork-based development. In 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), pages 230–241, 2019.
- [124] ReviewBoard, 2019. URL <http://www.reviewboard.org>.
- [125] P. Rigby, B. Cleary, F. Painchaud, M. Storey, and D. German. Contemporary peer review in action: Lessons from open source development. IEEE Software, pages 56–61, 2012.
- [126] Peter C. Rigby and Christian Bird. Convergent contemporary software peer review practices. In Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2013, pages 202–212, 2013.
- [127] 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, ICSE '11, pages 541–550, 2011.
- [128] 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, ICSE '08, pages 541–550, 2008.

- [129] Peter C. Rigby, Daniel M. German, Laura Cowen, and Margaret-Anne Storey. Peer review on open-source software projects: Parameters, statistical models, and theory. ACM Trans. Softw. Eng. Methodol., pages 35:1–35:33, 2014.
- [130] 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, page 973–1016, 2018.
- [131] P. Runeson, M. Alexandersson, and O. Nyholm. Detection of Duplicate Defect Reports Using Natural Language Processing. In Proceedings of the 29th International Conference on Software Engineering, pages 499–510, 2007.
- [132] Caitlin Sadowski, Emma Söderberg, Luke Church, Michal Sipko, and Alberto Bacchelli. Modern code review: A case study at google. In Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice, ICSE-SEIP '18, pages 181–190. ACM, 2018.
- [133] Shipra Sharma and Balwinder Sodhi. Using stack overflow content to assist in code review. Software: Practice and Experience, pages 1255–1277, 2019.
- [134] J. Shimagaki, Y. Kamei, S. McIntosh, D. Pursehouse, and N. Ubayashi. Why are commits being reverted?: A comparative study of industrial and open source projects. In 2016 IEEE International Conference on Software Maintenance and Evolution (ICSME), pages 301–311, 2016.
- [135] 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 Proceedings of the 38th International Conference on Software Engineering Companion, ICSE '16, pages 212–221. ACM, 2016.
- [136] Forrest Shull, Vic Basili, Barry Boehm, A. Winsor Brown, Patricia Costa, Mikael Lindvall, Dan Port, Ioana Rus, Roseanne Tesoriero, and Marvin Zelkowitz. What we have learned about fighting defects. METRICS '02, page 249, USA, 2002. IEEE Computer Society. ISBN 0769513395.
- [137] Forrest Shull, Raimund L. Feldmann, Carolyn B. Seaman, Myrna Regardie, and Sally Godfrey. Fully employing software inspections data. Innovations in Systems and Software Engineering, pages 243–254, 2010.

- [138] Daricélio M. Soares, Manoel L. de Lima Júnior, Alexandre Plastino, and Leonardo Murta. What factors influence the reviewer assignment to pull requests? Information and Software Technology, 98:32 – 43, 2018.
- [139] Behjat Soltanifar, Atakan Erdem, and Ayse Bener. Predicting defectiveness of software patches. In Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM '16, pages 22:1–22:10, 2016.
- [140] Davide Spadini, Maurício Aniche, Margaret-Anne Storey, Magiel Bruntink, and Alberto Bacchelli. When testing meets code review: Why and how developers review tests. In Proceedings of the 40th International Conference on Software Engineering, ICSE '18, pages 677–687. ACM, 2018.
- [141] Davide Spadini, Fabio Palomba, Tobias Baum, Stefan Hanenberg, Magiel Bruntink, and Alberto Bacchelli. Test-driven code review: An empirical study. In Proceedings of the 41st International Conference on Software Engineering, ICSE '19, page 1061–1072, 2019.
- [142] C. Sun, D. Lo, S. Khoo, and J. Jiang. Towards more accurate retrieval of duplicate bug reports. In 2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), pages 253–262, Nov 2011.
- [143] Yida Tao and Sunghun Kim. Partitioning composite code changes to facilitate code review. In Proceedings of the 12th Working Conference on Mining Software Repositories, MSR '15, pages 180–190, 2015.
- [144] Yida Tao, Yingnong Dang, Tao Xie, Dongmei Zhang, and Sunghun Kim. How do software engineers understand code changes?: An exploratory study in industry. In Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, FSE '12, pages 51:1–51:11, 2012.
- [145] Yida Tao, Donggyun Han, and Sunghun Kim. Writing acceptable patches: An empirical study of open source project patches. In Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution, ICSME '14, pages 271–280, 2014.
- [146] P. Thongtanunam and A. E. Hassan. Review dynamics and their impact on software quality. IEEE Transactions on Software Engineering, pages 1–1, 2020.

- [147] P. Thongtanunam, C. Tantithamthavorn, R. G. Kula, N. Yoshida, H. Iida, and K. Matsumoto. Who should review my code? a file location-based code-reviewer recommendation approach for modern code review. In 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER), pages 141–150, 2015.
- [148] 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, MSR '15, pages 168–179, 2015.
- [149] 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, ICSE '16, pages 1039–1050. ACM, 2016.
- [150] Patanamon Thongtanunam, Shane Mcintosh, Ahmed E. Hassan, and Hajimu Iida. Review participation in modern code review. Empirical Softw. Engg., pages 768–817, 2017.
- [151] Ferdian Thung, Pavneet Singh Kochhar, and David Lo. Dupfinder: Integrated tool support for duplicate bug report detection. In Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering, ASE '14, pages 871–874, 2014.
- [152] P. Tourani and B. Adams. The impact of human discussions on just-in-time quality assurance: An empirical study on openstack and eclipse. In 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), pages 189–200, 2016.
- [153] 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, ICSE 2014, pages 356–366, 2014.
- [154] Jason Tsay, Laura Dabbish, and James Herbsleb. Let’s talk about it: Evaluating contributions through discussion in github. In Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2014, pages 144–154, 2014.

- [155] Y. Tymchuk, A. Mocci, and M. Lanza. Code review: Veni, vidi, vici. In 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER), pages 151–160, 2015.
- [156] Anthony J Viera, Joanne M Garrett, et al. Understanding Interobserver Agreement: The Kappa Statistic. Family Medicine, 37(5):360–363, 2005.
- [157] Robert J. Walker, Shreya Rawal, and Jonathan Sillito. Do crosscutting concerns cause modularity problems? In Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, FSE '12, pages 49:1–49:11, 2012.
- [158] Dong Wang, Tao Xiao, Patanamon Thongtanunam, Raula Gaikovina Kula, and Kenichi Matsumoto. Understanding shared links and their intentions to meet information needs in modern code review. Empir. Softw. Eng., 26(5): 96, 2021. doi: 10.1007/s10664-021-09997-x. URL <https://doi.org/10.1007/s10664-021-09997-x>.
- [159] Jing Wang, Patrick C. Shih, Yu Wu, and John M. Carroll. Comparative case studies of open source software peer review practices. Inf. Softw. Technol., 67: 1–12, 2015.
- [160] M. Wang, Z. Lin, Y. Zou, and B. Xie. Cora: Decomposing and describing tangled code changes for reviewer. In 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), pages 1050–1061, 2019.
- [161] Qingye Wang, Xin Xia, David Lo, and Shanping Li. Why is my code change abandoned? Information and Software Technology, pages 108 – 120, 2019.
- [162] Qingye Wang, Bowen Xu, Xin Xia, Ting Wang, and Shanping Li. Duplicate pull request detection: When time matters. In Proceedings of the 11th Asia-Pacific Symposium on Internetware, Internetware '19, pages 1–10, 2019.
- [163] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun. An approach to detecting duplicate bug reports using natural language and execution information. In 2008 ACM/IEEE 30th International Conference on Software Engineering, pages 461–470, 2008.
- [164] Ruiyin Wen, David Gilbert, Michael G. Roche, and Shane McIntosh. BLIMP Tracer: Integrating Build Impact Analysis with Code Review. In Proc. of the

International Conference on Software Maintenance and Evolution (ICSME), page 685–694, 2018.

- [165] Roel Wieringa, Neil Maiden, Nancy Mead, and Colette Rolland. Requirements engineering paper classification and evaluation criteria: A proposal and a discussion. Requir. Eng., 11(1):102–107, December 2005.
- [166] Peter Willett. The porter stemming algorithm: then and now. Program, pages 219–223, 2006.
- [167] 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 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME), pages 261–270, 09 2015.
- [168] Yu xia Zhang, Minghui Zhou, Klaas-Jan Stol, Jian yu Wu, and Z. Jin. How do companies collaborate in open source ecosystems? an empirical study of open-stack. 2020 IEEE/ACM 42nd International Conference on Software Engineering, page 1196–1208, 2020.
- [169] X. Yang, R. G. Kula, N. Yoshida, and H. Iida. Mining the modern code review repositories: A dataset of people, process and product. In 2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR), pages 460–463, 2016.
- [170] Deheng Ye, Zhenchang Xing, and Nachiket Kapre. The structure and dynamics of knowledge network in domain-specific q&a sites: A case study of stack overflow. Empirical Softw. Engg., page 375–406, 2017.
- [171] Y. Yu, Z. Li, G. Yin, T. Wang, and H. Wang. A dataset of duplicate pull-requests in github. In 2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR), pages 22–25, May 2018.
- [172] Yue Yu, Huaimin Wang, Gang Yin, and Tao Wang. Reviewer recommendation for pull-requests in github. Inf. Softw. Technol., 74:204–218, 2016.
- [173] F. Zampetti, L. Ponzanelli, G. Bavota, A. Mocci, M. Di Penta, and M. Lanza. How developers document pull requests with external references. In 2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC), pages 23–33, 2017.

- [174] F. Zampetti, G. Bavota, G. Canfora, and M. D. Penta. A study on the interplay between pull request review and continuous integration builds. In 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), pages 38–48, 2019.
- [175] 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, ESEM '18, pages 11:1–11:10, 2018.
- [176] Motahareh Bahrami Zanjani, Huzefa Kagdi, and Christian Bird. Automatically recommending peer reviewers in modern code review. IEEE Trans. Softw. Eng., pages 530–543, 2016.
- [177] Tianyi Zhang, Myoungkyu Song, Joseph Pinedo, and Miryung Kim. Interactive code review for systematic changes. In Proceedings of the 37th International Conference on Software Engineering - Volume 1, ICSE '15, pages 111–122. IEEE Press, 2015.
- [178] X. Zhang, Y. Chen, Y. Gu, W. Zou, X. Xie, X. Jia, and J. Xuan. How do multiple pull requests change the same code: A study of competing pull requests in github. In 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), pages 228–239, 2018.
- [179] Yang Zhang, Yue Yu, Huaimin Wang, Bogdan Vasilescu, and Vladimir Filkov. Within-ecosystem issue linking: A large-scale study of rails. In Proceedings of the 7th International Workshop on Software Mining, SoftwareMining 2018, page 12–19, 2018.
- [180] Yang Zhang, Yiwen Wu, Tao Wang, and Huaimin Wang. ilinker: a novel approach for issue knowledge acquisition in github projects. World Wide Web, 23:1589–1619, 2020.
- [181] Yuxia Zhang, Minghui Zhou, Audris Mockus, and Zhi Jin. Companies’ participation in oss development - an empirical study of openstack. IEEE Transactions on Software Engineering, pages 1–1, 2019.
- [182] Guoliang Zhao, Daniel Alencar Costa, and Ying Zou. Improving the pull requests review process using learning-to-rank algorithms. Empirical Softw. Engg., page 2140–2170, 2019.

- [183] Jiaxin Zhu, Minghui Zhou, and Audris Mockus. Effectiveness of code contribution: From patch-based to pull-request-based tools. In Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, pages 871–882, 2016.
- [184] Weiqin Zou, Jifeng Xuan, Xiaoyuan Xie, Zhenyu Chen, and Baowen Xu. How does code style inconsistency affect pull request integration? an exploratory study on 117 github projects. Empirical Software Engineering, page 3871–3903, 2019.