

NAIST-IS-DD1461020

Doctoral Dissertation

**Studying Reviewer Selection and Involvement in
Modern Code Review Processes**

Patanamon Thongtanunam

September 5, 2016

Graduate School of Information Science
Nara Institute of Science and Technology

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

Patanamon Thongtanunam

Thesis Committee:

Professor Hajimu Iida	(Nara Institute of Science and Technology)
Processor Kenichi Matsumoto	(Nara Institute of Science and Technology)
Professor Ahmed E. Hassan	(Queen's University)
Professor Margaret-Anne Storey	(University of Victoria)
Associate Professor Kohei Ichikawa	(Nara Institute of Science and Technology)
Assistant Professor Eunjong Choi	(Nara Institute of Science and Technology)

Studying Reviewer Selection and Involvement in Modern Code Review Processes*

Patanamon Thongtanunam

Abstract

Software code review is a well-established software quality practice. During code review, team members examine each others' code changes in order to identify potential problems early in the development cycle. In a distributed development setting, code review in modern software organizations tends to converge on a lightweight variant called Modern Code Review (MCR). While enabling distributed reviews, MCR lacks mechanisms for ensuring a base level of review quality, which the formal software inspection of the past achieved through the careful review preparation and the formal review execution. Lax reviewing practices may creep into MCR processes, which can impact software quality.

In this thesis, we perform a series of empirical studies in order to better understand how teams can improve their reviewing practices in MCR processes. We first perform two empirical studies to investigate how reviewers should be selected. We observe that (1) the proportion of reviewers without an expertise of a module shares a strong, increasing relationship with the likelihood of that module having future defects; however, (2) it is often difficult selecting appropriate reviewers, which can slow down MCR processes. To address this problem, we propose REVFINDER, a file location-based reviewer recommendation approach.

*Doctoral Dissertation, Graduate School of Information Science,
Nara Institute of Science and Technology, NAIST-IS-DD1461020, September 5, 2016.

Our empirical evaluation shows that REVFINDER accurately recommends (within top 10 recommendation) an appropriate reviewer for 69%-86% of reviews.

We then perform another two empirical studies to investigate reviewer involvement in MCR process. We observe that (1) modules that will eventually have future defects or have been historically defective tend to be reviewed with less reviewer involvement than their clean counterparts and (2) past reviewer involvement tendencies are strong indicators of poor reviewer involvement.

Our results and empirical observations highlight the need for improved reviewing policies in an MCR context in order to mitigate the risk of having defects in software products.

Keywords:

Code Review, Software Quality, Empirical Studies

Related Publications

Early versions of the work in this thesis were published as listed below.

- **Revisiting Code Ownership and its Relationship with Software Quality in the Scope of Modern Code Review.** (Chapter 4)

Patanamon Thongtanunam, Shane McIntosh, Ahmed E. Hassan, and Hajimu Iida. In Proceedings of the 38th International Conference on Software Engineering (ICSE), 2016, pp. 1039–1050.

- **Who Should Review My Code?** (Chapter 5)

Patanamon Thongtanunam, Chakkrit Tantithamthavorn, Raula Gaikovina Kula, Norihiro Yoshida, Hajimu Iida, Kenichi Matsumoto. In Proceedings of the 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER), 2015, pp. 141–150.

- **Investigating Code Review Practices in Defective Files.** (Chapter 6)

Patanamon Thongtanunam, Shane McIntosh, Ahmed E. Hassan, and Hajimu Iida. In Proceedings of the 12th International Conference on Mining Software Repositories (MSR), 2015, pp. 168–179.

★*This paper won the IEEE Kansai Excellent Student Paper Award 2015*★

- **Review Participation in Modern Code Review.** (Chapter 7)

Patanamon Thongtanunam, Shane McIntosh, Ahmed E. Hassan, and Hajimu Iida. Springer Journal of Empirical Software Engineering (EMSE), 2016, 1–47.

The following publications are not directly related to the material in this thesis, but were produced in parallel to the research performed for this thesis.

- **ReDA: A Web-based Visualization Tool for Analyzing Modern Code Review Datasets.**

Patanamon Thongtanunam, Xin Yang, Norihiro Yoshida, Kenji Fujiwara, Raula Gaikovina Kula, Ana Erika Camargo Cruz, Hajimu Iida. In Proceedings of the 30th International Conference on Software Maintenance and Evolution (ICSME), 2014, pp. 605–608.

- **Assessing MCR Discussion Usefulness using Semantic Similarity.**

Thai Pangsakulyanont, Patanamon Thongtanunam, Daniel Port, Hajimu Iida. In Proceedings of the 6th International Workshop on Empirical Software Engineering in Practice (IWESEP), 2014, pp. 49–54.

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 express my sincere gratitude to my supervisor, Professor Hajimu Iida for his always support, guidance for my research as well as my life in Japan. Without his support, I would not successfully accomplish the doctoral degree and survive in Japan.

I also would like to express a deep appreciation to Professors Ahmed E. Hassan and Shane McIntosh for their wisdom and patience. They are very kind and insightful mentors. Not only for their research advices but they also took a serious attention to my work, provided constructive feedback and unbounded instructions.

A special thank to Professor Margaret-Anne Storey. I felt very privileged to have such an accomplished researcher as an external examiner of my thesis. Her feedback was instrumental in forming the complete thesis.

I would also like to thank my thesis committee members: Professors Kenichi Matsumoto, Kohei Ichikawa, and Eunjong Choi. Through expert critique and suggestions, they helped shape this thesis.

Furthermore, I would like to extend thanks to Professor Daniel M. German for taking the time to critique my work and providing valuable suggestions. I feel lucky to work with an amazing network of collaborators: Professors Daniel

Port, Raula Gaikovina Kula, Norihiro Yoshida, Dr. Kenji Fujiwara, Dr. Ana Erika Camargo Cruz, Xin Yang, Thai Pangsakulyanont for sharing wisdom and research perspectives. I also want to thank members of Software Design and Analysis Laboratory (SDLAB) and Software Analysis and Intelligence Laboratory (SAIL) who share research ideas and research experience.

Thanks to generous financial support from Japan Society for the Promotion of Science (JSPS), NEC C&C Foundation, and NAIST Global Initiatives Program.

Without the support of my family and friends, this thesis would not have been possible. I would like to thank my beloved parents and sister who were the main driver that kept me going. Finally, I always eternally indebted to Chakkrit Tantithamthavorn, for his always support and being by my side as we what through this wonderful journey together.

Dedication

To my parents.

Table of Contents

Related Publication	iii
Abstract	iii
Acknowledgements	v
Dedication	vii
List of Figures	xiii
List of Tables	xvii
1 Introduction	1
1.1 Problem Statement	3
1.2 Thesis Overview	4
1.3 Thesis Contribution	7
1.4 Thesis Organization	8
I The Modern Code Review Processes	9
2 The Code Review Process	11
2.1 Traditional Code Review	11
2.2 Modern Code Review	13
2.3 An Overview of the Modern Code Review Process	15

2.4	Chapter Summary	20
3	Related Research	21
3.1	Review Preparation	21
3.2	Reviewer Involvement	25
3.3	Chapter Summary	28
II	Reviewer Selection in Modern Code Review Processes	29
4	The Impact of Reviewer Selection on Software Quality	31
4.1	Introduction	32
4.2	Background & Definition	35
4.3	Case Study Design	39
4.4	Case Study Results	43
4.5	Practical Suggestions	62
4.6	Threats to Validity	63
4.7	Summary	66
5	Selecting Appropriate Reviewers	69
5.1	Introduction	70
5.2	Case Study Design	73
5.3	An Exploratory Study of the Reviewer Selection Problem in MCR processes	75
5.4	RevFinder: A File Location-Based Reviewer Recommendation Ap- proach	80
5.5	Empirical Evaluation	89
5.6	Evaluation Results	92
5.7	Discussion	95
5.8	Threats to Validity	97
5.9	Summary	98

III Reviewer Involvement in Modern Code Review Processes101

6	The Impact of Reviewer Involvement on Software Quality	103
6.1	Introduction	104
6.2	Case Study Design	106
6.3	Case Study Results	118
6.4	Discussion	132
6.5	Threats to Validity	134
6.6	Summary	135
7	Identifying Characteristics of Patches with Poor Reviewer Involvement139	
7.1	Introduction	140
7.2	Case Study Design	143
7.3	Case Study Results	157
7.4	Discussion	187
7.5	Threats to Validity	193
7.6	Summary	195

IV Conclusion and Future Work 197

8	Conclusion	199
8.1	Contribution and Findings	199
8.2	Opportunities for Future Research	202

References 207

Appendix A An Example R Script for Non-linear Logistic Regression

Models	227
A.1	Install and Load a Necessary R Library 227
A.2	Model Construction 227

A.3 Model Analysis	230
Appendix B Additional Results for Chapter 4	235
B.1 A Sensitivity Analysis of RSO thresholds	235
B.2 Additional Model Statistics	237
Appendix C Core Developers of the Studied Projects	243
C.1 The OpenStack Project	243
C.2 The Qt Project	245
Appendix D Additional Analysis for Chapter 7	249
D.1 An Analysis of Threshold Sensitivity	249

List of Figures

1.1	An overview of the scope of the thesis.	5
2.1	An overview of the MCR process.	16
2.2	An example of a review request in the MCR tool (Gerrit) of the Qt project.	18
2.3	An example of file-by-file difference viewing and inline commenting of Qt review ID #101397.	19
4.1	An overview of data preparation approach.	42
4.2	Number of developers who contribute to a module (i.e., authoring vs reviewing code changes).	45
4.3	An overview of our model construction and analysis approaches. .	49
4.4	The distribution of developers of defective (blue) and clean (gray) modules when using RSO_{Even} . The horizontal lines indicates the median of distributions.	54
4.5	Hierarchical clustering of variables according to Spearman's $ \rho $ in the Qt 5.0 dataset. The dashed line indicates the high correlation threshold (i.e., Spearman's $ \rho = 0.7$).	57
4.6	Dotplot of the Spearman multiple ρ^2 of each explanatory variable and defect-proneness in the Qt 5.0 dataset.	58
4.7	The estimated probability in a typical module for the proportion of developers in the minor author & minor reviewer category ranging. The gray area shows the 95% confidence interval.	61

5.1	An overview of our approach for RQ1.	75
5.2	An example of review discussion threads of patches that are identified as having a reviewer selection problem.	77
5.3	A comparison of the time that is spent on MCR processes (days) of reviews with and without the reviewer selection problem. The horizontal lines indicate the average (median) review time (days).	81
5.4	A calculation example of the Reviewer Ranking Algorithm.	82
5.5	An overview of the combination approach of using four string comparison techniques in REVFinder.	83
5.6	The rank distribution of the first correct reviewer that is recommended by REVFinder and REVIEWBOT.	94
6.1	An overview of data preparation and data analysis approaches.	108
6.2	Our approach to identify defective files.	110
6.3	Classification schema for changes that are addressed during the code review process [70]. We add the traceability type.	117
6.4	Distribution of change types that occurred during the code review of future-defective and clean files. The sum of review proportion is higher than 100%, since a review can contain many types of changes.	123
6.5	Distribution of change types that occurred during the code review of risky and normal files. The sum of review proportion is higher than 100%, since a review can contain many types of changes.	128
6.6	Distribution of change types that occurred during the code review of risky & future-defective and risky & clean files. The sum of review proportion is higher than 100%, since a review can contain many types of changes.	131
7.1	An overview of our data preparation approach.	144
7.2	An overview of our model construction and analysis approaches.	154

7.3	Hierarchical clustering of variables according to Spearman's $ \rho $ in the Android dataset (RQ1). The dashed line indicates the high correlation threshold (i.e., Spearman's $ \rho = 0.7$).	161
7.4	Dotplot of the Spearman multiple ρ^2 of each explanatory variable and the response (the likelihood that a patch will not attract reviewers) in the Android dataset. Larger values indicate a higher potential for a nonlinear relationship (RQ1).	162
7.5	The nonlinear relationship between the likelihood that a patch will not attract reviewers (y-axis) and the explanatory variables (x-axis). The larger the odds value is, the higher the likelihood that the patch will not attract reviewers. The gray area shows the 95% confidence interval estimated by using a bootstrap-derived approach.	167
7.6	Hierarchical clustering of variables according to Spearman's $ \rho $ in the Android dataset (RQ2). The dashed line indicates the high correlation threshold (i.e., Spearman's $ \rho = 0.7$).	171
7.7	Dotplot of the Spearman multiple ρ^2 of each explanatory variable and the response (the likelihood that a patch will not be discussed) in the Android dataset. Larger values indicate a higher potential for a nonlinear relationship (RQ2).	173
7.8	The nonlinear relationship between the likelihood that a patch will not be discussed (y-axis) and the explanatory variables (x-axis). The larger the odds value is, the higher the likelihood that the patch will not be not discussed. The gray area shows the 95% confidence interval estimated by using a bootstrap-derived approach.	175
7.9	An example of a calculation for feedback delay.	178
7.10	The hourly code review activity of the Qt system. The dark areas indicate the periods that are likely to be weekends.	180

7.11	Dotplot of the Spearman multiple ρ^2 of each explanatory variable and the response (the likelihood that a patch will receive slow initial feedback) in the Android model. Larger values indicate a higher potential for a nonlinear relationship (RQ3).	181
7.12	The nonlinear relationship between the likelihood that a patch will receive slow initial feedback (y-axis) and the explanatory variables (x-axis). The larger the odds value is, the higher the likelihood that the patch will receive slow initial feedback. The gray area shows the 95% confidence interval estimated by using a bootstrap-derived approach.	184
B.1	A magnitude of difference between the proportion of developers in each of expertise categories of defective and clean modules when using RSO_{Even} . Each dot in each expertise category represents the result of one studied release.	236
B.2	A magnitude of difference between the proportion of developers in each of expertise categories of defective and clean modules when using $RSO_{Proportional}$. Each dot in each expertise category represents the result of one studied release.	237
B.3	The estimated probability in a typical module for the proportion of developers in the minor author & major reviewer category ranging. The gray area shows the 95% confidence interval.	241
D.1	The percentages of patches that receive prompt (blue) and slow (gray) initial feedback. The x-axis shows the patch submission dates.	250

List of Tables

4.1	A contingency table of the review-aware ownership expertise category.	38
4.2	Overview of the studied systems. Systems above the double line satisfy our criteria for further analysis.	40
4.3	A taxonomy of the considered control (top) and code ownership metrics (bottom).	48
4.4	Results of one-tailed Mann-Whitney U tests for the developers in defective (D) and clean (C) models.	55
4.5	Statistics of defect models where review-specific and review-aware ownership are estimated using RSO_{Even} . The explanatory power (χ^2) of each variable is shown in a proportion to Wald χ^2 of the model.	59
5.1	A summary of the dataset for each studied system.	74
5.2	The numbers of statistically representative samples for each studied systems and the proportion of patches with the reviewer selection problem with a 95% confidence level and a $\pm 5\%$ bound. . . .	79
5.3	A description and examples of calculation for file path comparison techniques. The examples are obtained from the review history of Android for the LCP, LCSustr, and LCSubseq techniques; and Qt for the LCS techniques. For each technique, the example files were reviewed by the same reviewer.	88

5.4	The results of Mean Reciprocal Rank (MRR) of REVFinder and a baseline approach. A MRR value of 1 indicates a perfect ranking performance of the approach.	93
5.5	The results of top- k accuracy of our approach RevFinder and a baseline ReviewBot for each studied system. The results show that RevFinder outperforms ReviewBot.	93
6.1	An overview of the studied Qt system.	107
6.2	A taxonomy of the code review activity metrics. The metrics normalized by patch size are marked with a dagger symbol ($\dagger$). . . .	113
6.3	A contingency table of a code review activity metric (m), where a and c represent the number of reviews of defective files, and b and d represent the number of reviews of their clean counterparts. . .	115
6.4	An overview of the review database of RQ1.	119
6.5	Results of one-tailed Mann-Whitney U tests ($\alpha = 0.05$) for code review activity metrics of future-defective and clean files.	120
6.6	An overview of the review database to address RQ2.	125
6.7	Results of one-tailed Mann-Whitney U tests ($\alpha = 0.05$) for code review activity metrics of risky and normal files.	126
6.8	Results of one-tailed Mann-Whitney U tests ($\alpha = 0.05$) for code review activity metrics of risky & future-defective files and risky & clean files.	130
7.1	Overview of the studied systems.	144
7.2	A taxonomy of patch metrics.	147
7.3	Descriptive statistics of the studied patch metrics. Histograms are in a log scale.	159
7.4	Patch data for the study of RQ1.	160

7.5	Statistics of the logistic regression models for identifying patches that do not attract reviewers (RQ1). The explanatory variables that contribute the most significant explanatory power to a model (i.e., accounting for a large proportion of Wald χ^2) are shown in boldface.	163
7.6	Partial effect that our explanatory variables have on the likelihood that a patch will not attract reviewers (RQ1). The larger the magnitude of the odds ratio is, the larger the partial effect that an explanatory variable has on the likelihood that a patch will not attract reviewers.	168
7.7	Patch data for the study of RQ2.	169
7.8	Statistics of the logistic regression models for identifying patches that are not discussed (RQ2). The explanatory variables that contribute the most significant explanatory power to a model (i.e., accounting for a large proportion of Wald χ^2) are shown in boldface.	172
7.9	Partial effect that our explanatory variables have on the likelihood that a patch will not be discussed (RQ2). The larger the magnitude of the odds ratio is, the larger the partial effect that an explanatory variable has on the likelihood that a patch will not be discussed. .	176
7.10	Descriptive statistics of feedback delay (hours).	178
7.11	Patch data for the study of RQ3.	179
7.12	Statistics of the logistic regression models for identifying patches receiving slow initial feedback (RQ3). The explanatory variables that contribute the most significant explanatory power to a model (i.e., accounting for a large proportion of Wald χ^2) are shown in boldface.	182

7.13	Partial effect that our explanatory variables have on the likelihood that a patch will receive slow initial feedback (RQ3). The larger the magnitude of the odds ratio is, the larger the partial effect that an explanatory variable has on the likelihood that a patch will receive slow initial feedback.	185
B.1	Statistics of defect models where review-specific and review-aware ownership are estimated using $RSO_{\text{Proportional}}$	238
B.2	Statistics of defect models where the proportion of core developers is considered and review-specific and review-aware ownership are estimated using RSO_{Even}	239
B.3	Statistics of defect models where review-specific and review-aware ownership are estimated using RSO_{Even} and using the proportion of minor author & major reviewer instead of the proportion of minor author & minor reviewer.	240
C.1	A list of core developers of the OpenStack project.	243
C.2	A list of core developers of the Qt project.	245

Introduction

Code review is a well-established software quality practice. During code review, team members examine each others' changes to the source code. Boehm and Basili argue that code review is one of the best investments for defect reduction [22]. Moreover, Shull *et al.* find that code reviews often catch more than half of a product's detected defects [111]. Baker also reports that when performing code reviews during the development cycle, the number of serious problems found during testing phases decreases by 40% [8].

Formal software inspection, i.e., a rigidly structured code review process, has long been perceived as an effective method to improve the quality of a software product [2, 5, 34, 104]. In the formal software inspection process, time is explicitly allocated for the preparation and execution of in-person meetings. Documents like defect checklists are prepared beforehand. During the meetings, the roles of participants are well defined. For instance, the main focus of reviewers is to identify defects, while the task of the moderator is to keep the inspection meeting on point, and the role of the scribe is to document the defects that were found by reviewers.

For the globally-distributed teams of the modern software organizations, the formal software inspection process is hard to adopt due to the physical constraints of inspection meetings [77, 130]. Within distributed development teams,

collaboration between developers is asynchronous, which requires tools to support coordination and communication [71, 98].

Recent work finds that modern software organizations (e.g., Microsoft, Apache) tend to converge on a lightweight variant of the code review process called Modern Code Review (MCR) [97]. MCR is coordinated using light weight collaborative tools and processes. Broadly speaking, the process of MCR begins with a developer who first (1) uploads a proposed patch (i.e., a set of code changes) to an MCR tool and (2) selects reviewers (i.e., other members of the development team). Next, reviewers (3) critique the patch to identify potential problems. After receiving reviewer feedback, the patch author revises the patch to address the identified problems. Once the reviewer(s) agree that the patch is of sufficient quality, it can be (4) integrated into the project's Version Control System (VCS).

One of the main goals of MCR is to control the quality of patches. Bacchelli and Bird report that the top motivation for using MCR at Microsoft is to remove defects and improve code changes [6]. Baysal *et al.* find that the proposed patches of the Mozilla project tend to be revised several times before integration [13]. Tao *et al.* also uncover several concerns (e.g., compilation errors, un descriptive documentation) that reviewers in the Eclipse and Mozilla projects have pointed out in rejected patches [118].

Nevertheless, MCR is not solely focused on whether or not a patch should be integrated. Instead, MCR tends to focus on collaboration between a patch author and reviewers. For example, recent research finds that reviewers often help patch authors improve proposed patches by suggesting alternative solutions during code reviews at Microsoft [97], on GitHub [128], and within several open source projects [16, 97]. Moreover, from time to time, reviewers even help patch authors to fix problems that were found during reviews [100, 121].

1.1. Problem Statement

MCR does not impose review preparation and execution criteria. The flexibility of MCR is both a blessing and a curse. While enabling distributed code reviews, MCR lacks mechanisms for ensuring a base level of review quality:

Thesis Statement: The lightweight Modern Code Review process lacks mechanisms for preventing lax code reviewing practices, which can allow poor quality code to slip through to official software releases. A deeper understanding of these processes and their impact on software quality is needed as MCR continues to gain popularity and be adopted by projects worldwide.

Indeed, unlike the formal software inspections of the past, MCR processes do not mandate in-person meetings or the use of review checklists. Lax reviewing practices may creep into MCR processes, which can impact software quality [73, 74, 81]. Suboptimal reviewing practices may take several forms:

Reviewer selection – To begin the MCR process, a patch author needs to assign team members to examine the proposed patch. Intuitively, a patch author should assign developers who have related knowledge in the area of the system that is impacted by the patch. However, estimating expertise of developers is not a trivial task [3, 37, 38, 79]. Moreover, reviewer selection criteria are not mandated by the MCR process [101, 121]. Hence, MCR reviews may be suboptimal if reviewers are assigned to critique patches that modify areas of the system for which they have little knowledge.

Reviewer involvement – Due to the human-intensive nature of code reviewing, reviewer involvement plays a key role in code review effectiveness. For example, Linus’ law suggests that given enough reviewers, the defects that escape to the field will be shallow [96]. Fagan argues that a code review checklist can guide reviewers during code examination [33]. However, MCR

processes tend to use an *ad hoc* reading style, where reviewers critique a patch from their personal perspectives [100]. Hence, MCR reviews may not foster a sufficient amount of reviewer involvement to mitigate the risk of having defects in the patch.

The overarching goal of this thesis is to provide actionable insights and MCR process support. To achieve this goal, we perform a series of empirical studies to understand how teams can better prepare for and execute code reviews in an MCR context.

1.2. Thesis Overview

In this section, we provide a brief overview of the thesis. Figure 1.1 outlines the scope of this thesis. We first provide the necessary background material (pink box):

Part I: The Modern Code Review Processes

Chapter 2: *The Code Review Process*

In this chapter, we provide a broad background of code review processes and definitions of the steps of the MCR processes.

Chapter 3: *Related Research*

We present a survey and an in-depth discussion of prior research that is related to this thesis.

Next, we present the main body of the thesis, which consists of four chapters. These chapters are organized into review preparation and execution parts (blue box). Each part consists of two empirical studies (yellow box) that have compelling potential outcomes (green box).

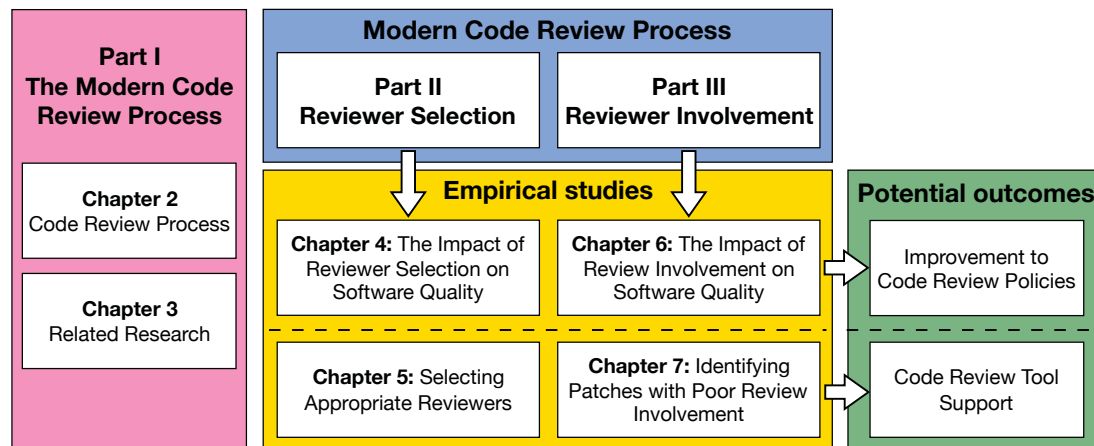


Figure 1.1. An overview of the scope of the thesis.

Part II: Reviewer Selection in Modern Code Review Processes

We perform two empirical studies to address the reviewer selection problem:

Chapter 4: *The Impact of Reviewer Selection on Software Quality*

Developer expertise is important in the software development cycle. Several studies have shown that code authoring expertise shares a strong relationship with software quality [21, 28, 42, 85]. Besides the knowledge that is gained from authoring experience, developers also can gain knowledge of modules by reviewing a patch that modifies those modules [6, 97]. Yet, little is known about whether assigning developers with reviewing expertise to examine a patch that modifies a module can reduce the risk of having defects in that module. Hence, we set out to study the relationship between reviewing expertise and the likelihood of having post-release defects in a module.

Chapter 5: *Selecting Appropriate Reviewers*

To help select appropriate reviewers, we propose REVFINDER, a file location-based reviewer recommendation approach. We leverage the similarities between the path of a newly changed file and the paths of

previously reviewed files to recommend an appropriate reviewer. The intuition behind REVFinder is that files that are located in similar file paths (i.e., similar modules) would be managed and reviewed by reviewers with similar experience. To evaluate REVFinder, we perform a case study on four open source systems.

Part III: Reviewer Involvement in Modern Code Review Processes

We perform two empirical studies to address the reviewer involvement problem:

Chapter 6: *The Impact of Reviewer Involvement on Software Quality*

Prior work provides an evidence that software modules with many patches that lack reviewer involvement (i.e., the large proportion of patches without reviews or discussions) are more likely to contain defects in the future [73, 74]. Yet, little is known about how much reviewer involvement is “enough” to mitigate the risk of future defects. Moreover, it is unclear if developers are carefully reviewing changes to risky areas of a codebase. Hence, we set out to evaluate the impact that characteristics of MCR reviewer involvement have on software quality.

Chapter 7: *Identifying Characteristics of Patches with Poor Reviewer Involvement*

To help improve reviewer involvement, we investigate the factors that influence reviewer involvement in MCR. To that end, we analyze statistical regression models that are trained using 196,712 reviews spread across three open source systems in order to better understand the characteristics of patches that: (1) do not attract reviewers, (2) are not discussed, and (3) receive slow initial feedback.

1.3. Thesis Contribution

We demonstrate that:

1. Developers who solely review patches of other developers account for the largest proportion of developers who contribute to that module. (Chapter 4)
2. Modules without future defects tend to have a smaller proportion of developers with major reviewing expertise than their clean counterparts do. On the other hand, the proportion of developers who have neither authored nor reviewed many patches to a module share a strong increasing relationship with defect-proneness of that module. (Chapter 4)
3. Patches where an author could not initially find appropriate reviewers tend to take a longer time in MCR processes than patches without such a problem. (Chapter 5)
4. REVFINDER can accurately recommends (within top 5 recommendation) an appropriate reviewer for 41%-79% of reviews. (Chapter 5)
5. Modules that will have defects in the future tend to be reviewed less rigorously than their future-defect-free counterparts. (Chapter 6)
6. Modules that have been historically defective tend to have less reviewer involvement than their defect-free counterparts. (Chapter 6)
7. Past reviewer involvement tendencies and the level of detail in the description of a patch can be used to accurately explain whether a patch will suffer from poor reviewer involvement. (Chapter 7)

1.4. Thesis Organization

The remainder of this thesis is organized as follows. Chapter 2 provides background information and defines key terms. Chapter 3 presents research related to our studies. Chapter 4 studies the impact of reviewer selection on software quality. Chapter 5 presents REVFINDER, our approach to select appropriate reviewers. Chapters 6 and 7 present empirical studies of the impact of reviewer involvement on software quality and the factors that influence reviewer involvement, respectively. Finally, Chapter 8 draws conclusions, presents a summary of the main contributions of this thesis, and outlines promising avenues for future work.

Part I.

The Modern Code Review Processes

The Code Review Process

Code review is the practice of having team members examine each others' code changes in search of potential problems early in the development cycle. Many code review processes, e.g., Software Inspections and Walkthroughs were developed in order to effectively perform code reviews that improve the quality of software products.

In this chapter, we provide a broad background of code review processes that were used in the past. Then, we describe the Modern Code Review (MCR) process.

2.1. Traditional Code Review

2.1.1. Formal Software Inspection

A formal software inspection is a well-structured code review process, that was famously advocated by Fagan [34]. The objective of a software inspection is to improve the quality of a software artifact (e.g., software module) by analyzing the artifact, detecting and removing defects before the software product is released. The software inspection process consists of six well-defined steps:

1. *Planning*: An inspection team is formed. Each team member has a specific

role, i.e., moderator, author, reader, scribe, and reviewers. Each role requires specific skills and knowledge. Thus, the selection of team members who occupy each role is important. For example, the developer who occupies a reviewer role should be a developer who is working on similar or interfacing components.

2. *Overview:* The inspection team is assigned and informed about an artifact that will be reviewed.
3. *Preparation:* Each reviewer in the inspection team studies the artifact.
4. *Examination:* The inspection team holds a meeting. The objective of the meeting is to document the defects that each reviewer found. During the meeting, approaches to address are not discussed. The meeting is managed by the moderator, and should not last longer than two hours.
5. *Rework:* The documented defects are fixed by the author.
6. *Follow-up:* The moderator verifies the fixes that were produced during the rework step.

Although software inspection has been demonstrated to be as an effective method to improve the quality of software products [2, 5, 34, 104], the rigid and formal nature of the software inspection process is not well perceived in terms of management. Votta argues that software inspection is very time-consuming, as it requires time for preparation and an inspection meeting [130]. Mashayekhi *et al.* argue that the face-to-face meetings are costly, while an inspection meeting covers a small portion of the product like software modules [71]. Moreover, Shull and Seaman report that due to its cumbersome, time-consuming, and synchronous nature, the adoption of the software inspection process is far from universal [112].

2.1.2. Informal Code Reviews

In contrast to the software inspection, less formal code review processes were introduced. For example, walkthroughs are informal code reviews where an author sets up a meeting and invites teammates to critique a software artifact (e.g., a software module). The focus of the meeting is to find and resolve problems in the artifact.

Email-based code review is another informal code review process that is widely used in open source projects [57,99,101,131]. A code review begins with an author who broadcasts a review request for a patch through project's mailing lists. Then, developers who are interested in the patch provide feedback by replying to the email. Rigby *et al.* observe two common processes of email-based reviews: Review-Then-Commit (RTC) and Commit-Then-Review (CTR) [100]. In the RTC process, every patch has to be reviewed before integration and the patches that do not satisfy reviewers cannot be integrated into the code base. In contrast, the CTR process allows patches to be integrated into the code base before review. Although prior studies have found that email-based code reviews can support asynchronous coordination in distributed software development teams and can improve the quality of software products [4, 87, 100], a lack of traceability can lead to poor process management [98].

2.2. Modern Code Review

In recent years, many modern software organizations have adopted Modern Code Review (MCR), which is based on collaborative tools. MCR tools provide a lightweight code review environment to manage reviewing processes that are similar to the formal software inspection, while allowing for asynchronous collaboration during code reviews similar to an email-based code review process.

As MCR tools tightly integrate with VCS repositories, MCR processes have

become a critical gate of quality control. In general, every new patch needs to pass through the code review process of the MCR tool before the patch can be integrated into upstream VCS repositories. If reviewers find some problems in a patch during the MCR process, the patch needs to be revised until the reviewers agree that the patch is of sufficient quality.

There are several MCR tools that support the code review process in modern software development. For instance, *Gerrit* is a web-based code review tool that is widely used in both proprietary and open source projects (e.g., eBay and Eclipse).¹ *ReviewBoard* is an alternative web-based code review tool that is used by several modern software development teams like VMware [9]. *CodeFlow* is an internal code review tool for Microsoft teams [6]. *Phabricator*, which is developed by Facebook, is used by several software teams at Dropbox.² GitHub also provides a code review system for pull requests, i.e., requests to integrate a patch (or set of patches) from the local VCS repositories of an author to the main VCS repository of a project [40].

These MCR tools provide similar code review processes with similar features like file-by-file difference, in-line commenting, and discussion threads [98]. Patches are updated and re-submitted to the MCR tools until the reviewers agree with the patches. To allow a patch to be integrated into VCS repositories, for instance, reviewers in Gerrit provide a review score [121], reviewers in ReviewBoard mark the patch as *approved* [9], and reviewers in CodeFlow *sign off* on the review [6]. Some MCR tools provide additional features to support the software development process. For example, Phabricator provides an issue tracking system and documentation portal. Nevertheless, the main goal of these MCR tools is to support the collaboration among developers while increasing traceability and integrating code reviewing practices with the current software development practices.

¹<https://www.gerritcodereview.com/>

²<http://phabricator.org/>

2.3. An Overview of the Modern Code Review Process

Process

Figure 2.1 provides a general overview of code review process in MCR tools. There are two main steps in the MCR process:

1. *Review preparation:* The patch author prepares the proposed patch for review by describing code changes and assigning reviewers.
2. *Review execution:* The patch is examined by reviewers and automatically tested. The patch author revises the patch if the reviewers or automated tests find problems in the patch. Once the patch is accepted by the reviewers and passes the automatic tests, the MCR tool will automatically integrate the patch into the upstream VCS repositories.

Below, we describe each step in the MCR process. Since the MCR process of the case studies in this thesis are based on Gerrit³ which is a popular web-based code review tool, we provide the detail of the MCR process based on the code review process in Gerrit.

2.3.1. Review Preparation in Modern Code Review

In Gerrit, the review preparation step mainly consists of two main stages:

Uploading a patch

To upload a proposed patch, a patch author commits the patch to the VCS repositories of Gerrit. In the commit, the patch author describes the changes in the commit message. If the intent of the patch is to address an issue, the patch author can link the patch to its issue report by including the issue ID in the

³<https://www.gerritcodereview.com/>

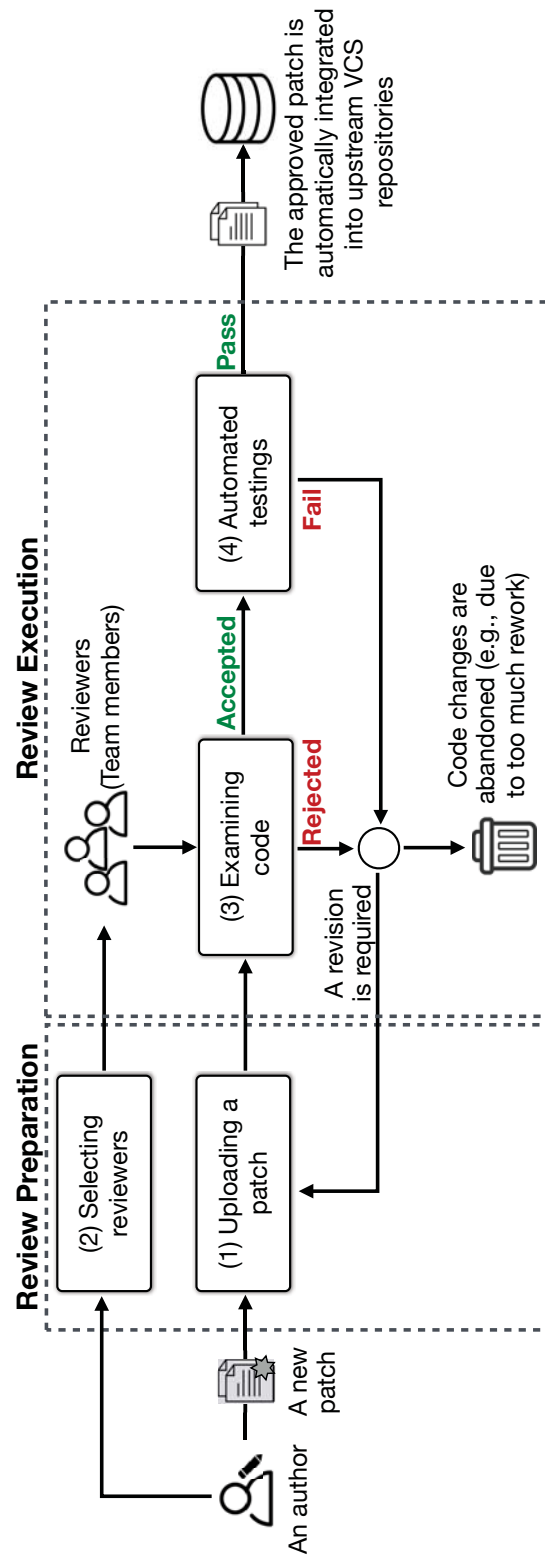


Figure 2.1. An overview of the MCR process.

commit message. Once the patch is uploaded, Gerrit automatically generates a new review request and assigns a *Change-ID* to the patch. During code review, an author can upload several revisions to improve the patch by including the *Change-ID* in the commit message.

Figure 2.2 shows an example of a review request in Gerrit, which consists of four main components, i.e., patch properties, a reviewer list, change sets, and a discussion thread. The patch properties consist of eight fields of information. The *Change-ID* field shows an identification number that Gerrit assigns to the patch. The *Owner* field shows the name of the patch author who initially uploaded the patch. The *Project* and *Branch* fields show the sub-project (or VCS repository) and the branch to which the patch will be integrated. The *Uploaded* field shows the date that the initial patch was uploaded, while the *Updated* field shows the date of the latest reviewing activities (e.g., uploading a revised patch or posting messages). The *Status* field shows the current status of the review, i.e., *review in progress*, *merged*, and *abandoned*. The *Commit Message* field describes the changes in the patch. The reviewer list shows a list of developers who the patch author assigned to the review. The change sets show the list of patch revisions and files that are changed by the patch.

Selecting reviewers

To assign a review task, the patch author adds the email address or name of a developer into the review request form. Then, Gerrit notifies the developers that they have been assigned to review the patch. Note that there is no explicit criterion for selecting reviewers. The patch author can select any team member(s) to review the patch, assign the author herself, or ask team members to recommend appropriate reviewers [124].

The screenshot displays the Qt Open Governance Code Review interface. Key components include:

- Change Properties:** A sidebar on the left showing details like Change-Id, Owner (Victor Heng), Project (qt-creator/qt-creator), Branch (4.0), Topic, Upload/Update dates, Submit Type (Cherry Pick), and Status (Review in Progress).
- Commit Message:** A section titled 'Commit Message' with a 'Permalink' button, containing the message: 'JsonWizard: Check widget visibility when determining if it's mandatory'. Below it, a note states: 'The wizard is unable to advance when a mandatory field is hidden. Assume a hidden widget is not mandatory.'
- Reviewer List:** A section titled 'Reviewer' with a 'Code-Review' tab, showing a list of reviewers (Need Code-Review) and an 'Add Reviewer' button.
- Change Sets:** A section titled 'Patch Set 1' showing the author (Victor Heng), committer (Victor Heng), parent(s), and download links. Below this is a table of change sets with columns: File Path, Comments, Size, Diff, and Reviewed. The table shows changes to 'Commit Message' and 'src/plugins/projectexplorer/jsonwizard/jsonfieldpage.cpp'.
- Discussion Threads:** A section titled 'Comments' with an 'Add Comment' button.

Figure 2.2. An example of a review request in the MCR tool (Gerrit) of the Qt project.

2.3.2. Review Execution in Modern Code Review

The review execution step in Gerrit consists of two main stages:

Examining code

In the code examination stage, reviewers examine the technical content of the patch and provide feedback to the patch author by posting either directly in the discussion thread or using in-line comments in the patch itself (see Figure 2.3).

To indicate the review decision, reviewers provide a review score ranging between -2 and $+2$, where positive values indicate agreement and negative values indicate disagreement. More specifically, a review score of $+1$ indicates that the

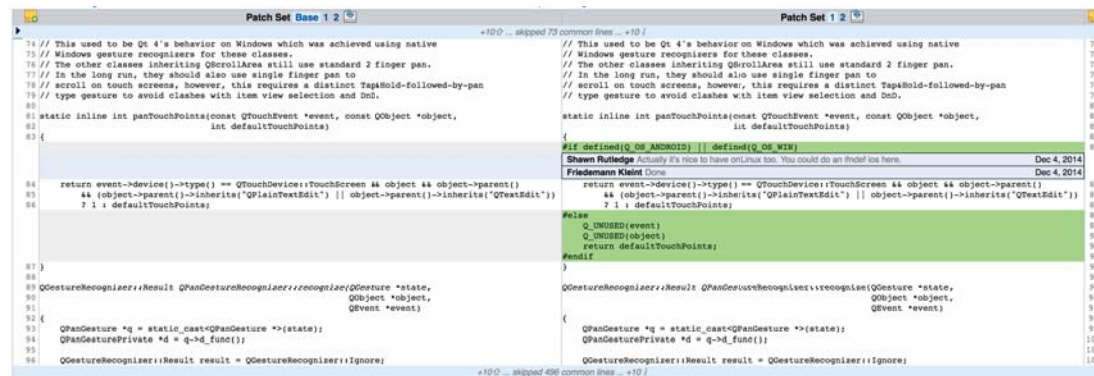


Figure 2.3. An example of file-by-file difference viewing and inline commenting of Qt review ID #101397.

patch is of sufficient quality from the reviewers' viewpoint, yet they need a confirmation from other reviewers. A review score of +2 indicates that the patch is of sufficient quality and is ready to be integrated into upstream VCS repositories. Similarly, for negative scores, a value of -1 indicates that the patch should be revised before integrating into upstream VCS repositories. A review score of -2 indicates that reviewer found a critical issue.

The patch author can revise the patch to address the feedback that was provided by the reviewers, and upload a new revision of the patch to Gerrit. Then, the reviewers re-examine the new revision of the patch. Once (at least) one reviewer assigns it a review score of +2, the latest revision of the patch will be moved to the next stage of automated testing. The patch author or reviewers may also mark the review status as *Abandoned* if the patch is irredeemably flawed or requires too much rework.

Automated testing

Prior to integration into upstream VCS repositories, automatic systems perform more rigorous testing on the patch. For example, Continuous Integration (CI) systems perform regression tests to detect whether there are compilation errors when the patch is applied to the latest version of upstream VCS repositories.

If the patch does not pass automated testing, the patch needs to be revised according to the testing reports, and a new revision needs to be uploaded. Once the patch passes the automated tests, Gerrit automatically integrates the patch into upstream VCS repositories and marks the review status as *Merged*.

2.4. Chapter Summary

This chapter provides a broad background of formal software inspection and informal code reviews that were used in the past, as well as an overview of code review processes that are supported using MCR tools. In particular, we define mechanisms in the review preparation and execution steps of Gerrit.

In the next chapter, we survey prior research on code reviews in order to situate our empirical studies with respect to the literature.

Related Research

In this chapter, we survey the related research on code reviews. We organize the work along the review preparation and execution of the MCR processes. More specifically, we describe how the related work motivates our four empirical studies.

3.1. Review Preparation

Review preparation helps reviewers better understand the proposed patch and provide useful feedback.

3.1.1. Patch upload

Much research suggests that a proposed patch should be small, i.e., the number of changed lines should be small. For example, Weißgerber *et al.* report that small patches are more likely to be integrated into VCS repositories than large patches for the FLAC and OpenAFS projects [131]. Similarly, Tao *et al.* find that the reviewers of the Eclipse and Mozilla projects tend to reject patches due to the large size of the patches [118]. Baysal *et al.* report that large patches have more re-work done before they are eventually integrated into VCS repositories than small patches do. Rigby *et al.* indicate that the patch size has a large impact on review interval [99]. Moreover, the reviewers of the Apache HTTP server project

suggest that an author should break a large patch into several small patches of logical and functional steps [101]. Furthermore, prior studies show that the patch size is associated with the risk of having defects in the patch [80] and with the defect density of a software system [84].

To help authors make small patches, recent work proposes approaches to decompose a large patch. For example, Barnett *et al.* developed CLUSTERCHANGES which is a lightweight static analysis technique in order to cluster areas of code changes that are related to each other [10]. Tao *et al.* proposed an approach to automatically partition code changes in a proposed patch based on semantic dependencies and logical change patterns [119].

Prior work also find that the description of a proposed patch can have an impact on the reviewing performance. Rigby and Storey find that the description of changes is important for reviewers to understand the purpose and the detail of changes [101]. Tao *et al.* find that missing or inconsistent documentation is one of the decisive reasons for patch rejection [118].

3.1.2. Reviewer selection

Selecting a developer to whom tasks or questions should be delegated is important for software productivity and quality. Code ownership concepts can help software development teams to estimate expertise of developers and establish a chain of responsibility. Many prior studies propose an approach to estimate the expertise of developers using code authoring activity. For example, Mockus and Herbsleb use Experience Atoms (EA), i.e., the development and maintenance tasks that a developer has completed, as a measure of developer expertise [79]. Schuler and Zimmermann create an expertise profile from a set of methods that developers have implemented or referenced [107]. Bird *et al.* estimate expertise of developers using the proportion of patches that developers have authored [21]. Moreover, several studies complement authorship data with the data that is recorded during developer IDE interactions [37, 38, 52, 102].

Moreover, several studies find that code authoring expertise is associated with defect-proneness. For example, Pinzger *et al.* show that contribution networks that are built from author contributions can identify defect-prone modules in the Microsoft Windows Vista system [90]. Meneely *et al.* find that there is an association between the number of commits that developers have made to a module and the incidence of security-related problems in the Red Hat Enterprise Linux 4 kernel [76]. Rahman and Devanbu also report that an area of source code that has been associated with defects in the past tends to be written by developers with low code authoring expertise [93].

In addition to code authoring expertise, the current review practices may help developers gain an expertise as well. Several studies find that reviewers in MCR processes are actively involved in a proposed patch more than just uncovering defects. The reviewers often help the author to improve the proposed patch by suggesting alternative solutions [16, 118, 128] or even providing updates to the patch themselves [100, 121]. Bacchelli and Bird also find that code reviews at Microsoft provide additional benefits to development teams, such as knowledge transfer among team members [6].

Moreover, recent work provides an empirical evidence of the importance of reviewing expertise. Meneely *et al.* find that the likelihood of having vulnerability-related problems in files is associated with the number of reviewers who have security experience [75]. Baysal *et al.* indicate that the choice of reviewers plays an important role on reviewing time [14]. Yet, it is not clear whether selecting developers based on reviewing expertise to examine a patch in MCR processes can reduce the risk of having defects. We, therefore, set out to investigate the relationship between the reviewing expertise of developers and the likelihood of having post-release defects in a module.

Empirical Study 1: The impact of reviewer selection on software quality.

Automated identification of relevant experts helps team to expedite the software development processes, especially for geographically distributed development teams. Several prior studies proposed approaches for expert recommendation in order to support software development. For example, for the bug triage process, Anvik *et al.* use machine learning techniques to recommend developers who should be responsible for a new bug issue report [3]. Shokripour *et al.* identify responsible developers using the information that is recorded in a bug issue report and the history of fixed files [110]. Xia *et al.* propose a developer recommendation using the information of a bug issue report and developers [133]. Lately, Tian *et al.* propose an expert recommendation system for Question & Answer communities using topic modeling and collaborative voting scores [126].

Such support tool may also be needed in MCR processes. Tsay *et al.* find that some pull requests in GitHub might await merging for over two months [128]. Weißgerber *et al.* find that some patches in the FLAC and OpenAFS projects were waiting more than two weeks before being integrated [131]. Rigby and Bird report that the reviewing time in industrial and several open source projects is ranging from one minute to as much as 365 days [97].

To better understand the long delay of MCR reviews, several studies investigate the factors that can influence reviewing time. Jiang *et al.* find that the reviewing time is impacted by the patch submission time, the number of affected subsystems, the number of reviewers and the experience of an author [57]. Bosu *et al.* observe that the reviewing time of patches that are submitted by core developers is shorter than patches that are submitted by peripheral developers [24]. Pinzger *et al.* also find that the developer's contribution history, the size of the project, the test coverage, and the project's openness to external contributions are the main factors that influence reviewing time in pull requests at GitHub [40].

Indeed, recent work points out that selecting developers who should be assigned for a review can be a time consuming task [9]. With the aim of reducing human effort, Balachandran proposed REVIEWBOT to recommend reviewers for

the industrial setting of VMware [9]. REVIEWBOT uses the line-by-line modification history of source code to recommend an appropriate reviewer for new code changes. However, there is likely a case where lines of source code are rarely changed like in projects that are already in the maintenance phase. Then, the history of changed lines is not sufficient to be used to identify appropriate reviewers. To this end, we propose REVFINDER that leverages the similarities between the path of a newly changed file and the paths of previously reviewed files.

Empirical Study 2: Selecting appropriate reviewers by leveraging the similarities between the path of a newly changed file and the paths of previously reviewed files.

3.2. Reviewer Involvement

MCR does not impose strict execution criteria. Many studies investigate the current reviewing practices of MCR processes. For example, Gousios *et al.* explore the amount of reviewer involvement (i.e., the number of reviewers and discussion length) in reviews of pull requests at GitHub [40]. Similarly, Rigby and Bird observe the number of reviewers, reviewing time, and the number of revisions in MCR reviews at Microsoft and within several open source projects [97]. Furthermore, prior work suggests that patches should be reviewed by at least two developers to maximize the number of defects found during the review, while minimizing the reviewing workload on the development team [91, 97, 106]. Baysal *et al.* report that the collaborations between organizations in the WebKit project can have an impact on the *positivity*, i.e., the proportion of accepted patches [14]. Rigby *et al.* report that the level of review participation is the most influential factor in the code review efficiency [99].

The reviewer concern in MCR processes is another aspect that has been extensively explored. Beller *et al.* observe a 75:25 ratio of maintainability-related and functional defects that were raised in the MCR reviews of several open source projects [16], which is a similar ratio as in the code reviews of student and commercial projects [70]. Tsay *et al.* report that reviewers are concerned with the appropriateness of a code solution and often provide alternative solutions during reviews of pull requests at GitHub [128]. Tao *et al.* report that risk concerns are frequently raised in the code reviews at Microsoft while it is often difficult to quantify the risk of proposed patches [117].

Other recent work has analyzed the usefulness of reviewer feedback in the MCR process. Pangsakulyanont *et al.* find that the semantic similarity between reviewer comments and the commit message can be used as an indicator of MCR comment usefulness [88]. Bosu *et al.* uncover characteristics of useful reviewer comments and investigate the factors that influence the comment usefulness density in reviews at Microsoft [26].

While MCR reviewer involvement has been explored in prior work, the impact of the reviewer involvement in MCR processes on software quality remains largely unexplored. Lately, empirical studies have provided an evidence that the review investment, i.e., the proportion of patches that are reviewed or are discussed, in a module has an impact on software quality [73, 74] and software design quality [81]. However, besides the review investment, little is known about the impact of other dimensions of reviewer involvement (e.g., code revisions and addressed reviewing concerns) on software quality. To this end, we characterize the reviewer involvement in MCR processes and evaluate its impact on software quality.

Empirical Study 3: The impact of reviewer involvement on software quality.

Although code reviews require the active involvement of reviewers to critique the proposed patches [1,96], recent studies uncover that patches might be ignored during the MCR process [17,19,87,101]. For example, Rigby and Storey indicate that patches can be ignored if they do not match with the interests of members of the core development team [101]. Bettenburg *et al.* report that in the voluntarily code reviews of open source projects, there is a risk that a contribution is completely ignored, when no volunteer steps up for a review [17].

In addition to the review participation, several studies provide evidence of the impact of other dimensions of reviewer involvement on reviewing performance. Prior work finds that careful consideration of the implications of changes improves their overall quality prior to integration [6,128]. Moreover, Bettenburg *et al.* indicate that a well-functioning code review process should yield responses to a new review request in a timely manner in order to avoid potential problems in the development process [17]. For example, due to continuous software development practices [36], it is possible that if a patch receives slow initial feedback, it can become outdated, requiring updates to be re-applied (and possibly re-implemented) to the latest version of the system. Moreover, Rigby *et al.* suggest that the earlier that a patch is reviewed, the lower the risk of deeply embedded defects [100].

In the MCR process, however, some patches are merged into upstream VCS repositories even though they do not have any reviewer involved nor feedback is provided in their reviews apart from the patch author [73,81]. It is not known whether there are characteristics of patches that share a relationship with the likelihood of receiving such poor reviewer involvement. A good understanding of these characteristics will help projects create mitigation strategies to avoid such poor participation.

Empirical Study 4: Identifying characteristics of patches with poor reviewer involvement.

3.3. Chapter Summary

In this chapter, we survey prior research along the review preparation, and execution of MCR processes. From the survey, we find that (1) little is known about the impact of reviewer selection and involvement on software quality and (2) MCR tools need additional features to support and increase the effectiveness of code review processes.

Broadly speaking, the remainder of this thesis describes our empirical studies that set out to better understand and improve the reviewer selection (Chapters 4 and 5), and in the reviewer involvement in MCR processes (Chapters 6 and 7).

Part II.

Reviewer Selection in Modern Code Review Processes

The Impact of Reviewer Selection on Software Quality

An earlier version of the work in this chapter appears in the Proceedings of the 38th International Conference on Software Engineering (ICSE) [123].

A patch author should assign a review task to developers who have an expertise or experience that is related to the impacted module. However, the lightweight MCR does not impose strict criteria for selecting reviewers. Recent work finds that the choice of reviewers has an impact on reviewing performance of MCR [14, 75]. Motivated by the findings of prior work, we suspect that the choice of reviewers may also have an impact on software quality. Hence, in this chapter, we adopt the concept of code ownership to estimate the expertise of developers. In particular, we include code review activity into traditional code ownership heuristics which are solely derived from code authoring activity [21], then examine the relationship between the level of reviewing expertise and defect-proneness of software products. The results of our case studies demonstrate that when considering code review activity, we find developers who are actively involved

in software projects through only reviewing patches, instead of authoring them. Our results also show that a code ownership heuristic that is aware of code review activity can reverse the association between code authoring expertise and defect-proneness. Our findings suggest that considering reviewing expertise in addition to authoring expertise when selecting a reviewer for a new patch can decrease the likelihood of having future defects.

4.1. Introduction

Code ownership is an important concept for large software teams. In large software systems, with hundreds or even thousands of modules, code ownership is used to establish a chain of responsibility. When tasks or questions need to be addressed to a module with strong code ownership, there is a module owner who is responsible for it. On the other hand, it is more difficult to identify the developer to whom tasks should be delegated in modules with weak code ownership.

In the literature, code ownership is typically estimated using heuristics that are derived from code authorship data. For example, Mockus and Herbsleb use the number of tasks that a developer has completed within a time window that modify a given module to identify the developer that is responsible for that module [79]. Furthermore, Bird *et al.* estimate code ownership for a developer within a module by computing the proportion of code changes that the developer has authored within that module [21]. Rahman and Devanbu estimate code ownership at a finer granularity by computing the proportion of lines of changed code that each developer has authored [93]. Indeed, the intuition behind traditional code ownership heuristics is that developers who author the majority of changes to a module are likely to be the owners of those modules.

However, in addition to authorship contributions, developers can also contribute to the evolution of a module by critiquing code changes that other developers have authored in that module. Indeed, many contemporary software

development teams use Modern Code Review (MCR), a tool-based code review process, which tightly integrates with the software development process [97]. In the MCR process, code changes are critiqued by reviewers (typically other developers) to ensure that such code changes are of sufficient quality prior to their integration with the codebase.

While several studies show that the modern code review activity shares a link to software quality [65, 73, 121], the MCR process is more than just a defect-hunting exercise. Indeed, Morales *et al.* show that developer involvement in the MCR process shares a relationship with software design quality [81]. Bacchelli and Bird report that the focus of code review at Microsoft has shifted from being defect-driven to collaboration-driven [6]. Others report that reviewers in MCR processes often suggest alternative solutions [16, 118, 128] or even provide updates to the code changes themselves [121].

Despite the active role that reviewers play in the MCR process, reviewer contributions are disregarded by traditional code ownership heuristics. For example, a senior developer who reviews many of the code changes to a module, while authoring relatively few will not be identified as a module owner by traditional code ownership heuristics. Therefore, in this chapter, we set out to complement traditional code ownership heuristics using data that is derived from code review repositories. More specifically, we adapt the popular code ownership heuristics of Bird *et al.* [21] to be: (1) *review-specific*, i.e., solely derived from code review data and (2) *review-aware*, i.e., combine code authorship and reviewing activities. We then use review-specific and review-aware code ownership heuristics to build regression models that classify modules as defect-prone or not in order to revisit the relationship between the code ownership and software quality. Through a case study of six releases of the large Qt and OpenStack open source systems, we address the following three research questions:

(RQ1) How do code authoring and reviewing contributions differ?

Motivation. While prior work examines the contributions of developers in terms of code authorship [21, 37, 38, 76, 90, 93], the reviewer contributions that these developers make to a module still remains largely unexplored. Hence, we first study how developers contribute to the evolution of a module in terms of code authorship and review.

Results. 67%-86% of the developers who contribute to a module did not author any code changes, yet they participated in the reviews of 21%-39% of the code changes made to that module. Moreover, 18%-50% of these review-only contributors are documented core team members.

(RQ2) Should code review activity be used to refine traditional code ownership heuristics?

Motivation. Prior work uses defect models to understand the relationship between traditional code ownership heuristics and software quality [21, 28, 42, 85]. Such an understanding of defect-proneness is essential to chart quality improvement plans, e.g., developing code ownership policies. Furthermore, Bird *et al.* find that modules with many minor authors (i.e., developers who seldom wrote code changes in that module) are likely to be defective [21]. However, there are likely cases where these minor authors review several of the code changes in that module. Hence, we investigate whether refining traditional code ownership heuristics using code review activity will provide a more comprehensive picture of the relationship between code ownership and software quality.

Results. 13%-58% of developers who are flagged as minor contributors of a module by traditional code ownership heuristics are actually major contributors when their code review activity is taken into consideration. Moreover, modules without post-release defects tend to have a large proportion of developers who have low traditional code ownership values but high review-

specific ownership values. Conversely, the modules with post-release defects tend to have a large proportion of developers who have both low traditional and review-specific ownership values. Moreover, when controlling for several factors that are known to share a relationship with defect-proneness, our defect models show that the proportion of developers who have both low traditional and review ownership values shares a strong, increasing relationship with the likelihood of having post-release defects.

Our results lead us to conclude that code review activity provides an important perspective that contributes to the code ownership concept. Moreover, reviewing expertise of developers can reverse the relationship with defect-proneness. These findings suggests that one should select reviewers who have reviewing expertise of a module (even if they never authored any code changes in that module) in order to mitigate the risk of having defects in the future.

4.1.1. Chapter Organization

The remainder of this chapter is organized as follow. Section 4.2 describes the studied code ownership heuristics in more detail. Section 4.3 describes the design of our case study, while Section 4.4 presents our empirical observations with respect to our three research questions. Section 4.5 discusses our broader implications of our observations. Section 4.6 discloses the threats to the validity of our study. Finally, Section 4.7 provide a summary of this study.

4.2. Background & Definition

In this section, we provide a description of the studied code ownership heuristics.

Code ownership heuristics that operate at different granularities have been proposed in the literature [21, 38, 42, 79, 93]. For example, Bird *et al.* estimate code ownership values using the code change granularity [21]. Fritz *et al.* propose

code ownership heuristics at the element (i.e., class) granularity [38]. Rahman and Devanbu estimate code ownership values at the finer granularity of a code fragment. Since the studied MCR processes are connected with code changes, i.e., proposed changes should be reviewed prior to integration into the VCS, we opt to extend the code ownership heuristics of Bird *et al.* [21] using code review activity.

Below, we describe the traditional, review-specific, and review-aware code ownership heuristics.

4.2.1. Traditional Code Ownership Heuristics

The traditional code ownership heuristics of Bird *et al.* [21] are computed using the authorship of code changes to estimate the code ownership of a developer for a module. For a developer D , **Traditional Code Ownership (TCO)** of a module M is computed as follows:

$$\text{TCO}(D, M) = \frac{a(D, M)}{C(M)} \quad (4.1)$$

where $a(D, M)$ is the number of code changes that D has authored in M and $C(M)$ is the total number of code changes made to M .

In addition, Bird *et al.* also define two levels of developer expertise within a module. Developers with low TCO values (i.e., below 5%) are considered to be *minor authors*, while developers with high TCO values (i.e., above 5%) are considered to be *major authors*.

4.2.2. Review-Specific Ownership Heuristics

We define review-specific ownership heuristics that use code review data to estimate the code ownership of developers according to the reviewer contributions

that they have made to a module. For a developer D , **Review-Specific Ownership (RSO)** of a module M is computed as follows:

$$\text{RSO}(D, M) = \frac{\sum_{k=1}^{r(D, M)} p(D, k)}{C(M)} \quad (4.2)$$

where $r(D, M)$ is the number of reviews of code changes made to M in which D has participated and $p(D, k)$ is a proportion of reviewer contributions that D made to code change k . Since one review can have many reviewers, we normalize the value of review-specific ownership in each code change using $p(D, k)$ in order to avoid having RSO values that do not sum up to 100%. We explore two RSO heuristics by varying the definition of $p(D, k)$:

1. RSO_{Even}: This normalization assumes that every reviewer contributes equally to k . Therefore, this normalization evenly divides the share of reviewer contributions to every reviewer of k .

$$p(D, k) = \frac{1}{R(k)} \quad (4.3)$$

$R(k)$ is the total number of developers who participated in the review of k .

2. RSO_{Proportional}: The intuition behind this normalization is that the more the feedback that D provides during the review of k , the larger the share of reviewer contributions that D made to k . Therefore, this normalization assigns a share of the reviewer contribution to D that is proportional to the amount of feedback (i.e., number of reviewer comments) that D has provided to the review of k .

$$p(D, k) = \frac{c(D, k)}{C(k)} \quad (4.4)$$

$c(D, k)$ is the number of reviewer comments that are provided by D in the review of k , while $C(k)$ is the total number of reviewer comments in the review of k .

Table 4.1. A contingency table of the review-aware ownership expertise category.

		Traditional code ownership	
		$\leq 5\%$	$> 5\%$
Review-specific ownership	$\leq 5\%$	Minor author & minor reviewer	Major author & minor reviewer
	$> 5\%$	Minor author & major reviewer	Major author & major reviewer

Similar to traditional code ownership, we consider developers with low RSO values (i.e., below 5%) to be *minor reviewers*, and developers with high RSO values (i.e., above 5%) to be *major reviewers*. We also perform a sensitivity analysis with RSO threshold values ranging from 2% to 10%. We observe a similar result when using these RSO thresholds (see Appendix B.1).

4.2.3. Review-Aware Ownership Heuristics

In practice, developers act as both authors and reviewers. For example, a developer can be the author of a code change to a module, while also being a reviewer of code changes that other developers have authored to that module. However, the traditional and review-specific ownership heuristics independently estimate code ownership using either authorship or reviewer contributions, respectively.

To address this, we propose review-aware code ownership heuristics. We adapt the traditional code ownership heuristics to be review-aware using the pair of TCO and RSO values that a developer can have within a module. Then, we refine the levels of the traditional code ownership using the levels of review-specific code ownership as shown in Table 4.1. For example, for a module M , if a developer D has a TCO value of 3% and an RSO value of 25%, then D would have a review-aware ownership value of (3%, 25%), which falls into the minor author & major reviewer category.

4.3. Case Study Design

In this section, we outline our criteria for selecting the studied systems and our data preparation approach.

4.3.1. Studied Systems

In order to address our research questions, we perform an empirical study on large, rapidly-evolving open source systems. In selecting the subject system, we identified two important criteria that needed to be satisfied:

Criterion 1: Traceability — We focus our study on systems where the MCR process records explicit links between code changes and the associated reviews.

Criterion 2: Full Review Coverage — Since we will investigate the effect of review-specific and review-aware ownership heuristics on software quality, we need to ensure that unreviewed changes are not a confounding factor [73]. Hence, we focus our study on systems that have a large number of modules with 100% review coverage, i.e., modules where every code change made to them has been reviewed by at least one reviewer other than its author.

To satisfy criterion 1, we began our study with four software systems that use the Gerrit code review tool. To mitigate bias that may be introduced into our datasets through noisy manual linkage [18], we select systems that use the Gerrit code reviewing tool for our analysis. Gerrit automatically records a unique ID that can link code changes to the reviews that they have undergone. We discard VTK, since its linkage rate is too low. We also remove ITK from our analysis, since it does not satisfy criterion 2.

Table 4.2 shows that the Qt and OpenStack systems satisfy our criteria for analysis. Qt is a cross-platform application and UI framework that is developed

Table 4.2. Overview of the studied systems. Systems above the double line satisfy our criteria for further analysis.

System		Commits		Modules			Personnel	
Name	Version	Tag name	Total	Linkage rate	Total	With 100% review coverage	With defects	Authors Reviewers
Qt	5.0	v5.0.0	2,955	95%	389	328 (84%)	70 (21%)	156 156
	5.1	v5.1.0	2,509	96%	450	438 (97%)	77 (18%)	186 170
OpenStack	Folsom	2012.2	2,315	99%	258	241 (93%)	70 (29%)	235 152
	Grizzly	2013.1	2,881	99%	336	326 (97%)	123 (37%)	330 205
	Havana	2013.2	3,583	99%	527	515 (97%)	128 (25%)	451 359
	Icehouse	2014.1	3,021	100%	499	499 (100%)	198 (40%)	499 480
VTK	5.10	v5.10.0	1,431	39%	170	8 (5%)	-	- -
ITK	4.0	v4.3.0	352	97%	218	125 (57%)	-	- -

by the Digia corporation, while welcoming contributions from the community-at-large.¹ OpenStack is an open-source software platform for cloud computing that is developed by many well-known companies, e.g., IBM, VMware, and NEC.²

4.3.2. Data Preparation

We use the review dataset that is provided by Hamasaki *et al.* [46]. The dataset describes patch information, reviewer scoring, the involved personnel, and review discussion. We use the code dataset for the Qt system from prior work [73]. The code dataset describes the recorded commits on the `release` branch of the Qt VCSs during the development and maintenance of each studied Qt release. We also expand the code dataset for the OpenStack system using the same approach as the prior work [73].

In order to produce the datasets that are necessary for our study, we link the review and code datasets, and compute the code ownership heuristics. Figure 4.1 provides an overview of our data preparation process, which is broken down into the following three steps.

(DP1) Link code changes to reviews. Similar to prior work [73], we link the code and review datasets using the change ID, i.e., a review reference that is automatically generated by Gerrit. For each code change, we extract the change ID recorded in the commit message. To link the associated review, Gerrit uses “<subsystem name>__<VCS branch>__<change ID>” as a unique reference. Hence, we extract the commit data from the VCSs to generate a reference, then link the code change to the associated review.

Once the code and review datasets are linked, we measure the review coverage for each module (i.e., directory). Since this study revisits the traditional code ownership heuristics [21] which were previously studied at the module level, we conduct our study at the module level as well to enable the comparability of our

¹<http://qt-project.org/>

²<http://www.openstack.org/>

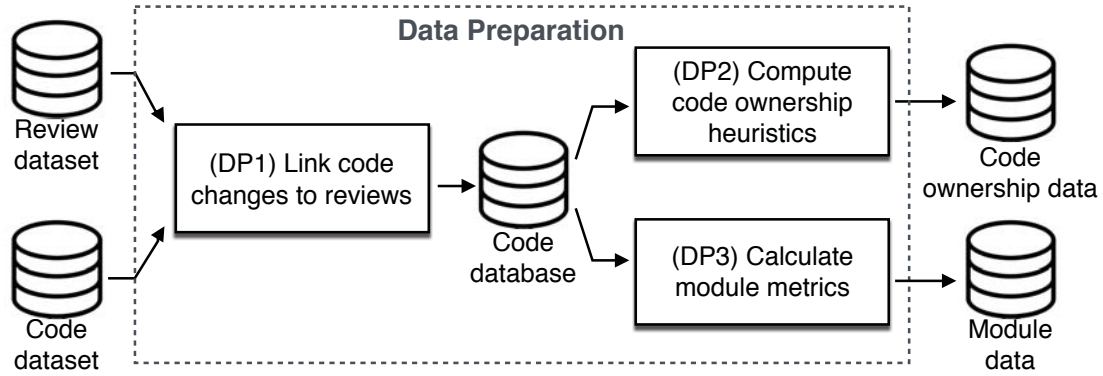


Figure 4.1. An overview of data preparation approach.

results with the prior work. We then remove modules that do not have 100% review coverage from our datasets in order to control for the confounding effect that a lack of review coverage may have [73].

(DP2) Compute code ownership heuristics. To estimate the code ownership of a developer for a module, we first identify code changes that the developer has authored using the owner field as recorded in Gerrit. We then identify the code changes that the developer has reviewed using the reviewer comments that the developer posted. A code change that spans multiple modules is treated as contributing to all of the changed modules.

Next, we estimate code ownership using the traditional, review-specific, and review-aware code ownership heuristics (see Section 4.2) for every developer who was involved with the development of each module. Similar to prior work [58, 74], we use a six-month period prior to each release date to capture the authorship and reviewer contributions that developers made during the development cycle.

(DP3) Calculate module metrics. Prior work finds that several types of metrics have an impact on software quality. Hence, we also measure popular product and process metrics that are known to have a relationship with defect-proneness in order to control for their impact [51, 109].

For the product metrics, we measure the *size* of the source code in a module at the release time by aggregating the number of lines of code in each of the files

of the module. For the process metrics, we use churn and entropy to measure the change activity that occurred during the development cycle of a studied release. We again use the six-month period prior to each release date to capture the change activity. *Churn* counts the total number of lines added to and removed from a module prior to release. *Entropy* measures how the complexity of a change process is distributed across source code files [51]. We measure the entropy of a module using a calculation of $H(M) = -\frac{1}{\log_2 n} \sum_{k=1}^n (p_k \times \log_2 p_k)$, where n is the number of source code files in module M , and p_k is the proportion of the changes to M that occur in file k .

We also detect whether there are post-release defects in each module. To detect post-release defects in a module, we identify defect-fixing changes that occurred after a studied release date. By studying the release practice of the studied systems, we found that the studied systems release sub-versions every two month after the main version is released. Hence, we use a two-month window to capture the defect-fixing changes. Similar to prior work [73], we search the VCS commit messages for co-occurrences of defect identifiers with keywords like “bug”, “fix”, or “defect”. A similar approach is commonly used to determine defect-fixing and defect-inducing changes in prior work [58, 61, 74].

4.4. Case Study Results

In this section, we present the results of our case study with respect to our three research questions. For each research question, we present our empirical observations, followed by a general conclusion.

(RQ1) How do code authoring and reviewing contributions differ?

4.4.1. Approach

To address our RQ1, we examine the contributions of developers in each release of the studied systems. We analyze descriptive statistics of the number of developers who contribute by authoring or reviewing code changes to modules. Since the number of involved developers can vary among modules, we analyze the proportion of developers in a module instead of the actual number of developers.

4.4.2. Results

We now present our empirical observations.

Observation 1 – 67%-86% of developers who contribute to a module did not previously author any code changes, yet they had previously reviewed 21%-39% of the code changes in that module. We find that there are on average 6-8 (Qt) and 17-32 (OpenStack) developers who contribute to a module. Figure 4.2 shows that the developers who previously only reviewed code changes are the largest set of developers who contribute to a module. 67%-70% (Qt) and 67%-86% (OpenStack) of developers are review-only contributors. On average, these review-only contributors have reviewed 29%-39% (Qt) and 21%-31% (OpenStack) of the code changes made to a module. This suggests that many developers contribute by only reviewing code changes to a module, yet their expertise is not captured by traditional code ownership heuristics.

Observation 2 – 18%-50% of the developers who only review code changes made to a module are documented core developers. 44%-51% (Qt) and 18%-21% (OpenStack) of the review-only contributors to modules are documented core developers (see Appendix C). On the other hand, 18%-20% (Qt) and 12%-26% (OpenStack) of developers who have authored a code change to a module are

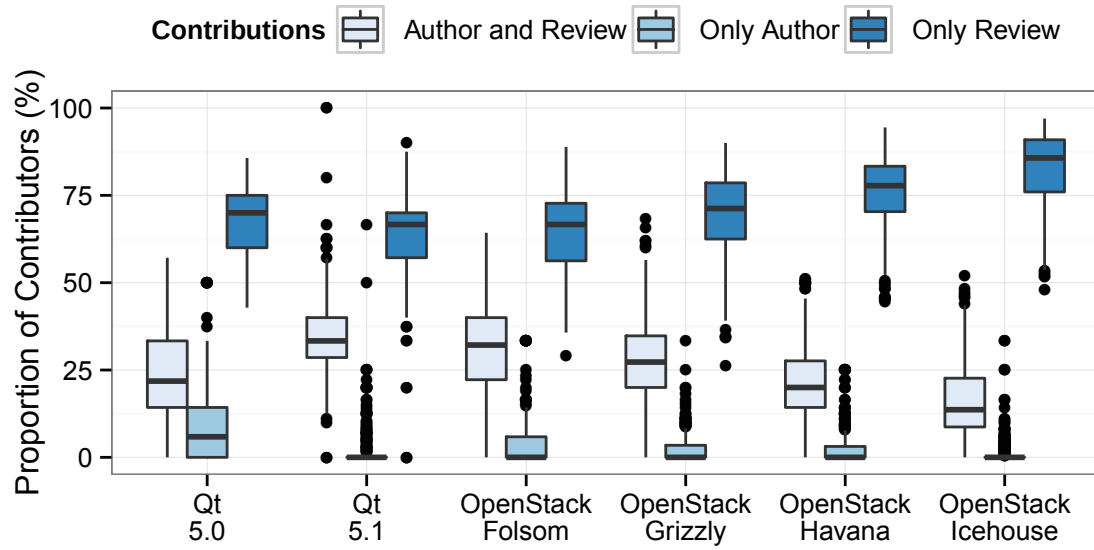


Figure 4.2. Number of developers who contribute to a module (i.e., authoring vs reviewing code changes).

documented as core developers. Moreover, we observe that core developers tend to often contribute to a module as reviewers rather than authors. Indeed, 51%-54% (Qt) and 42%-60% (OpenStack) of modules do not have any code changes authored by core developers. On the other hand, 0%-8% (Qt) and 23%-36% (OpenStack) of modules that do not have any code changes reviewed by core developers.

Summary: The developers who contribute to a module by only reviewing code changes account for the largest set of contributors to that module. Moreover, 18%-50% of these review-only developers are documented core developers of the studied systems, suggesting that code ownership heuristics that only consider authorship activity are missing the activity of these major contributors.

(RQ2) Should code review activity be used to refine traditional code ownership heuristics?

4.4.3. Approach

To address RQ2, we first examine the TCO values of each developer in each module against their RSO values. We then analyze the relationship between review-aware ownership and defect-proneness. To do so, we perform two data analysis approaches: (DA1) statistical analysis and (DA2) defect model analysis. Below, we describe each of our analysis approaches.

(DA1) Statistical Analysis

We use statistical analysis to determine the difference between defective and clean modules in terms of developer expertise. To do so, we compute the proportion of developers in each expertise category of the review-aware ownership heuristic (see Table 4.1), i.e., (1) minor author & minor reviewer, (2) minor author & major reviewer, (3) major author & minor reviewer, and (4) major author & major reviewer for each module. Then, we compare the proportion of developers in each of the expertise categories of defective and clean modules using beanplots [59]. Beanplots are boxplots in which the vertical curves summarize the distributions of different datasets. Defective modules are those that have at least one post-release defect, while clean modules are those that are free from post-release defects.

We use one-tailed Mann-Whitney U tests ($\alpha = 0.05$) to detect whether there is a statistically significant difference among the defective and clean modules in terms of the proportion of developers in the different expertise categories. We use Mann-Whitney U tests instead of T-tests because we observe that the distributions of the proportions of developers do not follow a normal distribution (Shapiro-Wilk test p -values are less than 2.2×10^{-16} for all distributions). We also measure the effect size, i.e., the magnitude of the difference using Cliff's δ [69].

Cliff's δ is considered as negligible for $\delta < 0.147$, small for $0.147 \leq \delta < 0.33$, medium for $0.33 \leq \delta < 0.474$ and large for $\delta \geq 0.474$ [103].

(DA2) Defect Model Analysis

Although our statistical analysis can explain the association between each of the expertise categories and defect-proneness, their effects could be correlated with other metrics that are known to share a relationship with defect-proneness (e.g., module size) [136]. Hence, we examine the impact that review-specific and review-aware code ownership heuristics can have on defect-proneness using defect models that control for several confounding factors. Similar to Bird *et al.* and other work [21, 29, 74, 136], our main goal of building defect models is not to predict defect-prone modules, but to understand the relationship between the explanatory variables and defect-proneness.

To build defect models, we use logistic regression models to fit our studied datasets. We adopt a nonlinear regression modeling approach, which enhances the fit of the data to be more accurate and robust, while carefully considering the potential for overfitting [48]. Our models operate at the module-level, where the response variable is assigned a value of TRUE if a module has at least one post-release defect, and FALSE otherwise. The explanatory variables are outlined in Table 4.3. Similar to prior work [21], we estimate the traditional and review-specific code ownership for a module by using the largest TCO and RSO values of the developers who contributed to that module. We also estimate the review-aware ownership for a module by computing the proportion of developers in each expertise category. Furthermore, in addition to the product and process metrics (i.e., size, churn, and entropy), we control for the number of contributors, authors, and reviewers in our models, since these metrics may have an impact on defect-proneness.

Table 4.3. A taxonomy of the considered control (top) and code ownership metrics (bottom).

Metrics	Description
<i>Control Metrics</i>	
Size	Number of lines of code.
Churn	Sum of added and removed lines of code.
Entropy	Distribution of changes among files.
#Contributor	The number of developers who contribute by authoring or reviewing code changes to the module.
#Author	The number of developers who have authored code changes to the module.
#Reviewer	The number of developers who have reviewed code changes to the module.
<i>Code Ownership Metrics</i>	
Top TCO	The traditional code ownership value of the developer who authored the most code changes to the module.
Top RSO	The review-specific ownership value of the developer who reviewed the most code changes to the module.
<i>Review-Aware Ownership Metrics</i>	
Proportion of minor author & major reviewer	A proportion of developers in the minor author & major reviewer category.
Proportion of major author & major reviewer	A proportion of developers in the major author & major reviewer category.
Proportion of minor author & minor reviewer	A proportion of developers in the minor author & minor reviewer category.
Proportion of major author & minor reviewer	A proportion of developers in the major author & minor reviewer category.

Similar to prior work [74], we adopt the model construction and analysis approaches of [48, p. 79] to allow nonlinear relationships between explanatory and response variables to be modelled. Furthermore, these techniques can enable a more accurate and robust fit of the data, while carefully considering the potential for overfitting (i.e., a model is too specifically fit to the training dataset to be applicable to other datasets). An overfit model will overestimate the performance of the model and exaggerate spurious relationships between explanatory and response variables. To facilitate future research and replication study, we provide an example R script of model construction and analysis in Appendix A.

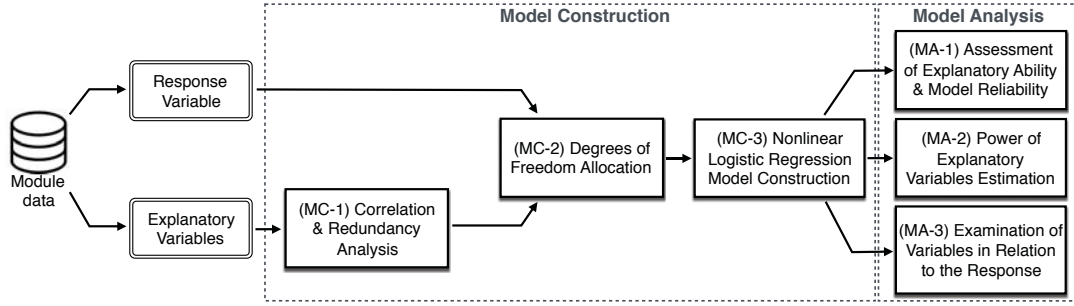


Figure 4.3. An overview of our model construction and analysis approaches.

Figure 4.3 provides an overview of the model construction and analysis approaches, which we describe below. We now describes each step in our model construction and analysis approaches.

(MC-1) Correlation & Redundancy Analysis. Explanatory variables that are highly correlated with each other can interfere with the results of model analysis. Hence, we measure the correlation between explanatory variables using Spearman rank correlation tests (ρ). We then use a variable clustering analysis technique [105] to construct a hierarchical overview of the correlation and remove explanatory variables with a high correlation. According to [54, p. 120], Spearman correlation coefficient values that are greater than 0.7 are considered to be strong correlation. Hence, we select $|\rho| = 0.7$ as our threshold for removing highly correlated variables. We perform this analysis iteratively until all clusters of surviving variables have $|\rho|$ values below 0.7.

Some explanatory variables are not highly correlated but can still be redundant, i.e., variables that do not have a unique signal from the other explanatory variables. Redundant variables in an explanatory model will distort the modelled relationship between the explanatory and response variables. To detect redundant variables, we use the `redun` function in the `rms` R package [49] to fit models that explain each explanatory variable using the remaining explanatory variables. We then remove the explanatory variables where models are fit with an R^2 value greater than 0.9 (the default threshold of the `redun` function).

(MC-2) Degrees of Freedom Allocation. In order to allow nonlinear relationships between explanatory and response variables to be modelled, we must decide how to allocate our budgeted degrees of freedom to each of our explanatory variables. To allocate the degrees of freedom most effectively, the explanatory variables that have more potential for sharing nonlinear relationship with the response variable should be allocated more degrees of freedom than the explanatory variables that have less potential. Hence, we measure the potential for nonlinearity in the relationship between explanatory and response variables using a calculation of the Spearman multiple ρ^2 . A large Spearman multiple ρ^2 score indicates that there is potential for a strong nonlinear relationship between an explanatory variable and the response variable. Nevertheless, we limit the maximum degrees of freedom that we allocate to any given explanatory variable to five in order to minimize the risk of overfitting [48, p. 23]. Finally, we allocate degree of freedom for explanatory variable x_i as follow:

$$\text{D.F.}(x_i) = \begin{cases} 5 & \text{for strong: } \rho^2 > 0.3 \\ 3 & \text{for moderate: } 0.15 < \rho^2 \leq 0.3 \\ 1 & \text{for small: } \rho^2 \leq 0.15 \end{cases}$$

(MC-3) Logistic Regression Model Construction. After removing the highly correlated and redundant variables and allocating the degrees of freedom to the surviving explanatory variables, we fit our logistic regression models to the data. We use the restricted cubic splines of the `rcs` function in the `rms` R package [49] to fit the allocated degrees of freedom to the explanatory variables. We use the restricted cubic splines because the smooth nature of cubic curves will more realistically fit natural phenomena than linear splines, which introduce abrupt changes in direction [48, p. 20].

Once the logistic regression model has been constructed, we analyze the model in order to understand the relationship between the explanatory variables (i.e.,

patch and MCR process metrics) and the response variable (i.e., whether a patch had review participation or not).

(MA-1) Assessment of Explanatory Ability & Model Reliability. To evaluate the performance of our models, we use the Area Under the receiver operating characteristic Curve (AUC) [47]. AUC measures how well a model can discriminate between the potential responses. AUC is computed by measuring the area under the curve that plots the true positive rate against the false positive rate while varying the threshold that is used to determine whether a patch is classified as receiving review participation or not. An AUC value of 1 indicates perfect discrimination, i.e., perfect separation of patches that receive review participation and those that do not, while an AUC value of 0.5 indicates that the model does not discriminate better than random guessing.

Although the AUC can measure the explanatory power, it may overestimate the performance of the model if it is an overfit model. To evaluate the reliability of our models, we estimate the optimism of the AUC using a bootstrap-derived approach [32]. First, the approach trains a model using a bootstrap sample, i.e., a dataset sampled with replacement from the original dataset, which has the same population size as the original dataset. Then, the optimism is estimated using the difference in performance between the bootstrap model when applied to the original dataset and the bootstrap sample. Finally, the approach is repeated 1,000 times in order to compute the average optimism. The smaller the average optimism is, the more reliable the performance estimates of the original fit are.

(MA-2) Power of Explanatory Variables Estimation. Similar to prior work [74], we use Wald statistics to estimate the impact that each explanatory variable has on the model’s performance. Since the explanatory variables that were assigned additional degrees of freedom are represented in the model by multiple terms, Wald statistics are used to jointly test all model terms that relate to a given explanatory variable. For the tests, we use the `anova` function in the `rms` R package [49] to estimate the relative contribution (Wald χ^2) and the statistically

significance (p-value) of each explanatory variable in the model. The larger the Wald χ^2 value is, the larger the explanatory power that a particular explanatory variable contributes to the performance of the model.

(MA-3) Examination of Variables in Relation to the Response. The power of explanatory variables indicates the magnitude of the impact that an explanatory variable has on model performance, yet it does not provide a notion of the direction or the shape of the relationship between the explanatory variables and the response. To better understand the direction and shape of these relationships, we plot the likelihood of module defect-proneness produced by our models against an explanatory variable while holding the other explanatory variables at constant values. We use the `Predict` function in the `rms` R package [49] to compute and plot the odds values for various explanatory variables.

4.4.4. Results

We now present the results of statistical and defect model analyses.

(DA1) Statistical Analysis

Observation 3 – 13%-58% of minor authors are major reviewers. When we assume that reviewers contribute equally (i.e., using RSO_{Even}), 41%-58% (Qt) and 13%-22% (OpenStack) of minor authors fall in the minor author & major reviewer category. Similarly, when we assume that reviewers who post more comments make more of a contribution (i.e., using $\text{RSO}_{\text{Proportional}}$), 32%-48% (Qt) and 11%-18% (OpenStack) of minor authors fall in the minor author & major reviewer category. This indicates that many minor authors make large reviewer contributions.

Observation 4 – Clean modules tend to have more developers in the minor author & major reviewer category than defective modules do. Figure 4.4(a) shows that when we use RSO_{Even} , the proportion of developers in the minor

author & major reviewer category of clean modules is larger than that of defective modules. Mann-Whitney U tests confirm that the differences are statistically significant ($p < 0.001$), with medium or large effect sizes ($0.468 \leq \delta \leq 0.698$) for all of the studied datasets. Table 4.4 shows that when using either RSO_{Even} or $\text{RSO}_{\text{Proportional}}$, the differences are statistically significant, with medium or large effect sizes. Our results indicate that post-release defects occur less frequently in the modules with a large proportion of developers in the minor author & major reviewer category than modules with a smaller proportion of developers in minor author & major reviewer category.

Observation 5 – Conversely, defective modules tend to have more developers in the minor author & minor reviewer category than clean modules do. Figure 4.4(b) shows that when we use RSO_{Even} , the proportion of developers in the minor author & minor reviewer category of defective modules is larger than that of clean modules. Mann-Whitney U tests confirm that the differences are statistically significant ($p < 0.001$), with medium or large effect sizes ($0.444 \leq \delta \leq 0.708$) for all of the studied datasets. Similarly, when using $\text{RSO}_{\text{Proportional}}$ —Mann-Whitney U tests confirm that the differences are statistically significant ($p < 0.001$), with medium or large effect sizes ($0.435 \leq \delta \leq 0.663$) for all of the studied datasets. Our results indicate that post-release defects occur more frequently in the modules with a large proportion of developers in the minor author & minor reviewer category than those with a smaller proportion of developers in minor author & minor reviewer category.

We observe similar differences among the defective and clean modules for the major author & minor reviewer and the major author & major reviewer categories. When we use either RSO_{Even} or $\text{RSO}_{\text{Proportional}}$, Table 4.4 shows that the proportion of developers in the major author & minor reviewer category of defective modules is significantly larger than that of clean modules, with small or medium effect sizes for the Qt datasets ($0.158 \leq \delta \leq 0.343$). Conversely, the proportion of developers in the major author & major reviewer category of defective modules

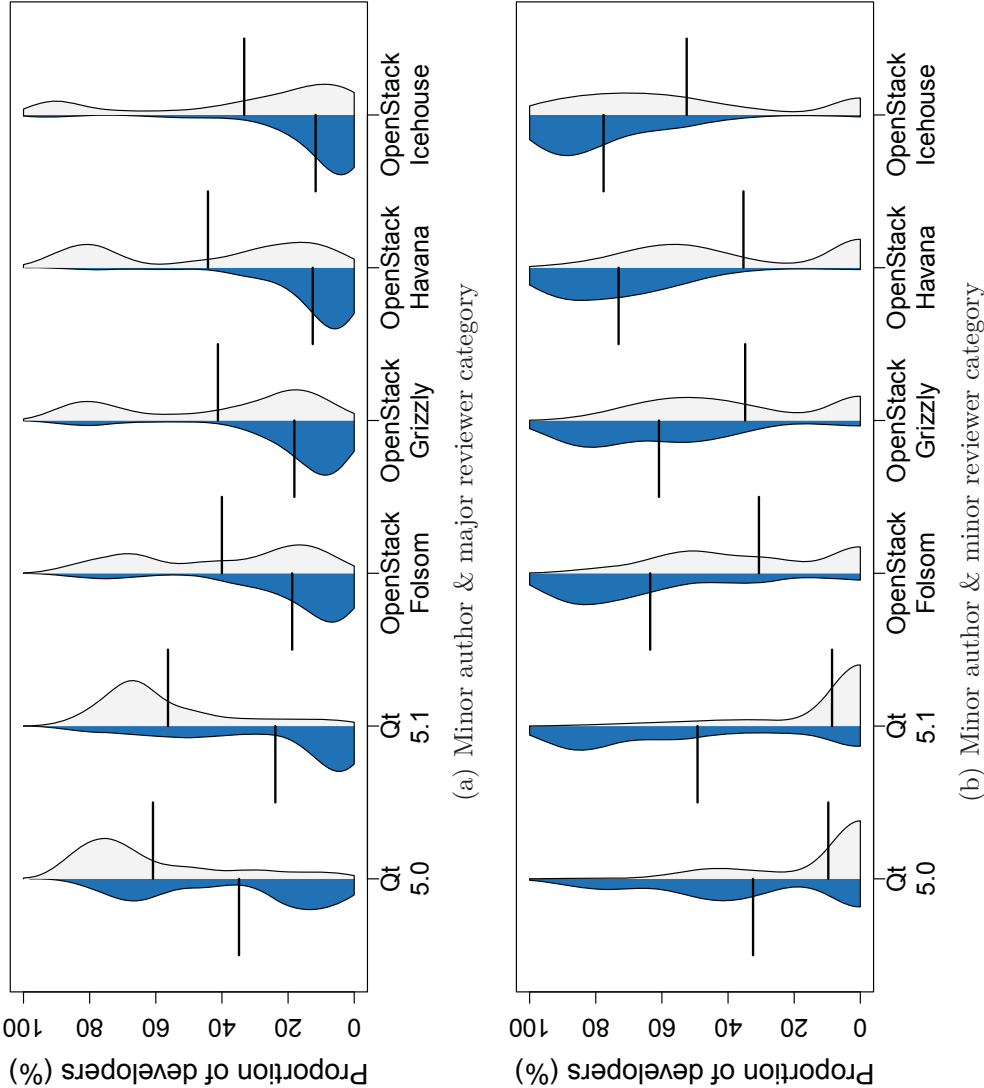


Figure 4.4. The distribution of developers of defective (blue) and clean (gray) modules when using RSO_{Even} . The horizontal lines indicates the median of distributions.

Table 4.4. Results of one-tailed Mann-Whitney U tests for the developers in defective (D) and clean (C) models.

Release	RSO _{Even}				RSO _{Proportional}			
	Minor author & minor reviewer	Minor author & major reviewer	Major author & minor reviewer	Major author & major reviewer	Minor author & minor reviewer	Minor author & major reviewer	Major author & minor reviewer	Major author & major reviewer
5.0	D>C*** 0.444 (M)	D<C*** 0.556 (L)	D>C* 0.183 (S)	o	D>C*** 0.435 (M)	D<C*** 0.524 (L)	D>C* 0.158 (S)	o
5.1	D>C*** 0.633 (L)	D<C*** 0.649 (L)	D>C*** 0.343 (M)	D<C*** 0.462 (M)	D>C*** 0.663 (L)	D<C*** 0.685 (L)	D>C*** 0.279 (S)	D<C*** 0.445 (M)
Folsom	D>C*** 0.622 (L)	D<C*** 0.532 (L)	D<C* 0.170 (S)	D<C*** 0.461 (M)	D>C*** 0.623 (L)	D<C*** 0.512 (L)	o	D<C*** 0.528 (L)
Grizzly	D>C*** 0.518 (L)	D<C*** 0.562 (L)	o	D<C*** 0.351 (M)	D>C*** 0.476 (L)	D<C*** 0.524 (L)	D>C* 0.128 (N)	D<C*** 0.347 (M)
Havana	D>C*** 0.708 (L)	D<C*** 0.698 (L)	o	D<C*** 0.437 (M)	D>C*** 0.664 (L)	D<C*** 0.637 (L)	D>C** 0.142 (N)	D<C*** 0.548 (L)
Icehouse	D>C*** 0.486 (L)	D<C*** 0.468 (M)	o	D<C*** 0.363 (M)	D>C*** 0.562 (L)	D<C*** 0.568 (L)	D>C*** 0.186 (S)	D<C*** 0.483 (L)

Statistical significance: $op \geq 0.05$, $*p < 0.05$, $**p < 0.01$, $***p < 0.001$

Effect size: (L: Large) Cliff's $\delta \geq 0.474$, (M: Medium) $0.33 \leq \delta < 0.474$, (S: Small) $0.147 \leq \delta < 0.33$, (N: Negligible) $\delta < 0.147$

is significantly smaller than that of clean modules, with medium effect sizes for the Qt 5.1 ($0.445 \leq \delta \leq 0.462$) and all OpenStack datasets ($0.347 \leq \delta \leq 0.528$).

Summary: Many minor authors are major reviewers who actually make large contributions to the evolution of modules by reviewing code changes. Code review activity can be used to refine traditional code ownership heuristics to more accurately identify the defect-prone modules.

(DA2) Defect Model Analysis

For our defect model analysis, we present the results of our model construction and analysis when using RSO_{Even} to estimate review-specific and review-aware ownership. We complement our results with the models where the review-specific and review-aware ownership are estimated using $RSO_{\text{Proportional}}$ in Appendix B.2 (see Table B.1).

(MC-1) Correlation & Redundancy Analysis. Figure 4.5 shows an example of the hierarchical clustering of explanatory variables. We find that the top TCO, the number of contributors, authors, and reviewers are often highly correlated. We select the number of contributors as the representative for these variables, since the number of contributors is simpler to calculate and can capture the number of both authors and reviewers. We also find that the proportions of minor author & major reviewer and minor author & minor reviewer are highly correlated. We remove the proportion of minor author & major reviewer, since we want to revisit the relationship between the proportion of minor contributors (i.e., minor author & minor reviewer) and defect-proneness. For the sake of completeness, we analyze models that use the proportion of minor author & major reviewer instead of the proportion of minor author & minor reviewer (see Table B.3 in Appendix B.2). We found that the proportion of minor author & major reviewer had no discernible impact on model performance.

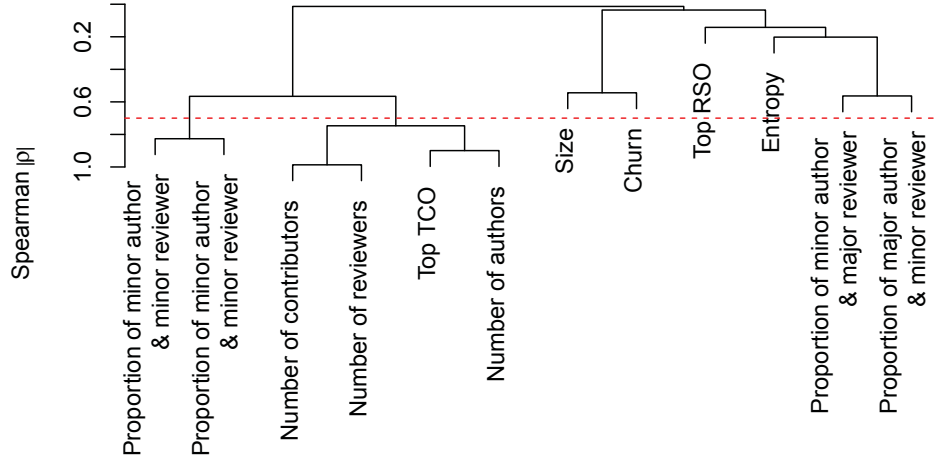


Figure 4.5. Hierarchical clustering of variables according to Spearman’s $|\rho|$ in the Qt 5.0 dataset. The dashed line indicates the high correlation threshold (i.e., Spearman’s $|\rho| = 0.7$).

After removing the highly correlated variables, we repeat the variables clustering analysis and find that the number of contributors and the proportions of minor author & minor reviewer are highly correlated. We opt to remove the number of contributors, since the proportion of minor author & minor reviewer metrics are already controlled by the number of contributors.

(MC-2) Degree of Freedom Allocation. We allocate the budgeted degrees of freedom to the surviving explanatory variables based on their potential for sharing a nonlinear relationship with the response variable. For example, Figure 4.6 shows the potential for nonlinearity in the relationship between explanatory variables and the response variable in the Android dataset. We allocate: (1) three degrees of freedom to size and the proportion of minor author & minor reviewer, and (2) one degree of freedom to the remaining variables. We repeat the same process for the other studied datasets.

We then build our logistic regression models to fit our patch data using the surviving explanatory variables with the allocated degrees of freedom. Table 4.5 shows that the number of degrees of freedom that we spent to fit our models did not exceed the budgeted degrees of freedom.

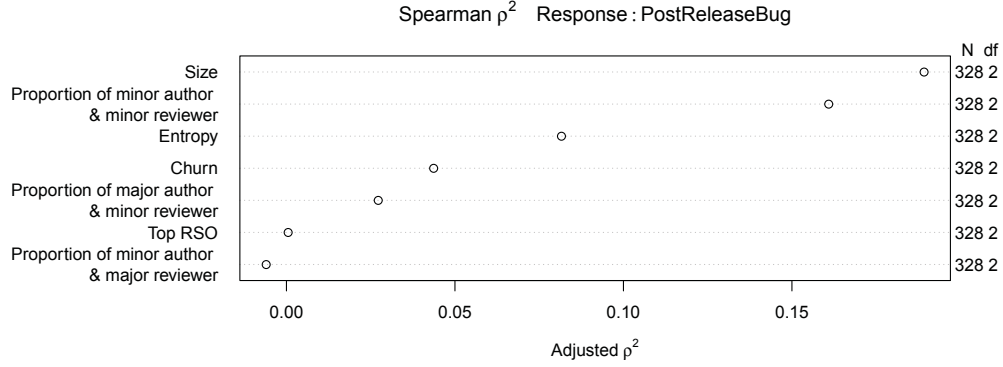


Figure 4.6. Dotplot of the Spearman multiple ρ^2 of each explanatory variable and defect-proneness in the Qt 5.0 dataset.

Model Analysis Results

Table 4.5 shows that our defect models achieve an AUC of between 0.81 (Qt 5.0 and OpenStack Icehouse) and 0.89 (OpenStack Havana). The AUC optimism is also relatively small ranging from 0.01 (OpenStack Havana and Icehouse) to 0.03 (OpenStack Folsom). We obtain similar model statistics when using $\text{RSO}_{\text{Proportional}}$ (see Appendix B.2). The AUC values of the $\text{RSO}_{\text{Proportional}}$ models range from 0.81 (Qt 5.0) to 0.87 (Qt 5.1 and OpenStack Havana), with an AUC optimism of 0.01-0.03. Since RSO_{Even} models use a simpler approximation of review-specific and review-aware ownership, we report our observations for the analysis results of the RSO_{Even} models below.

Observation 6 – The proportion of developers in the minor author & minor reviewer category shares a strong relationship with post-release defect proneness. Table 4.5 shows that the proportion of minor author & minor reviewer contributes a significant amount of explanatory power to the fit of our models (shown in the overall Wald χ^2 column). The OpenStack Folsom model is the only one where the proportion of minor author & minor reviewer did not contribute a significant amount of explanatory power. Furthermore, we observe that the proportion of minor author & minor reviewer accounts for most of the explanatory power in the Qt 5.0, Qt 5.1, and OpenStack Havana models. This result indicates

Table 4.5. Statistics of defect models where review-specific and review-aware ownership are estimated using RSO_{Even} . The explanatory power (χ^2) of each variable is shown in a proportion to Wald χ^2 of the model.

	Qt 5.0		Qt 5.1		OpenStack Folsom		OpenStack Grizzly		OpenStack Havana		OpenStack Icehouse	
AUC	AUC optimum Wald χ^2	0.81	0.86	0.89	0.83	0.88	0.81	0.81	0.81	0.81	0.81	0.81
		0.02	0.02	0.03	0.02	0.01	0.01	0.01	0.01	0.01	0.01	0.01
Size	D.F. χ^2	48***	81***	57***	68***	114***	93***	93***	93***	93***	93***	93***
		Overall	Nonlinear	Overall	Nonlinear	Overall	Nonlinear	Overall	Nonlinear	Overall	Nonlinear	Overall
Churn	D.F. χ^2	2	1	2	1	2	1	2	1	2	1	2
		13%*	0%°	24%***	10%*	13%*	0%°	4%°	0%°	27%***	1%°	1%°
Entropy	D.F. χ^2	1	—	1	—	1	—	1	—	1	—	1
		4%°	—	0%°	—	1%°	—	15%***	—	12%**	—	—
Top TCO	D.F. χ^2	1	—	1	—	1	—	1	—	1	—	1
		1%°	—	6%°	—	2%°	—	0%°	—	2%°	—	—
Top RSO_{Even}	D.F. χ^2	†	†	1	4	3	2	1	1	†	†	†
		1	—	5%°	13%°	2%°	1%°	0%°	—	1	—	—
Major author & major reviewer	D.F. χ^2	0%°	—	2%°	6%*	—	0%°	—	—	4%°	—	—
		1	—	2	2	1	1	—	—	1	—	—
Minor author & minor reviewer	D.F. χ^2	11%*	8%*	14%*	6%°	2%°	3%*	—	—	2%°	—	—
		2	4	4	3	2	1	4	3	2	1	1
Major author & minor reviewer	D.F. χ^2	46%***	4%°	10%°	9%°	11%*	6%*	32%***	6%°	13%**	1%°	1%°
		1	—	1	—	1	—	1	—	1	—	—
Major reviewer	D.F. χ^2	6%°	—	5%°	—	3%°	1%°	—	—	3%°	—	—
		1	—	1	—	1	—	1	—	1	—	—

†: Discarded during variable clustering analysis ($|\rho| \geq 0.7$)

The number of contributors, authors, reviewers, and the proportion of minor author & major reviewer are also discarded during variable clustering analysis.

—: Nonlinear degrees of freedom not allocated.

Statistical significance of explanatory power according to Wald χ^2 likelihood ratio test: $op \geq 0.05$; * $p < 0.05$; ** $p < 0.01$; *** $p < 0.001$

that the rate of contributors who lack both authorship and reviewing expertise in a module shares a strong relationship with the post-release quality.

Table 4.5 also shows that the additional degrees of freedom that we allocate to the proportion of minor author & minor reviewer did not contribute a significant amount of explanatory power to the fit of our models (shown in the nonlinear Wald χ^2 column). This result indicates that there was no significant benefit in spending additional degrees of freedom on this metric—the relationship is primarily log-linear.

Observation 7 – Modules with a higher rate of developers in the minor author & minor reviewer category are more likely to be defect-prone. Figure 4.7 shows that there is an increasing trend in the probability that a typical module will have post-release defects as the proportion of minor author & minor reviewer increases. The narrow breadth of the confidence interval (gray area) indicates that there is sufficient data to support the curve. Moreover, Figure 4.7 shows that the probability of having post-release defects rapidly increases when the proportion of minor author & minor reviewer increases beyond 0.5 in the Qt 5.0, Qt 5.1, OpenStack Grizzly and Havana models.

In addition, we check whether the roles of developers (i.e., core and non-core developers) have an impact on our results. To do so, we add the proportion of core developers who are involved in a module as another control metrics and repeat the model construction and analysis again. We find that the proportion of core developers did not contribute a significant amount of explanatory in four of the six defect models (i.e., the Qt 5.0, 5.1 and OpenStack Folsom, and Havana models), while the proportion of core developers accounts for a small explanatory power to the OpenStack Grizzly and Icehouse models (See Table B.2 in Appendix B.2). On the other hand, the proportion of minor author & minor reviewer still contributes a large explanatory power to our models, even though we control for the roles of developers. This result suggests that the roles of developers did not have an impact on software quality as much as the expertise of developers has.

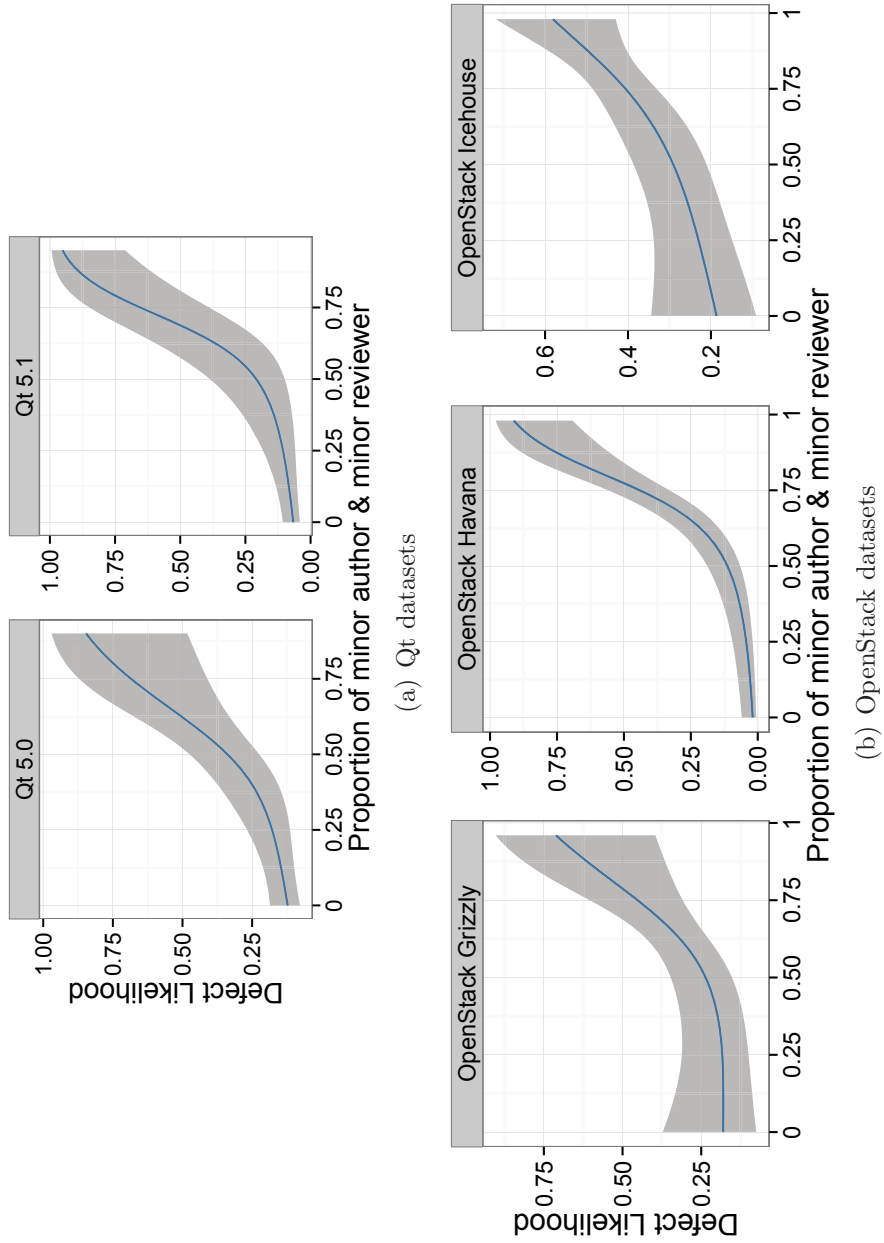


Figure 4.7. The estimated probability in a typical module for the proportion of developers in the minor author & minor reviewer category ranging. The gray area shows the 95% confidence interval.

We also analyze the defect models that use the proportion of minor author & major reviewer instead of the proportion of minor author & minor reviewer. We find that the proportion of minor author & major reviewer contributes a significant amount of explanatory power to the Qt 5.0, Qt 5.1, OpenStack Havana and Icehouse models (see Table B.3 in Appendix B.2). Furthermore, we observe that there is an inverse relationship between the proportion of minor author & major reviewer and the probability that a typical module will have post-release defects in the Qt 5.0, Qt 5.1, OpenStack Havana and Icehouse models (See Figure B.3 in Appendix B.2). This result indicates that the larger the proportion of developers in the minor author & major reviewer category, the lower the likelihood of a module having post-release defects.

Summary: Even when we control for several confounding factors, the proportion of developers in the minor author & minor reviewer category shares a strong relationship with defect-proneness. Indeed, modules with a larger proportion of developers without authorship or reviewing expertise are more likely to be defect-prone.

4.5. Practical Suggestions

In this section, we discuss the broader implications of our observations by offering the following suggestions:

(1) Code review activity should be included in future approximations of code ownership.

Observations 1 and 2 show that apart from the developers who author code changes to a module, there are major contributors who only contribute to that module by reviewing code changes. Furthermore, observation 3 shows that many developers who were identified as minor contributors by

traditional code ownership heuristics are actually major contributors when their code reviewer contributions are taken into consideration.

- (2) Teams should apply additional scrutiny to module contributions from developers who have neither authored nor reviewed many code changes to that module in the past.**

Observation 5 shows that modules with post-release defects tend to have a larger proportion of minor authors who are also minor reviewers than modules without post-release defects do. Furthermore, observations 6 and 7 show that the proportion of developers in the minor author & minor reviewer category shares a strong increasing relationship with the likelihood of having post-release defects in a module, even when we control for several confounding factors.

- (3) A module with many developers who have not authored many code changes should not be considered risky if those developers have reviewed many of the code changes to that module.**

While observations 6 and 7 confirm the findings of prior work [21, 93], i.e., the number of minor contributors shares a relationship with defect-proneness, we find that the proportion of developers in another category of minor authors (i.e., minor author & major reviewer) shares an inverse relationship with the defect-proneness (see RQ3). Indeed, observation 4 shows that modules without post-release defects tend to have a larger proportion of developers in minor author & major reviewer than modules with post-release defects do.

4.6. Threats to Validity

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

4.6.1. External validity

We focus our study on two open source systems, due to the low number of systems that satisfied our eligibility criteria for analysis (see Section 4.3.1). Thus, our results may not generalize to all software systems. However, the goal of this study is not to build a theory that applies to all systems, but rather to show that code review activity can have an impact on code ownership approximations. Our results suggest that code review activity should be considered in future studies of code ownership. Nonetheless, additional replication studies are needed to generalize our results.

4.6.2. Construct validity

Our analysis is focused on the code review activity that is recorded in the code review tools of the studied systems, i.e., Gerrit. However, there are likely cases where developers perform code reviews through other communication media, such as through in-person discussions [16], a group IRC [108], or a mailing list [45, 101]. Unfortunately, there are no explicit links of code changes to those communication threads, and recovering these links is a non-trivial research problem [7, 19]. Nevertheless, we perform our study on modules where every code change could be linked to the reviews in Gerrit, which should capture the majority of the review discussion in the studied systems.

Since we identify defect-fixing changes using keyword-based approach, there are likely cases that our defect data is inaccurate [18]. To evaluate this, we measure the accuracy of our approach by manually examining samples of defect-fixing changes. For each studied system, we randomly select 50 code changes that are flagged as bug-fixing changes by our keyword-based approach. We find that 88%-92% of the sampled code changes are correctly identified.

4.6.3. Internal validity

We identify developers who post at least one reviewer comment as reviewers of a module, although some of the identified reviewers may only leave superficial or unrelated reviewer comments [26, 88]. We attempt to mitigate this risk with our $\text{RSO}_{\text{Proportional}}$ heuristic, which allocates less ownership value for reviewers who provide less feedback. We find that the results of our study using $\text{RSO}_{\text{Proportional}}$ heuristic are similar to the results using the simpler, RSO_{Even} heuristic, which allocates an even share of the ownership value to every reviewer of a code change. This suggests that the noise of reviewer contributions is not heavily biasing our results.

In this study, we opt to measure RSO and TCO values separately. Even when they are combined in our the review-aware ownership heuristics, the values are plotted on orthogonal axes rather than summed. Summation of RSO and TCO values may have a different association with software quality. However, since reviewing and authoring are different activities, a naive summation may not be desired. Additional work is needed to investigate appropriate means of computing a generic expertise metric.

There may still be cases where developers who make many contributions by either authoring or reviewing code changes are not the actual owner of the modules. Unfortunately, ground truth data is not available for us to validate against. Nevertheless, we use a list of core developers that is available in the documentation of the studied systems to validate our heuristics. Our results also show that many of these core developers can only be module owners if their code reviewer contributions are considered in code ownership heuristics.

4.7. Summary

Code ownership heuristics have been used in many studies for identifying the developers who are responsible for maintaining modules. These heuristics are traditionally computed using code authorship contributions, and many studies discover an association between code authorship and defect-proneness. However, developers can also make important contributions to modules by critiquing code changes during the code review process.

In this chapter, we extend the traditional code ownership heuristics to be: (1) review-specific, i.e., a code ownership approximation that is derived solely using reviewer contributions and (2) review-aware, i.e., a code ownership approximation that is derived using both authorship and reviewer contributions. Through a case study of six releases of the large Qt and OpenStack open source systems, we make the following observations:

- 67%-86% of developers only contribute to a module by reviewing code changes. 18%-50% of these review-only contributors are documented core developers of the studied systems (Observations 1 and 2).
- 13%-58% of developers who are flagged as minor contributors by traditional code ownership heuristics are actually major contributors when their code review activity is considered (Observation 3).
- When traditional code ownership heuristics are refined by code review activity, we find that modules without post-release defects tend to have a higher rate of developers in the minor author & major reviewer category, but a lower rate of developers in the minor author & minor reviewer category than modules with post-release defects do (Observations 4 and 5).
- Even when we control for several factors that are known to have an impact on software quality, the proportion of developers in the minor author &

minor reviewer category shares a strong, increasing relationship with the likelihood of having post-release defects in a module (Observations [6](#) and [7](#)).

Selecting Appropriate Reviewers

An earlier version of the work in this chapter appears in the Proceedings of the 22nd International Conference on Software Analysis, Evolution, and Reengineering (SAnER) [124].

In the previous chapter, our results show that the more reviewing experts the module has, the less likely that the module will have post-release defects. This finding suggests that to mitigate the risk of having defects in the future, a new patch should be examined by reviewers who have expertise on the modules impacted by that patch. However, manually finding reviewers who have expertise can be a tedious task for the patch author, especially for the new comer author. To better understand this, we first perform an empirical study on the difficulty of selecting reviewers and its impact on MCR timeliness. The results of our case studies demonstrate that patches where the patch author cannot find appropriate reviewers tend to be in MCR tools longer than patches without such a problem. Furthermore, to help developers find appropriate reviewers, we propose REVFINDER, a file location-based reviewer recommendation approach. The evaluation results also demonstrate that our REVFINDER can accurately recom-

mend reviewers who should examine a new patch. Motivated by this study, recent research has begun to develop reviewer recommendation approaches in order to support MCR processes at Microsoft [135], on GitHub [56, 72, 134], and within several open source projects [132].

5.1. Introduction

To effectively evaluate a new patch, a patch author should find appropriate reviewers who have a deep understanding of the related source code to well examine the patch and identify defects [5]. Indeed, Tao *et al.* report that determining the risk of a new patch is difficult for developers at Microsoft if the affected modules are beyond the developers' knowledge [117]. Rigby and Storey find that developers tend to ignore patches that do not meet their expertise [101].

Similar to other collaborative processes [3, 113, 126], manually finding relevant experts for reviewing a new patch can be time-consuming, especially at the distributed software development teams. For example, a patch author who newly joined the development team may know a small set of developers. To find appropriate reviewers, the patch author needs to ask other developers for recommendations. Such practices may lead to a longer time that is spent on MCR processes.

In this chapter, we first set out to investigate the reviewer selection problem in a patch, i.e., an author cannot initially select appropriate reviewers for a new patch. Furthermore, as a means of reducing the time spent on selecting appropriate reviewers, we propose REVFINDER, a file location-based reviewer recommendation approach. We leverage the similarities between the path of a newly changed file and the paths of previously reviewed files to recommend an appropriate reviewer. The intuition behind REVFINDER is that *files that are located in similar file paths would be managed and reviewed by a similar set of experienced reviewers.* We also combine several string comparison techniques to increase the accuracy

and the ranking performance of REVFinder. Finally, we compare the performance of REVFinder with a baseline approach or REVIEWBOT, i.e., which is a previously proposed approach for MCR process at VMware [9].

Through a manual examination of 7,597 comments and an empirical evaluation of 42,045 patches spread across Android, Qt, OpenStack, and LibreOffice open source systems, we address the following research questions:

(RQ1) Does the reviewer assignment problem impact the time that is spent on MCR processes?

Motivation. Recent research points out that selecting an appropriate reviewer can be an effort-intensive task for developers at VMware [9]. Yet, it is still unclear whether a patch author is suffering from selecting appropriate reviewers and whether this problem is associated with the time that is spent on MCR processes. Hence, we set out to empirically investigate the difference in the time that is spent on MCR processes between patches with the reviewer selection problem and those without such a problem.

Results. We find that 4%-30% of representative patch samples have the reviewer selection problem. Moreover, these patches with the reviewer selection problem typically take 6 to 21 days in the MCR processes longer than patches without the reviewer selection problem.

(RQ2) Does RevFinder accurately recommend reviewers?

Motivation. Better performing reviewer recommendation approach is of great value as it enables a better selecting decision for developers. Moreover, the higher ranks the correct reviewers that the recommendation approach can recommend, the more effective the recommendation approach is [43]. In other words, recommending correct reviewers in the top ranks could help a patch author in selecting the appropriate reviewers as well as avoid involving unrelated reviewers. Therefore, we set out to evaluate REVFinder in terms of ranking performance and accuracy of recommenda-

tion. To do so, we measure Mean Reciprocal Rank (MRR) which calculates an average of reciprocal ranks of correct reviewers in a recommendation list. We also measure the Top- k accuracy which calculates the percentage of patches that will receive a correct recommendation from REVFINDER within k recommendation.

Results. REVFINDER can recommend correct reviewers with a median rank of 4. Moreover, the overall ranking of REVFINDER is 3 times better than that of the baseline approach, indicating that REVFINDER provides a good ranking of correct reviewers. REVFINDER can correctly recommend reviewers with a Top-5 accuracy of 41%-79%, which is more accurate than REVIEWBOT. These results indicate that leveraging the similarities between the path of a newly changed file and the paths of previously reviewed files can help teams to accurately recommend reviewers.

Our results lead us to conclude that REVFINDER can help developers find appropriate reviewers, which is likely to speed up the overall code review process. Furthermore, the main contributions of this chapter are:

- An exploratory study of the impact of reviewer assignment on the time that is spent on MCR processes.
- REVFINDER, a file location-based reviewers recommendation approach, with promising evaluation results to automatically suggest appropriate reviewers for MCR.
- A rich data set of reviews data in order to encourage future research in the area of reviewer recommendation.¹

¹<http://github.com/patanamon/revfinder>

5.1.1. Chapter Organization

The remainder of the chapter is organized as follows. Section 5.2 describes our case study design. Section 5.3 presents an exploratory study of the reviewer selection problem in MCR processes. Section 5.4 describes REVFinder. Section 5.5 describes an empirical evaluation of REVFinder, while Section 5.6 presents the results of our empirical evaluation. Section 5.7 discusses the performance and applicability of REVFinder, and addresses the threats to validity. Finally, Section 5.9 summarizes this work.

5.2. Case Study Design

In this section, we describe our studied systems and present the data collection approach that we used for an exploratory study and empirical evaluation.

5.2.1. Studied Systems

In order to address our research questions, we perform an empirical study of large software systems that actively use MCR for the code review process, i.e., examine and discuss software changes through a code review tool. Hence, we select to study the MCR processes of Android, Qt, OpenStack and LibreOffice open source systems. These systems are using Gerrit as for their code review processes. Table 5.1 shows a statistical summary of the studied systems.

The Android open source system² is a mobile operating system developed by Google. Qt³ is a cross-platform application and UI framework developed by Digia Plc. OpenStack⁴ is a free and open-source software cloud computing software platform supported by many well-known companies e.g., IBM, VMware, and NEC. LibreOffice⁵ is a free and open source office suite.

²<https://source.android.com/>

³<http://qt-project.org/>

⁴<http://www.OpenStack.org/>

⁵<http://www.libreoffice.org/>

Table 5.1. A summary of the dataset for each studied system.

	Android	OpenStack	Qt	LibreOffice
Studied Period	10/2008 - 01/2012	07/2011 - 05/2012	05/2011 - 05/2012	03/2012 - 06/2014
# Selected Reviews	5,126	6,586	23,810	6,523
# Reviewers	94	82	202	63
# Files	26,840	16,953	78,401	35,273
Avg. Reviewers per Review	1.06	1.44	1.07	1.01
Avg. Files per Review	8.26	6.04	10.64	11.14

5.2.2. Data Collection

To obtain review data of the studied systems, we began with the review datasets of Android, Qt, and OpenStack systems that are provided by Hamasaki *et al.* [46]. We also expand the review datasets to include review data of the LibreOffice system using the same collection technique as the prior work.

Once we obtained the review data, we exclude the patches that do not satisfy the following criteria:

1. The review of a patch must be closed, i.e., the review status is “*Merged*” or “*Abandoned*”.
2. A patch must contain at least one changed files.

For criterion 1, we filter out patches that are still open in order to ensure that the patches have been examined. Moreover, we cannot build our ground-truth data since these open-reviewing patches may not yet have reviewers. For criterion 2, we filter out patches that do not contain any changed files because those patches may be related to VCS bookkeeping (like branch merging) for other patches that have already been examined and integrated. Such patches generally do not need an expert since the earlier patches have already been reviewed. Table 5.1 shows the statistical summary for each studied system.

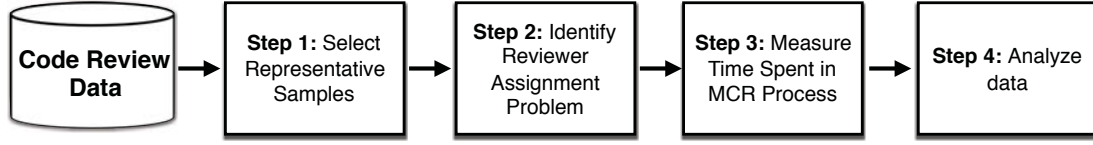


Figure 5.1. An overview of our approach for RQ1.

5.3. An Exploratory Study of the Reviewer Selection Problem in MCR processes

In this section, we describe our approach and present results for our RQ1. More specifically, we present our exploratory study on the difficulty of selecting appropriate reviewers in MCR processes.

(RQ1) Does the reviewer assignment problem impact the time that is spent on MCR processes?

5.3.1. Approach

To address RQ1, we manually identify the reviewer selection problem on representative patch samples. Then, we analyze the difference in the time that is spent on MCR processes between patches with the reviewer selection problem and patches that do not have such a problem. Figure 5.1 provides an overview of our approach for RQ1. We describe each step of our approach below.

Step 1: Selecting Representative Samples

Since the full sets of patches are too large to manually examine entirely, we randomly select a statistically representative sample for our analysis. To obtain a representative sample with a confidence level of 95%, we determine a sample size using the following equation:

$$s = \frac{z^2 p(1-p)}{c^2} \quad (5.1)$$

where p is the proportion that we want to estimate, $z = 1.96$ to achieve a 95% confidence level, and $c = 0.05$ for a 10% bound of the actual proportion [63, 67]. Since we did not know the proportion in advance, we use $p = 0.5$. We further correct for the finite population of reviews P using the following equation in order to obtain our sample for manual analysis.

$$ss = \frac{s}{1 + \frac{s-1}{P}} \quad (5.2)$$

Step 2: Identify the Reviewer Selection Problem

We use the review discussion history to determine whether a patch has the reviewer selection problem. Since messages in the review discussion history are written in natural language, we manually identify a patch that has the reviewer selection problem instead of using an automatic approach in order to yield an accurate result. We identify patches as *having a reviewer selection problem* if the discussions of the patches have an issue like who should review the patch or recommending reviewers. For example, Figure 5.2(a) shows that the patch author of the Android review ID 18767 posted a message into discussion threads for a reviewer recommendation.⁶ Figure 5.2(b) shows that Chris Adam was invited to examine the Qt review ID 13006, however, he did not have a confidence to review the patch.⁷ Then, he invited Aaron Kennedy as another reviewer of the Qt review ID 13006. We identify the patches of these Android review ID 18767 and Qt review ID 13006 as patches that have a reviewer selection problem. Patches without such discussion issues are identified as *patches that do not have a reviewer selection problem*. This step was conducted by the author of this thesis and verified by the second author of our paper [124].

⁶<https://android-review.googlesource.com/#/c/18767/>

⁷<https://codereview.qt-project.org/#/c/13006/>

Step 3: Measure Time Spent in MCR processes

Once we manually identify patches that have the reviewer selection problem in the representative patch samples, we measure the time that is spent on MCR processes of the review samples. To calculate the time that is spent on MCR processes of the studied patches, we measure the time difference in days from the time of a patch submission to the review decision is made, i.e., till when the reviewers provide a score of +2 or -2. Since prior studies find that time spent on MCR processes is often influenced by patch size [40, 131], we control this factor by normalizing time that is spent on MCR processes by the patch size.

Step 4: Analyze data

To determine whether the reviewer selection problem is associated with the time that is spent in MCR processes, we compare the distributions of the time that is spent on MCR processes between patches with the reviewer selection problem and patches without the reviewer selection problem using beanplots [59]. Beanplots are boxplots in which the vertical curves summarize the distributions of different datasets. The horizontal lines indicate median of the distributions. Furthermore, we use a statistical test to detect a statistically significant level of the differences. We use the Mann-Whitney's U test ($\alpha = 0.05$) since we observe that the distributions of the time spent in the MCR processes of the studied patches do not follow a normal distribution.

5.3.2. Results

We now present the results and our empirical observations from our analysis. Table 5.2 shows the number of representative samples for our manual examination which are 357 Android patches, 378 Qt patches, 363 OpenStack patches, and 363 LibreOffice patches. In total, we manually examine 7,597 comments of 1,461 review samples.

Table 5.2. The numbers of statistically representative samples for each studied systems and the proportion of patches with the reviewer selection problem with a 95% confidence level and a $\pm 5\%$ bound.

	Android	Qt	OpenStack	LibreOffice
Number of Representative Patch Sample	357	378	363	363
Patches with the reviewer selection problem	10%	30%	5%	4%

Observation 8 – 4%-30% of reviews have a reviewer selection problem. Table 5.2 shows the results of our manual identification of patches that have the reviewer selection problem. We find that 10%, 30%, 5%, and 4% of the patches have the reviewer selection problem in Android, OpenStack, Qt, and LibreOffice systems, respectively. We observe that Qt has the highest proportion of patches with the reviewer selection problem. This may in part be due to the size of the community and software system (i.e., the amount of reviews, reviewers, and files), suggesting that the larger the system is, the more likely that the patch author have a difficult of selecting appropriate reviewers.

During our manual examination, we find that patch authors indeed have difficulties selecting appropriate reviewers. For example, a Qt developer suggested a patch author who initially did not assign any developers for a review: “*You might want to add some approvers to the reviewers list if you want it reviewed/approved*”.⁸ Additionally, we find that patch authors often ask questions about selecting the appropriate reviewers. For example, a Qt developer post a message into the general discussion thread for a reviewer recommendation: “*Feel free to add reviewers, I am not sure who needs to review this...*”.⁹ An Android developer also asked team members for a review: “*Can you please add appropriate reviewers for this change?*”.¹⁰ Moreover, a Qt developer pointed out that assigning

⁸Qt review ID 16803, <https://codereview.qt-project.org/#/c/16803>

⁹Qt review ID 40477, <https://codereview.qt-project.org/#/c/40477>

¹⁰Android review ID 18767, <https://android-review.googlesource.com/#/c/18767/>

reviewers can speed up the code review process: “for the future, it speeds things up often if you add reviewers for your changes :)”.¹¹

Observation 9 – Patches with the reviewer selection problem tend to take longer time than patches without the problem. Figure 5.3 shows that patches with the reviewer selection problem take 21, 9, 13, and 6 days (at median) in the MCR process of Android, OpenStack, Qt, and LibreOffice, respectively. On the other hand, patches without the reviewer selection problem tend to complete within one day. Mann-Whitney U tests confirm that the differences are statistically significant (p -value < 0.001 for Android, and Qt, OpenStack, and p -value < 0.01 for LibreOffice). This result suggests that the reviewer selection problem may slow down the code review processes.

Summary: We find that 4%-30% of patches have a reviewer selection problem. Moreover, these patches typically take 6-21 days in the MCR processes and tend to take longer than patches without the reviewer selection problem. Indeed, some patch authors have a difficulty of selecting appropriate reviewers, suggesting that a reviewer recommendation tool may need to help the patch author and speed up the processes.

5.4. RevFinder: A File Location-Based Reviewer Recommendation Approach

We now present REVFINDER. In this section, we first provide an overview of REVFINDER. Then, we describe the reviewer ranking algorithm of REVFINDER, the string comparison techniques that we use to measure the similarities between the path of a newly changed file and the paths of previously reviewed files, and the combination techniques which combine the lists of reviewer candidates that are produced using different string comparison techniques.

¹¹Qt review ID 14251, <https://codereview.qt-project.org/#/c/14251>

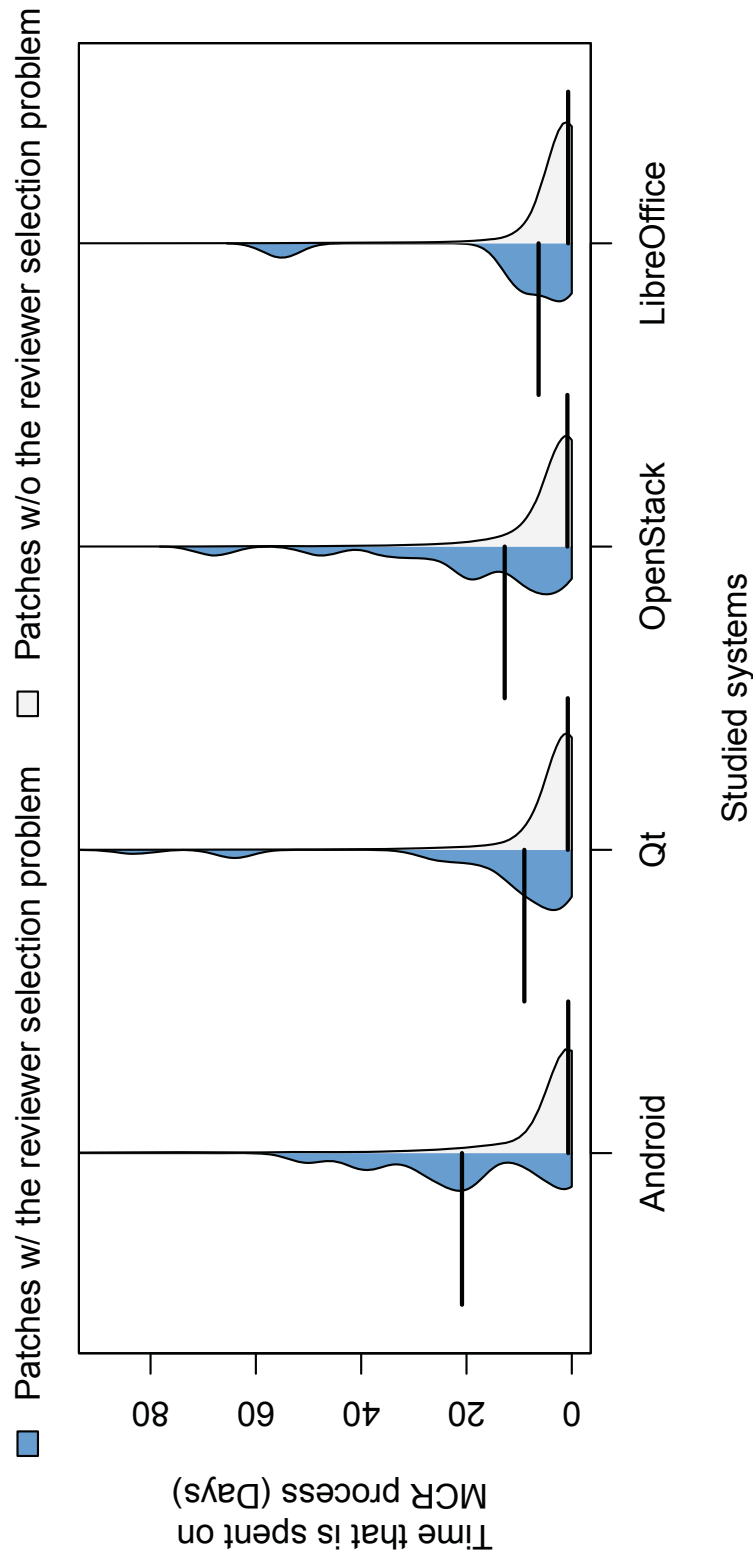


Figure 5.3. A comparison of the time that is spent on MCR processes (days) of reviews with and without the reviewer selection problem. The horizontal lines indicate the average (median) review time (days).

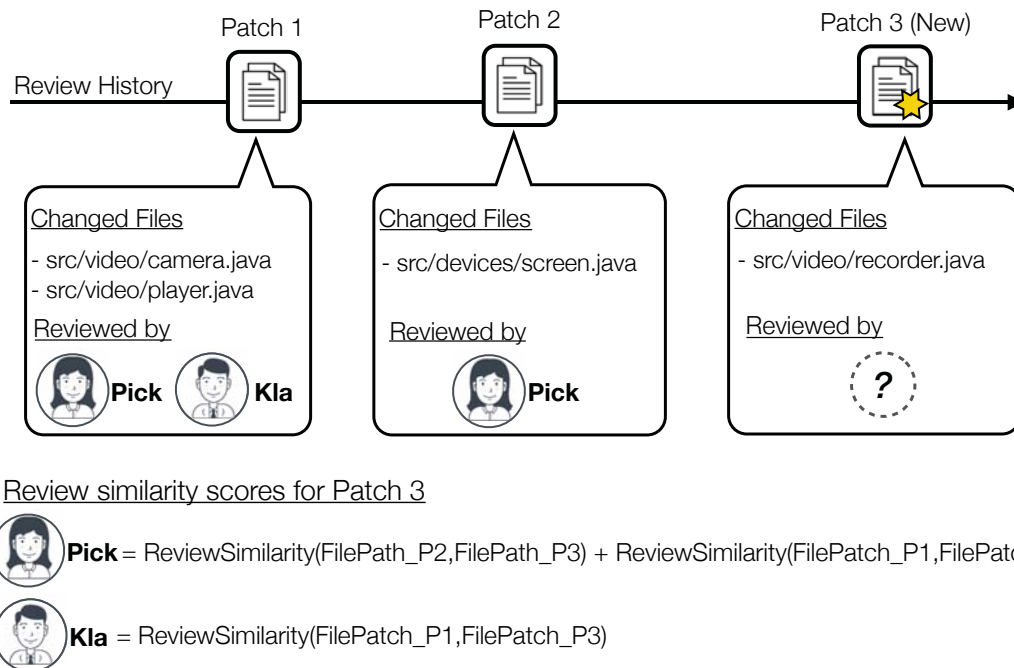


Figure 5.4. A calculation example of the Reviewer Ranking Algorithm.

5.4.1. An Overview of RevFinder

REVFINDER aims to recommend reviewers who have reviewing experience within a similar functionality of a new patch. Therefore, we leverage the similarities between the path of a newly changed file and the paths of previously reviewed files to recommend reviewers. Figure 5.4 provides an example of REVFinder where Patch 3 is a new patch that needs a reviewer, while Patches 1 and 2 are prior patches that have been reviewed by Pick and Kla. To recommend a reviewer, REVFinder first calculates the review similarity between Patch 3 and the prior patches (Patches 1 and 2) by comparing the file paths of the changed files between the new patch and the prior patches. Then, REVFinder assigns these patch similarity scores to the developers who reviewed the prior patches. For this example, REVFinder computes patch similarity scores, i.e., $\text{ReviewSimilarity}(\text{FilePath_P2}, \text{FilePath_P3})$ and $\text{ReviewSimilarity}(\text{FilePatch_P1}, \text{FilePatch_P3})$.

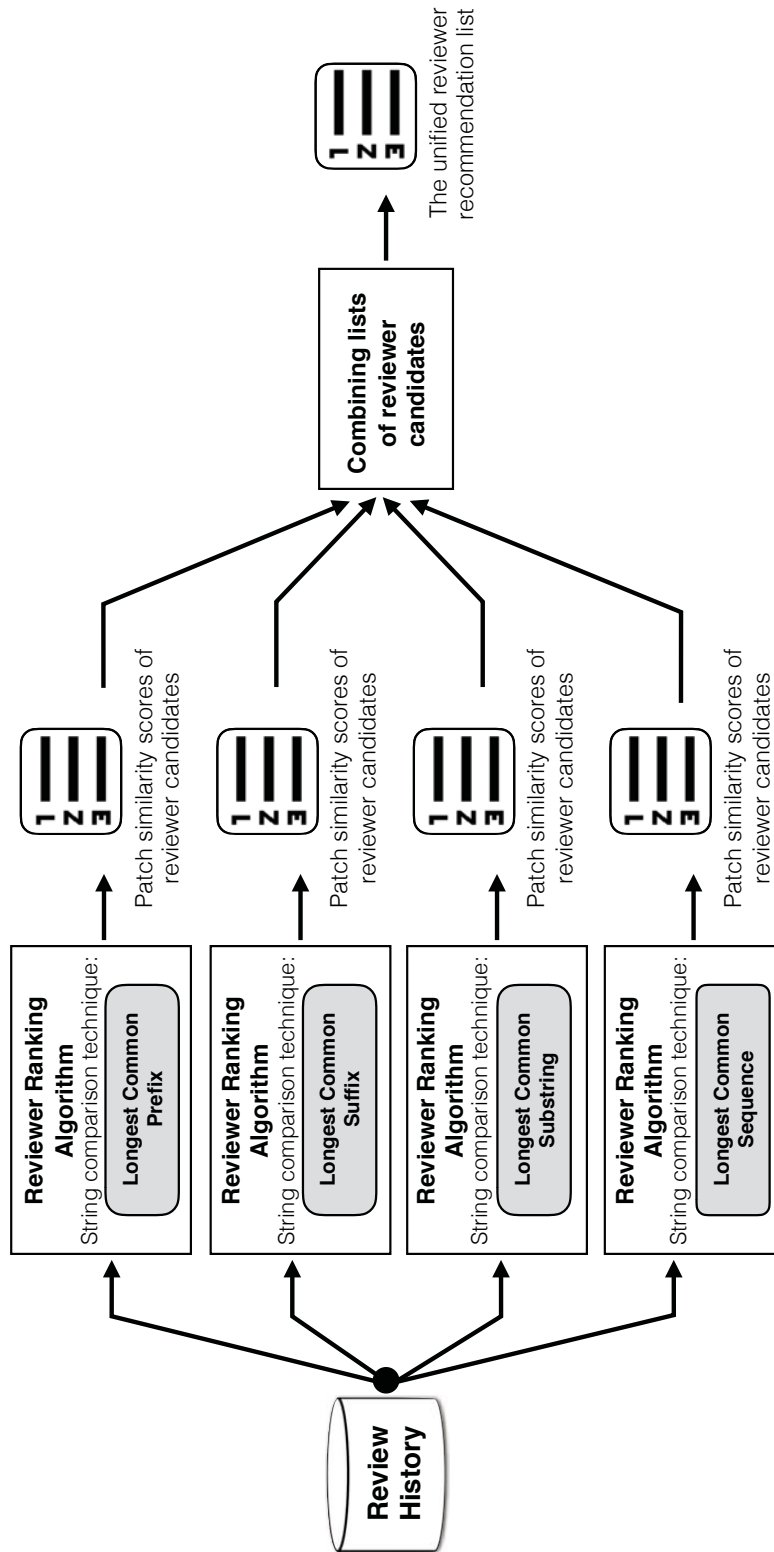


Figure 5.5. An overview of the combination approach of using four string comparison techniques in REVFinder.

REVFINDER then assigns the patch similarity scores of Patches 1 and 2 to Pick and the review similarity of Patch 1 to Kla. Finally, REVFINDER produces a recommendation list based on the patch similarity scores of the reviewers.

The recommendation list of REVFINDER is a combination of the patch similarity scores that are computed using different string comparison techniques. Since each software system may have a different structure of source files, we use four state-of-the-art string comparison techniques, i.e., Longest Common Prefix (LCS), Longest Common Suffix (LCS), Longest Common Substring (LCSubstr), and Longest Common Subsequence (LCSubseq) to compare the similarity of file paths [44].

Figure 5.5 provides an overview of a combination approach of using four string comparison techniques in REVFINDER. The reviewer ranking algorithm of REVFINDER uses each string comparison technique to compute patch similarity scores of reviewer candidates. Once the patch similarity scores of reviewer candidates are computed, the lists of reviewer candidates are combined into an unified recommendation list. This technique will make the truly-relevant reviewers “bubble up” to the top of the recommendation list, while reducing the false positive recommendations. This combination approach of individual lists has been successfully shown to improve the performance in the data mining and software engineering domains [55, 64].

Below, we describe the calculation of the reviewer ranking algorithm, the string comparison techniques, and the combination technique that we used in REVFINDER.

5.4.2. The Reviewer Ranking Algorithm of RevFinder

Algorithm 1 shows the pseudo-code of the reviewer ranking algorithm of REVFINDER. The input of the algorithm is a new patch (P_{new}) and the output is a list of reviewer candidates (C) with patch similarity scores. The algorithm consists of three main steps. Below, we describe each step of the algorithm.

Data Preparation Step

The algorithm first retrieves prior patches that were submitted before the submission date of P_{new} , and that were closed (i.e., the review status is merged or abandoned). Then, the list of prior patches is stored in `priorPatches` and sorted by their submission date in reverse chronological order (Lines 7 and 8).

Patch Comparison Step

For each patch in `priorPatches` (i.e., P_{past}), the algorithm computes a patch similarity score using the `filePathSimilarity` function (Lines 12 to 18). To do so, the `filePathSimilarity` function first compare each changed file in P_{past} with each changed file in P_{new} . For a file f_p of P_{past} and a file f_n of P_{new} , the algorithm splits the file paths into components using the slash character (“/”) as a delimiter. Then, the `filePathSimilarity` function calculates a file patch similarity score of f_p and f_n as follow:

$$\text{filePathSimilarity}(f_n, f_p) = \frac{\text{StringComparison}(f_n, f_p)}{\max(\text{Length}(f_n), \text{Length}(f_p))} \quad (5.3)$$

The `StringComparison` function uses a string comparison technique (e.g., LCP), which we describe the detail in Section 5.4.3. The `Length` function returns the number of file path components. Once the algorithm computes the file path similarity scores for all file paths in P_{past} and P_{new} , the patch similarity score of P_{past} is an average score of these file path similarity scores (Line 19).

Reviewer Score Assignment Step

Finally, the algorithm assigns the patch similarity score of P_{past} to reviewers of P_{past} (Lines 20 to 23). The patch similarity scores of reviewers are accumulated for all prior patches in which the reviewers have participated.

Algorithm 1 The reviewer ranking algorithm of REVFINDER.

```

1: Input:
2:  $P_{new}$  : A new patch
3: Output:
4:  $C$  : A list of reviewer candidates
5: Method:
6: #Data preparation step.
7:  $priorPatches \leftarrow$  A list of prior patches that were reviewed
8:  $priorPatches \leftarrow sort(priorPatches).by(submissionDate)$ 
9: #Patch comparison step computes a patch similarity score.
10: for Patch  $P_{past} \in priorPatches$  do
11:    $Files_n \leftarrow getFiles(P_{new})$ 
12:    $Files_p \leftarrow getFiles(P_{past})$ 
13:    $Score_{P_{past}} \leftarrow 0$ 
14:   for  $f_n \in Files_n$  do
15:     for  $f_p \in Files_p$  do
16:        $Score_{P_{past}} \leftarrow Score_{P_{past}} + filePathSimilarity(f_n, f_p)$ 
17:     end for
18:   end for
19:    $Score_{P_{past}} \leftarrow Score_{P_{past}} / (\text{length}(Files_n) \times \text{length}(Files_p))$ 
20:   #Reviewer score assignment step assigns the patch similarity scores to reviewers of
    $P_{past}$ 
21:   for Reviewer  $r : getCodeReviewers(P_{past})$  do
22:      $C[r].score \leftarrow C[r].score + Score_{P_{past}}$ 
23:   end for
24: end for
25: return  $C$ 

```

5.4.3. String Comparison Techniques

To compare file paths, we use four state-of-the-art string comparison techniques [44] i.e., Longest Common Prefix (LCP), Longest Common Suffix (LCS), Longest Common Substring (LCSubstr), and Longest Common Subsequence (LCSubseq). These four techniques were successfully used for string and sequence analysis in prior studies [10, 39, 120, 129]. Table 5.3 provides a definition and a calculation example for these four techniques. We briefly describe each technique and its rationale below.

Longest Common Prefix (LCP) LCP counts the number of common path components that appears in both file paths from the beginning to the last. The intuition of using LCP is that files under the same directory would have similar or related functionality [11].

Longest Common Suffix (LCS) LCS is a simply reverse calculation of LCP. LCS counts the number of common path components that appears from the end of both file paths. The intuition of using LCS is that files having the similar names would have similar functionality [44].

Longest Common Substring (LCSubstr) LCSubstr counts the number of consecutive path components that appear in both file paths. The advantage of LCSubstr is that the common paths can appear at any position of the file path. Since the file path can capture the functionality of a file [27], the related functionality should be under the same directory structure. However, the root directories or filename may not be the same.

Longest Common Subsequence LCSubseq counts the number of path components that appears in the same relative order int both file paths. The advantage of LCSubseq is that the common paths for this technique are not necessary to be contiguous. The intuition is that files under similar directory structures would have similar or related functionality [44].

5.4.4. Combination Technique

To combine lists of reviewer candidates that are produced using different string comparison techniques, we use Borda count [94]. Borda count is a voting technique that simply combines the recommendation lists based on the rank. The reviewer candidate who is in the highest rank of the lists of reviewer candidates will receive the highest points. For lists of reviewer candidates $R \in \{R_{LCP}, R_{LCS}, R_{LCSubstr}, R_{LCSubseq}\}$, a score for a reviewer candidate c_k is calculated as follow:

Table 5.3. A description and examples of calculation for file path comparison techniques. The examples are obtained from the review history of Android for the LCP, LCSustr, and LCSuSubseq techniques; and Qt for the LCS techniques. For each technique, the example files were reviewed by the same reviewer.

Functions	Description	Example
Longest Common Prefix (LCP)	Longest <u>consecutive</u> path components that appears at the <u>beginning</u> of both file paths.	$f_1 = \text{"src/com/android/settings/LocationSettings.java"}$ $f_2 = \text{"src/com/android/settings/Utils.java"}$ $\text{LCP}(f_1, f_2) = \text{length}([\text{src, com, android, settings}]) = 4$
Longest Common Suffix (LCS)	Longest <u>consecutive</u> path components that appears at the <u>end</u> of both file paths	$f_1 = \text{"src/imports/undo/undo.pro"}$ $f_2 = \text{"tests/auto/undo/undo.pro"}$ $\text{LCS}(f_1, f_2) = \text{length}([\text{undo, undo.pro}]) = 2$
Longest Common Substring (LCSustr)	Longest <u>consecutive</u> path components that appear in both file paths	$f_1 = \text{"res/layout/bluetooth_pin_entry.xml"}$ $f_2 = \text{"tests/res/layout/operator_main.xml"}$ $\text{LCSustr}(f_1, f_2) = \text{length}([\text{res, layout}]) = 2$
Longest Common Subsequence (LCSuSubseq)	Longest path components that appear in both file paths <u>in relative order</u> but not necessarily contiguous	$f_1 = \text{"apps/CtsVerifier/src/com/android/cts/verifier/sensors/MagnetometerTestActivity.java"}$ $f_2 = \text{"tests/tests/hardware/src/android/hardware/cts/SensorTest.java"}$ $\text{LCSuSubseq}(f_1, f_2) = \text{length}([\text{src, android, cts}]) = 3$

$$\text{Combination}(c_k) = \sum_{R_i \in R} M_i - \text{rank}(c_k | R_i) \quad (5.4)$$

M_i is the total number of reviewer candidates who received a non-zero score in R_i , and $\text{rank}(c_k | R_i)$ is the rank of c_k in R_i . For each c_k , Borda count assigns a point based on the rank of c_k in each recommendation list. For example, if R_{LCP} votes c_k as the 1st rank, Borda count assigns a point of M_{LCP} from R_{LCP} to c_k . Finally, the reviewer recommendation list of REVFinder is a list of reviewer candidates that are ranked according to the scores of the Borda count.

5.5. Empirical Evaluation

In this section, we present our empirical evaluation approach for REVFinder. We first describe our ground truth data for an evaluation. Then, we describe evaluation metrics that we used to evaluate REVFinder. Finally, we briefly describe a baseline comparison approach (REVIEWBOT [9]).

5.5.1. Ground Truth Data

To build the ground truth data for an evaluation, we identify developers who historically participated in the review of a studied patch as an appropriate reviewer of that patch. To ensure that the reviewers of a studied patch have sufficient expertise to examine the patch, we only select developers who confidently gave a review score of -2 or +2 to the studied patch. We also exclude the patch author who assign herself as a reviewer. Table 5.1 shows the number of reviewers that we identified for our ground truth data.

5.5.2. Evaluation Metrics

To evaluate the recommendation performance, we use the Mean Reciprocal Rank (MRR) and the top- k accuracy. These metrics are commonly used in recommendation systems for software engineering [9, 114, 116]. Since most of the patches have only one reviewer (see Table 5.1), other evaluation metrics (e.g. Mean Average Precision) that consider all of the correct answers might not be appropriate for this evaluation. We now briefly describe each evaluation metric below.

Mean Reciprocal Rank

Mean Reciprocal Rank (MRR) calculates an average of reciprocal ranks of correct reviewers in a recommendation list. The higher the value of MRR is, the better recommendation performance of REVFinder. For a set of patches P , MRR can be calculated using Equation 5.5.

$$\text{MRR} = \frac{1}{|P|} \sum_{p \in P} \frac{1}{\text{rank}(\text{candidates}(p))} \quad (5.5)$$

The $\text{rank}(\text{candidates}(p))$ returns the highest rank of the correct reviewers in the recommendation list of $\text{candidates}(p)$. If there is no correct reviewers in the recommendation list, $\frac{1}{\text{rank}(\text{candidates}(p))}$ will have a value of 0. Ideally, an approach that can provide a perfect ranking (i.e., always recommending correct reviewers at the 1st rank) should achieve an MRR value of 1.

Top- k Accuracy

Top- k accuracy calculates a rate that REVFinder can correctly recommend reviewers within k candidates. Given a set of Patches P , the top- k accuracy is calculated using Equation 5.6.

$$\text{Top-}k \text{ accuracy}(P) = \frac{\sum_{p \in P} \text{isCorrect}(p, \text{Top-}k)}{|P|} \times 100\% \quad (5.6)$$

The $\text{isCorrect}(r, \text{Top-}k)$ function returns a value of 1 if at least one of the top- k reviewer candidates are an appropriate reviewer of the patch p ; and returns a value of 0 for otherwise. For example, a top-10 accuracy of 75% indicates that REVFINDER can recommend appropriate reviewers for 75% of the patches within the top-10 reviewer candidates.

5.5.3. ReviewBot: A Baseline Approach

REVIEWBOT has been recently proposed to help developers selecting appropriate reviewers. REVIEWBOT was evaluated in the MCR process of VMware [9]. Hence, we select REVIEWBOT as our baseline approach. REVIEWBOT is a reviewer recommendation approach based on the assumption that *“the most appropriate reviewers for a code review are those who previously modified or previously reviewed the sections of code which are included in the current review”* [9, p.932]. Therefore, REVIEWBOT selects appropriate reviewers using the line-by-line modification history of source code. Below, we briefly describe the calculation of REVIEWBOT, which consists of three main steps:

1. REVIEWBOT first computes line change history, i.e., a list of past patches that impact the same changed lines as in the new patch.
2. Each reviewer candidate receives a point based on his or her frequency of being involved in the changed lines. In other words, the more frequent involvement in the changed lines of the new patch that the reviewer has done in the past, the more likely that the reviewer would be an appropriate reviewer.

3. Finally, REVIEWBOT prioritizes the reviewer candidates who recently reviewed and have the highest scores to be the most appropriate reviewers for the new patch.

5.6. Evaluation Results

In this section, we present the results of our empirical evaluation with respect to our RQ2.

(RQ2) Does RevFinder accurately recommend reviewers?

5.6.1. Approach

To address RQ2, for each studied system, we execute REVFINDER for every patch in chronological and obtain a reviewer recommendation list for each patch. We then observe the REVFINDER rank of the correct reviewers. We use MRR to measure the ranking performance of REVFINDER. The results are then compared with the MRR values of REVIEWBOT. We also evaluate the accuracy of the REVFINDER recommendation using the top- k accuracy.

5.6.2. Result

Observation 10 – RevFinder recommended the correct reviewers with a median rank of 4. Figure 5.6 shows a distribution of ranks of correct reviewers that REVFINDER and REVIEWBOT recommended. The median ranks for REVFINDER are 2, 8, 3, and 4 for Android, Qt, OpenStack, and LibreOffice, respectively. On the other hand, the median ranks for REVIEWBOT are 94, 202, 82, 63 for Android, Qt, OpenStack, and LibreOffice, respectively. We also observe that the distributions of ranks recommended by REVFINDER tend to be skewed as much as close to the first rank. These results indicate that REVFINDER is more likely to

Table 5.4. The results of Mean Reciprocal Rank (MRR) of REVFinder and a baseline approach. A MRR value of 1 indicates a perfect ranking performance of the approach.

Approach	Android	Qt	OpenStack	LibreOffice
RevFinder	0.60	0.31	0.55	0.40
REVIEWBOT	0.25	0.22	0.30	0.07

Table 5.5. The results of top- k accuracy of our approach RevFinder and a baseline ReviewBot for each studied system. The results show that RevFinder outperforms ReviewBot.

System	REVFinder			REVIEWBOT		
	Top-1	Top-3	Top-5	Top-1	Top-3	Top-5
Android	46 %	71 %	79 %	21 %	29 %	29 %
Qt	20 %	34 %	41 %	19 %	26 %	27 %
OpenStack	38 %	66 %	77 %	23 %	35 %	39 %
LibreOffice	24 %	47 %	59 %	6 %	9 %	9%

invite a correct reviewer at the top ranks while it is less likely to involve unrelated reviewers.

Observation 11 – On average, the ranking performance of RevFinder is 3 times better than that of ReviewBot. Table 5.4 shows the MRR values of REVFinder and REVIEWBOT for each studied system. Table 5.4 shows that REVFinder achieves an MRR of 0.31(Qt) - 0.60(Android) which is closer to one than REVFinder does. The MRR values of REVFinder are 2.4, 1.4, 1.8, and 5.7 times better than that of REVIEWBOT for Android, Qt, OpenStack, and LibreOffice, respectively. These results indicate that the top ranks of REVFinder are more appropriate than the top ranks of REVIEWBOT.

As Observation 10 shows that REVFinder recommended the correct reviewers with a median rank of 4, we measure the top- k accuracy with the k value of 1, 3, and 5. Table 5.5 shows the top-1, top-3, and top-5 accuracies of REVFinder and REVIEWBOT for each studied system.

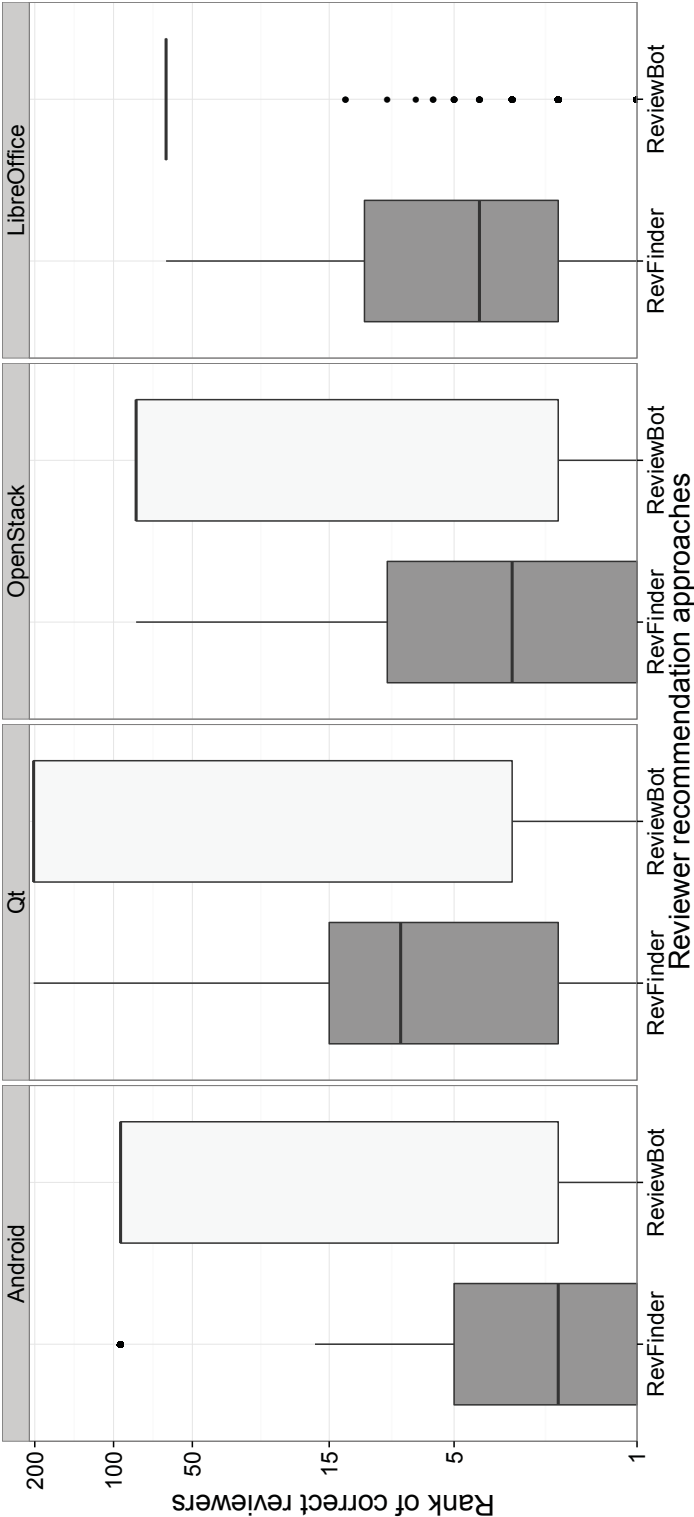


Figure 5.6. The rank distribution of the first correct reviewer that is recommended by REV-FINDER and REVIEW-BOT.

Observation 12 – RevFinder can correctly recommend reviewers for 59%-79% of patches with a top-5 recommendation. REVFINDER achieves top-5 accuracy of 79%, 41%, 77%, and 59% for Android, Qt, OpenStack, and LibreOffice, respectively. Yet, the top-1 accuracy of REVFINDER is relatively low, indicating that there is a room for improvement if a correct recommendation of a reviewer within one candidate is needed. Nevertheless, the results of top-5 accuracy leads us to conclude that leveraging the similarities between the path of a newly changed file and the paths of previously reviewed files can be used to accurately recommend appropriate reviewers.

Table 5.5 also shows that, for every studied system, REVFINDER achieves higher top- k accuracy than REVIEWBOT. At the top-5 accuracy, REVFINDER achieves an accuracy of 2.7, 1.5, 2, and 7 times higher than that of REVIEWBOT for Android, Qt, OpenStack, and LibreOffice, respectively. We also find similar results for other top- k accuracy metrics. This result indicates that REVFINDER considerably outperforms REVIEWBOT.

Summary: REVFINDER can recommend correct reviewers with a median rank of 4. The ranking performance of REVFINDER is 3 times better than that of REVIEWBOT, indicating that REVFINDER ranks the appropriate reviewers higher than REVIEWBOT. Moreover, REVFINDER correctly recommended 59%-79% of patches with a top-5 recommendation. These results indicate that leveraging the similarities between the path of a newly changed file and the paths of previously reviewed files can accurately recommend reviewers.

5.7. Discussion

We now discuss the performance and applicability of REVFINDER.

5.7.1. Performance: Why does RevFinder outperform

ReviewBot?

Our empirical evaluation shows that REVFINDER outperforms REVIEWBOT which is the baseline approach for this study (Observations 11 and 12). The difference between REVFINDER and REVIEWBOT is the granularity of code review history that is used for selecting the appropriate reviewers. REVFINDER uses the code review history at the file path-level, while REVIEWBOT uses the code review history at the source code line-level. Intuitively, selecting reviewers who have examined the exact same lines seems to be the best choice for those projects with high frequent changes of source code. However, it is not often that files are frequently changed at the same lines [68]. As the MCR processes is relatively new and recently adopted, the performance of REVIEWBOT would be limited due to the small amount of review history. To better understand this, we count the frequency of change history at the line level. We observed that 70%-90% of lines of code are changed only once, indicating that MCR tools have a short history of the changed line level. Hence, REVIEWBOT achieved the low performance in the context of our study.

5.7.2. Applicability: Can RevFinder help a patch author to find appropriate reviewers?

In RQ1, our exploratory study has shown that the patch authors of many patches were suffering from selecting appropriate reviewers (Observation 8). Moreover, we find that patches with the reviewer selection problem take a longer time in MCR processes than patches without the problem (Observation 9). To confirm whether REVFINDER can help a patch author to find appropriate reviewers, we execute REVFINDER for the patches with the reviewer selection problem that we had identified in our explanatory study (see Table 5.2). We find that, on average,

REVFINDER can correctly recommend reviewers for 80% of these patches within its top 10 recommendations. This result indicates that the patch authors can use a list of recommendation produced by REVFinder to assign a reviewer if they cannot find an appropriate reviewer for their patches. This finding leads us to believe that REVFinder can help a patch author find appropriate reviewers, which in turn would reduce the time that is spent in MCR processes.

5.8. Threats to Validity

We discuss potential threats to the validity of our work as follows:

5.8.1. Internal Validity

Our review classification process in RQ1 involves a manual classification. The classification process was conducted by the author who is not involved in the code review process of the studied systems. The results of the manual classification by a domain expert might be different. Nevertheless, our classification method is based on the discussion that is written by the developers who are involved in the MCR processes (see Figure 5.2), and the review classification results is verified by the second author of our paper [124].

In our exploratory study, we measure time that is spent in MCR processes to investigate the impact that the reviewer selection problem can have on MCR processes. However, this time measurement includes both the time length of finding reviewers and reviewing. Unfortunately, the code review tool did not record the time when a reviewer is selected. We also could not use a feedback delay, i.e., time from the patch submission to the posting of the first reviewer message as our time measure. This is because those patches that have the reviewer selection problem often receive messages about suggesting a reviewer during the review discussion (see Figure 5.2(b)). Since there is a limitation of measuring

time to find reviewers, we must rely on heuristics to recover this information. Nevertheless, we normalize the time that is spent in MCR processes by patch size in order to control a confounding factor, i.e., a larger patch is more likely to take longer time in MCR processes.

5.8.2. External Validity

Our empirical results are limited to four datasets i.e., Android, OpenStack, Qt, and LibreOffice. However, we cannot claim that the same observations would hold for other systems. Future work should focus on an evaluation in other systems with larger number of reviewers to better generalize the results of our approach.

5.8.3. Construct Validity

The first threat involves a lack of reviewer retirement information. It is possible that reviewers are retired or are no longer involved in the code review system. Therefore, the performance of our approach might be affected by its recommending retired reviewers. Another threat involves the workload of reviewers. It is possible that reviewers would be burdened with a large number of assigned reviews. Therefore, considering workload balancing would reduce tasks of these potential reviewers and the number of awaiting reviews.

5.9. Summary

Finding an appropriate reviewer to critique a new patch can be a tedious task for a patch author. Therefore, in this chapter, we empirically evaluate the impact of the reviewer selection problem can have on the time that is spent on MCR processes. Moreover, in order to help developers find appropriate reviewers, we propose REVFINDER, a file location-based reviewer recommendation approach. We leverage the similarities between the path of a newly changed file and the paths

of previously reviewed files to recommend an appropriate reviewer. The intuition behind REVFinder is that *files that are located in similar file paths would be managed and reviewed by a similar set of experienced reviewers*. We evaluate the recommendation performance in terms of the accuracy and ranking performance. We also compare the performance of REVFinder with the previously proposed approach called REVIEWBOT. Through an empirical study of 42,045 patches spread across the Android, Qt, OpenStack, and LibreOffice open source systems, we make the following observations:

- 4%-30% of patches have the reviewer selection problem. These patches with the problem take 12 days longer than patches without the problem. A patch author indeed do have difficulties selecting appropriate reviewers (Observations 8 and 9).
- REVFinder can recommend the correct reviewers with a median rank of 4 (Observation 10). Moreover, the ranking performance of REVFinder is 3 times better than that of REVIEWBOT (Observation 11).
- Moreover, REVFinder correctly recommends 59%-79% of patches within a top-5 recommendation, which is more accurate than REVIEWBOT (Observation 12).

Our results lead us to conclude that patch authors are suffering from selecting appropriate reviewers for a new patch. Hence, we believe that implementing a reviewer recommendation into MCR tools could help developers find appropriate reviewers, which in turn lead to a more effective code review process.

Part III.

Reviewer Involvement in Modern Code Review Processes

The Impact of Reviewer Involvement on Software Quality

An earlier version of the work in this chapter appears in
the Proceedings of the 12th International Working
Conference on Mining Software Repositories (MSR) [121].

In addition to the expertise, the involvement of developers is another key factor that has an impact on the quality of software systems [1]. Recent research finds that the investment of MCR, i.e., the proportion of patches that were reviewed or discussed, shares an inverse relationship with the incidence of both post-release defects [73, 74] and software design anti-patterns [81]. Motivated by the findings of the recent research, in this chapter, we investigate the amount reviewer involvement in MCR processes. In particular, we investigate the reviewer involvement in defective files along two perspectives: (1) files that will eventually have defects, i.e., future-defective files and (2) files that have historically been defective, i.e., risky files. The results of our case studies demonstrate that both future-defective files and risky files tend to be reviewed less rigorously than their clean counterparts. We also observe that the concerns addressed during the MCR

reviews tend to enhance evolvability, i.e., ease future maintenance (like documentation), rather than focus on functional issues (like incorrect program logic). Our findings suggest that although functionality concerns are rarely addressed during code review, the rigor of the reviewing process that is applied to a source code file throughout a development cycle shares a link with its defect proneness.

6.1. Introduction

Since MCR tools do not impose a strict criteria of reviewer involvement, MCR reviews may not foster a sufficient amount of reviewer involvement to mitigate the risk of having defects in reviewed patches. Hence, in this chapter, we investigate the code reviewing practices of MCR in terms of: (1) code review activity, and (2) concerns addressed during code review. We characterize code reviewing practices using 11 metrics that are grouped along three dimensions, i.e., review intensity, review participation, and reviewing time. We then comparatively study the difference of these code reviewing practices in defective and clean source code files. We also investigate defective files along two perspectives: (1) files that will eventually have defects (called *future-defective files*), and (2) files that have historically been defective (called *risky files*). Using data that is collected from the Qt open source system, we address the following two research questions:

(RQ1) Do developers follow lax code reviewing practices in files that will eventually have defects?

Motivation. Prior work shows that lax code reviewing practices are correlated with future defects in the corresponding software components [73]. For example, components with many changes that have no associated review discussion tend to have post-release defects. While these prior findings suggest that a total lack of code review activity may increase the risk of post-release defects, little is known about how much code review activity is

“enough” to mitigate this risk.

Results. We find that future-defective files tend to undergo reviews that are less intensely scrutinized, having less team participation, and a faster rate of code examination than files without future defects. Moreover, most of the changes made during the code reviews of future-defective files are made to ease future maintenance rather than to fix functional issues.

(RQ2) Do developers adapt their code reviewing practices in files that have historically been defective?

Motivation. Since the number of prior defects is a strong indicator of the incidence of future defects [41], the files that have historically been defective may require additional attention during the code review process. In other words, to reduce the likelihood of having future defects, developers should more carefully review changes made to these risky files than changes made to files that have historically been defect-free. However, whether or not developers are actually giving such risky files more careful attention during code reviewing remains largely unexplored.

Results. We find that developers are likely to review changes of risky files with less scrutiny and less team participation than files that have historically been defect-free (called normal files). Developers tend to address evolvability and functionality concerns in the reviews of risky files more often than they do in normal files. Moreover, such risky files that will eventually have defects tend to undergo less rigorous code reviews that more frequently address evolvability concerns than the risky files that will eventually be defect-free.

Our results lead us to conclude that lax code reviewing practices could lead to future defects in software systems. Developers are not as careful when they review changes made to risky files despite their historically defective nature. These findings suggest that files that have historically been defective should be given

more careful attention during the code review process, since the rigor of the reviewing process shares a link with defect proneness.

6.1.1. Chapter Organization

The remainder of this chapter is organized as follow. Section 6.2 describes our case study design, while Section 6.3 presents the study results. Section 6.4 provides a broader implication of the results. Section 6.5 discloses the threats to the validity. Finally, Section 6.6 summarizes this study.

6.2. Case Study Design

In this section, we describe the studied system, and our data preparation and analysis approaches.

6.2.1. Studied System

In order to address our research questions, we perform an empirical study on a large, rapidly-evolving open source system with a globally-distributed development team. In selecting the subject system, we identified two important criteria:

Criterion 1: Active MCR Practices — We want to study a subject system that actively uses MCR for the code review process, i.e., a system that examines and discusses software changes through a code review tool. Hence, we only study a subject system where a large number of reviews are performed using a code review tool.

Criterion 2: High Review Coverage — Since we will investigate the differences of MCR practices in defective and clean files, we need to ensure that a lack of code review is not associated with defects [73]. Hence, we focus our study on a system that has a large number of files with 100% review coverage, i.e.,

Table 6.1. An overview of the studied Qt system.

Version	LOC	Commits		Files	
		with Review	Total	with 100% review coverage	Changed Files
Qt 5.0	5,560,317	9,677	10,163	11,689	25,615
Qt 5.1	5,187,788	6,848	7,106	12,797	19,119

files where every change made to them is reviewed by at least one reviewer other than its author.

Due to the human-intensive nature of carefully studying the code review process, we decided to perform an in-depth study on a single system instead of examining a large number of projects. With our criteria in mind, we select Qt, an open source cross-platform application and UI framework developed by the Digia corporation. Table 6.1 shows that the Qt system satisfies our criteria for analysis. In terms of criterion 1, the development process of the Qt system has achieved a large proportion of commits that are reviewed. In terms of criterion 2, the Qt system has a large number of files where 100% of the integrated changes are reviewed.

6.2.2. Data Preparation

We used the Gerrit review and code datasets that are provided by prior work [46, 73]. The Gerrit review dataset of Hamasaki *et al.* describes patch information, reviewer scoring, the personnel involved, and review discussion history [46]. The code dataset of McIntosh *et al.* describes the recorded commits on the **release** branch of the Qt VCSs during the development and maintenance of each studied Qt release [73]. For each of our research questions, we construct a review database by linking the Gerrit review and code datasets. Figure 6.1 provides an overview of our dataset linking process, which is broken down into three steps that we describe below.

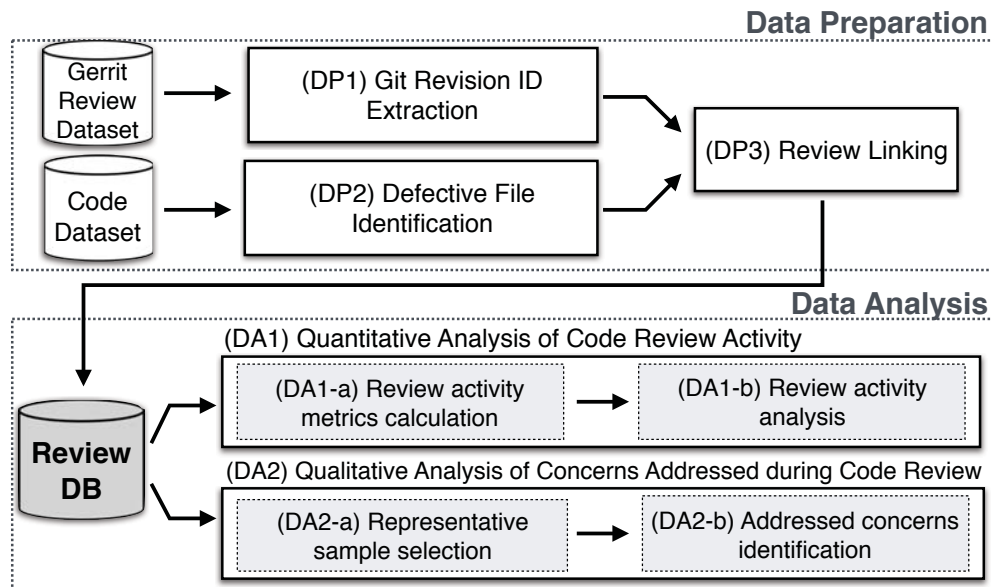


Figure 6.1. An overview of data preparation and data analysis approaches.

(DP1) Git Revision ID Extraction

In order to link the Gerrit review and code datasets, we extract the Git revision ID from the message that is automatically generated by the Qt CI bot. After a patch is accepted by reviewers, it is merged into the `release` branch and the bot adds a message to the review discussion of the form: “*Change has been successfully cherry-picked as <Git Revision ID>*”.

(DP2) Defective File Identification

To identify the future-defective and risky files, we count the number of post-release defects of each file. Similar to prior work [73], we identify post-release defects using defect-fixing commits that are recorded during the six-month period after the release date.

Figure 6.2 illustrates our approach to identify the future-defective files for RQ1 and the risky files for RQ2. We use the following criteria to identify defective files:

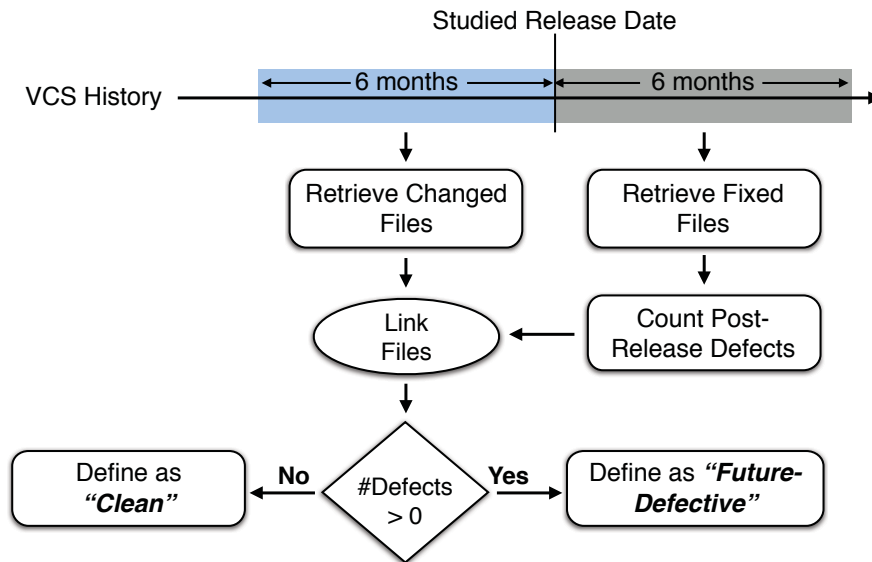
Future-defective files identification for RQ1: Figure 6.2(a) shows our classification approach for future-defective files. We classify the files that have post-release defects as *future-defective files*. Files that do not have post-release defect are classified as *clean files*.

Risky files identification for RQ2: Figure 6.2(b) shows our classification approach for risky files. We classify the files that had post-release defects in the prior release as *risky files*, while *normal files* are files that did not have post-release defects in the prior release.

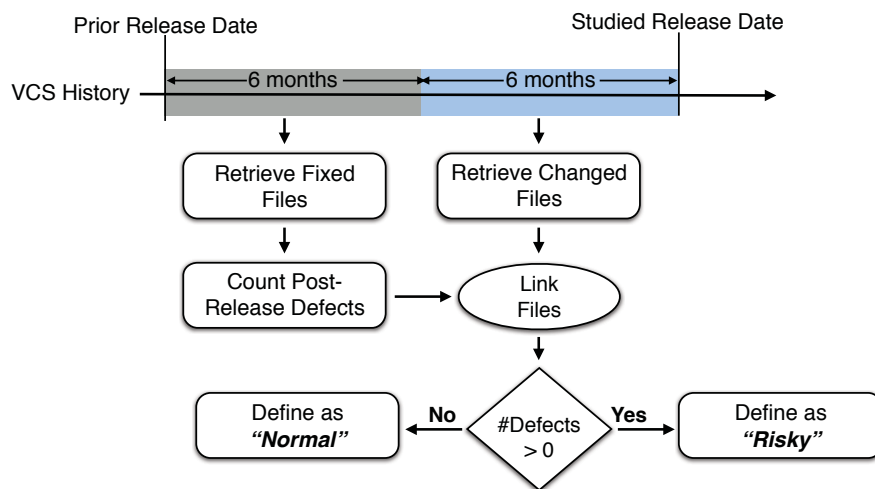
(DP3) Review Linking

To find the reviews that are associated with changes made to the studied files, we first link the commits to reviews using the Git revision ID. We only study the reviews of changes that can be linked between the code and review datasets. We then produce a review database by selecting pre-release reviews of the studied files. We select reviews of future-defective (or risky) files from those reviews that are associated with at least one future-defective (or risky) file. The reviews that are not associated with any future-defective (or risky) files are linked to clean (or normal) files.

We conservatively link the reviews that are associated with both future-defective (or risky) files and clean (or normal) files to future-defective (or risky) files, when these reviews could have been linked to clean (or normal) files. We do so because we feel that the worst-case scenario where we mistakenly link some reviewing activities to future-defective (or risky) files is more acceptable than the worst-case scenario where we mistakenly link some reviewing activities to clean (or normal) files.



(a) RQ1: An identification of future-defective files



(b) RQ2: An identification of risky files

Figure 6.2. Our approach to identify defective files.

6.2.3. Data Analysis

To address our research questions, we perform quantitative (DA1) and qualitative (DA2) analyses. Figure 6.1 provides an overview of our analyses. We describe each analysis below.

(DA1) Quantitative Analysis of Code Review Activity

We study the differences in code review activity between future-defective (or risky) and clean (or normal) files. To do so, we calculate several code review activity metrics and measure the difference in code review activity metrics using a statistical approach. We describe each step in this process below.

(DA1-a) Reviewing Activities Metrics Calculation. Table 6.2 provides an overview of the 11 metrics that we use to measure code review activity. Since patch size is often correlated with code review activity [57, 99, 127, 131] and defect-proneness [80, 84], we normalize each raw code review activity metric by the patch size. Our metrics are grouped into review intensity, review participation, and reviewing time dimensions, which we describe below:

Review Intensity measures the scrutiny that was applied during the code review process. We conjecture that files that are intensely scrutinized before integration are less likely to have future defects. We use four metrics to measure review intensity: (1) *Number of iterations* counts how many times a patch has been revised prior to its integration. (2) *Discussion length* counts how many messages posted in the patch discussion and the inline comments inserted into the patch. We exclude the automatically-generated comments from EWS and CI bots in order to focus on reviewer feedback. (3) *Proportion of revisions inspired by feedback* measures the proportion of revisions that are inspired by a reviewer either posting a message or assigning a review score. (4) *Churn between revisions* counts the number of

lines that were added and deleted between revisions of the patch. We filter out revisions that have a large difference because they are likely related to version control bookkeeping tasks, such as merging branch updates or rebasing [125].

Review Participation measures how much the development team invests in the code review process. We conjecture that files with a large and active involvement are less likely to have future defects. We use four metrics to measure review participation: (1) *Number of reviewers* counts how many participants who either post a general comment, inline comment, or assign a review score. (2) *Number of authors* counts the number of developers who upload a revision of a patch. (3) *Number of non-author voters* counts the number of participants who assign a review score for a patch but are not the author. (4) *Proportion of reviewer agreement* measures the proportion of review scores for a patch that are positive, i.e., +1 or +2.

Reviewing Time measures the duration of a code review. We conjecture that files that are reviewed for a longer time are less likely to have future defects. We use three metrics to measure reviewing time: (1) *Review length* measures the duration of a code review from the time that the first revision of the patch is uploaded to the time that reviewers accept the patch for integration. (2) *Response delay* measures the time from when the first patch is uploaded to the time that the first reviewer provides feedback. (3) *Average review rate* measures an average of the number of lines that was reviewed within an hour for each patch revision (KLOC/Hour).

(DA1-b) Reviewing Activities Analysis. We use a statistical approach to determine whether code review activity in future-defective (or risky) files is significantly different from code review activity in clean (or normal) files.

To statistically confirm the difference of the code review activity in future-defective (or risky) and clean (or normal) files, we first test for distribution in our

Table 6.2. A taxonomy of the code review activity metrics. The metrics normalized by patch size are marked with a dagger symbol ([†]).

Metric	Description	Conjecture
<i>Review Intensity Dimension</i>		
Number of Iterations [†]	Number of review iterations for a patch prior to its integration.	Fixing a defect found in each round of multiple iterations of a review would reduce the number of defects more than a single iteration of review [91].
Discussion Length [†]	Number of general comments and inline comments written by reviewers.	Reviewing proposed changes with a long discussion would find more defects and provide a better solution [81, 128].
Proportion of Revisions without Feedback	Proportion of iterations that are not inspired by a reviewer neither posting a message nor a score.	Although a code review of MCR can be done by bots, the suggestion can be either superficial or false positives [89]. More revisions that are manually examined by reviewers would lead to a lower likelihood of having future defects.
Churn during Code Review [†]	Number of lines that were added and deleted between revisions.	More lines of codes that were revised during code review would lead to a lower likelihood of having a future defect [25].
<i>Review Participation Dimension</i>		
Number of Reviewers [†]	Number of developers who participate in a code review, i.e., posting a general comment, or inline comment, and assigning a review score.	Changes examined by many developers are less likely to have future defects [96, 100].

Continued on next page

Table 6.2 A taxonomy of the code review activity metrics. *Continued from previous page*

Metric	Description	Conjecture
Number of Authors [†]	Number of developers who upload a revision for proposed changes.	Changes revised by many authors may be more defective [21, 41].
Number of Non-Author Voters [†]	Number of developers who assign a review score, excluding the patch author.	Changes that receive a review score from the author may have essentially not been reviewed [73].
Proportion of Review Disagreement	A proportion of reviewers that vote for a disagreement to accept the patch, i.e., assigning a negative review score.	A review with a high rate of acceptance discrepancy may induce a future fix.
<i>Reviewing Time Dimension</i>		
Review Length [†]	Time in days from the first patch submission to the reviewers acceptance for integration.	The longer time of code review, the more defects would be found and fixed [91, 100].
Response Delay	Time in days from the first patch submission to the posting of the first reviewer message.	Reviewing a patch promptly when it is submitted would reduce the likelihood that a defect will become embedded [100].
Average Review Rate	An average of the number of lines that was reviewed in an hour for each revision of the patch (KLOC/Hour).	A review with a fast review rate may lead the changes that are defective [35, 60, 104].

Table 6.3. A contingency table of a code review activity metric (m), where a and c represent the number of reviews of defective files, and b and d represent the number of reviews of their clean counterparts.

	Low Metric Value $m \leq \text{median}_m$	High Metric Value $m > \text{median}_m$
Have defects	a	c
No defects	b	d

data using the Shapiro-Wilk test ($\alpha = 0.05$). We observe that the distributions of code review activity metrics do not follow a normal distribution ($p < 2.2 \times 10^{-16}$ for all of the metrics). Thus, we use a non-parametric test, i.e., the one-tailed Mann-Whitney U test to check for significant differences in the code review activity metrics of future-defective (or risky) files and clean (or normal) files ($\alpha = 0.05$).

We also measure the relative impact in order to understand the magnitude of the relationship. We estimate the relative impact using the odds ratio [31]. We compare the odds of future-defective (or risky) files that undergo reviews with high metric values (greater than the median) and reviews with low metric values (less than or equal to the median). From the contingency table constructed for a code review activity metric (m) as shown in Table 6.3, we can measure the relative impact using the calculation below.

$$\text{imp}(m) = \frac{(c/d) - (a/b)}{(a/b)} \quad (6.1)$$

A positive relative impact indicates that a shift from low metric values to high metric values is accompanied by an increase in the likelihood of future (or past) defect proneness, whereas a negative relative impact indicates a decrease in that likelihood of future (or past) defect proneness.

(DA2) Qualitative Analysis of Concerns Addressed during Code Review

We compare the concerns that were addressed during code review of future-defective (or risky) files and clean (or normal) files. Similar to Beller *et al.* [16], we identify the concerns by manually labelling the types of changes made to a patch between revisions. We describe each step below.

(DA2-a) Representative Sample Selection. As the full set of review data is too large to manually examine in its entirety, we randomly select a statistically representative sample for our analysis. To select a representative sample, we determine the sample size using a calculation of $s = \frac{z^2 p(1-p)}{c^2}$, where p is the proportion that we want to estimate, $z = 1.96$ to achieve a 95% confidence level, and $c = 0.1$ for 10% bounds of the actual proportion [63, 67]. Since we did not know the proportion in advance, we use $p = 0.5$. We further correct the sample size for the finite population of reviews P using $ss = \frac{s}{1 + \frac{s-1}{P}}$. Since we consider only changes that occur during code review, we randomly select the representative sample from those reviews that have at least two revisions.

(DA2-b) Addressed Concerns Identification. To identify the concerns that were addressed during code review, we manually label the changes that occurred between revisions using the file-by-file comparison view of the Gerrit system. Each change is labelled as being inspired by reviewer feedback or not, as well as the corresponding type. For the type of a change, we use the change classification defined by Mäntylä and Lassenius [70]. We also add the traceability category, which refers to bookkeeping changes for VCSs, into the change classification. Figure 6.3 shows the classification schema that we use in this study. For each type of change, we count how many reviews make such changes. Since many changes can be made during a review, the sum of the frequencies of each type can be larger than the total number of reviews. Below, we briefly describe each type in our change classification schema.

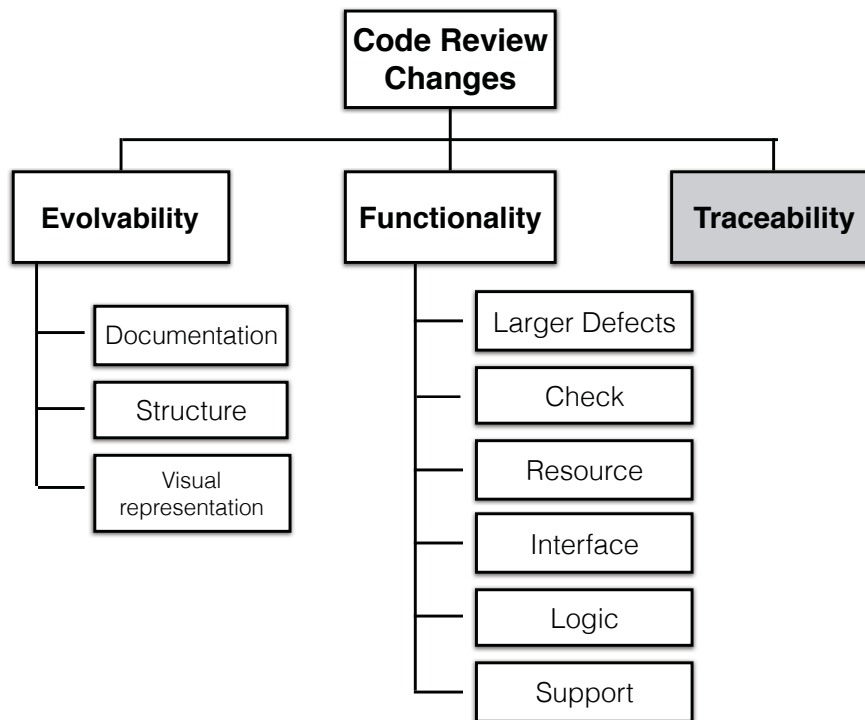


Figure 6.3. Classification schema for changes that are addressed during the code review process [70]. We add the traceability type.

Evolvability refers to changes made to ease the future maintenance of the code.

This change type is composed of three sub-types: (1) *Documentation* refers to changes in parts of the source code that describe the intent of the code, e.g., identifier names or code comments, (2) *Structure* refers to code organization, e.g., refactoring large functions, and (3) *Visual representation* refers to changes that improve code readability, e.g., code indentation or blank line usage.

Functionality refers to changes that impact the functionality of the system. This

change type is composed of six sub-types: (1) *Larger defects* refer to changes that add missing functionality or correct implementation errors, (2) *Check* refers to validation mistakes, or mistakes made when detecting an invalid value, (3) *Resource* refers to mistakes made with data initialization or ma-

nipulation, (4) *Interface* refers to mistakes made when interacting with other parts of the software, such as other internal functions or external libraries, (5) *Logic* refers to computation mistakes, e.g., incorrect comparison operators or control flow, and 6) *Support* refers to mistakes made with respect to system configuration or unit testing.

Traceability refers to bookkeeping changes for VCSs. We add this type into our classification schema because the software development of large software systems is also concerned with the management of source code repositories [50, 53]. For instance, developers should describe the proposed change using a detailed commit message, and the proposed change must be applied to the latest version of the codebase.¹

6.3. Case Study Results

In this section, we present the results of our case study with respect to our two research questions. For each research question, we present and discuss the results of quantitative (DA1) and qualitative (DA2) analyses.

(RQ1) Do developers follow lax code reviewing practices in files that will eventually have defects?

Table 6.4 provides an overview of the review database that we construct to address RQ1. We conjecture that files that are intensely scrutinized, with more team participation, that are reviewed for a longer time, and often address functionality concerns are less likely to have defects in the future.

We now present our empirical observations, followed by our general conclusion.

¹<http://qt-project.org/wiki/Gerrit-Introduction>

Table 6.4. An overview of the review database of RQ1.

	Qt 5.0		Qt 5.1	
	Future-defective	Clean	Future-defective	Clean
Studied Files	1,176	10,513	866	11,931
Related Reviews	3,470	2,727	2,849	2,690
Review Sample	93 (405 revisions)	93 (344 revisions)	93 (371 revisions)	93 (342 revisions)

(RQ1-DA1) Quantitative Analysis of Code Review Activity

Observation 13 – Future-defective files tend to undergo less intense code review than clean files do. As we suspected, Table 6.5 shows that future-defective files tend to undergo reviews that have fewer iterations, shorter discussions, and have more revisions that are not inspired by reviewer feedback than clean files do. Mann-Whitney U tests confirm that the differences are statistically significant ($p < 0.001$ for all of the review intensity metrics).

However, future-defective files tend to change more during code review than clean files. Table 6.5 shows that the churn during code review of future-defective files is higher than that of clean files ($p < 0.001$). This may be because changes made to future-defective files tend to be more controversial. For example, review ID 27977² proposes a controversial change that reviewers disagreed with, asking the author to revise the proposed changes many times before reluctantly allowing the integration of the changes into the upstream VCSs.

Observation 14 – Future-defective files tend to undergo reviews with less team participation than clean files do. Table 6.5 shows that future-defective files tend to be reviewed by fewer reviewers, involve fewer non-author voters, and have a higher rate of review disagreement than clean files do. Mann-Whitney U tests confirm that the differences are statistically significant ($p < 0.001$ for the number of reviewers and the non-author voters metrics, and $p < 0.05$ for the proportion of review disagreement metric).

²<https://codereview.qt-project.org/#/c/27977>

Table 6.5. Results of one-tailed Mann-Whitney U tests ($\alpha = 0.05$) for code review activity metrics of future-defective and clean files.

Metric	Statistical Test		Relative Impact (%)	
	Qt 5.0	Qt 5.1	Qt 5.0	Qt 5.1
<i>Review Intensity</i>				
#Iterations [†]	Future-defective < Clean***	Future-defective < Clean***	-24 ↓	-20 ↓
Discussion Length [†]	Future-defective < Clean***	Future-defective < Clean**	-19 ↓	-14 ↓
Revisions without Feedback	Future-defective > Clean***	Future-defective > Clean*	7 ↑	5 ↑
Churn during Code Review [†]	Future-defective > Clean***	Future-defective > Clean***	11 ↑	11 ↑
<i>Review Participation</i>				
#Reviewers [†]	Future-defective < Clean***	Future-defective < Clean***	-26 ↓	-20 ↓
#Authors [†]	Future-defective < Clean***	Future-defective < Clean***	-33 ↓	-27 ↓
#Non-Author Voters [†]	Future-defective < Clean***	Future-defective < Clean***	-30 ↓	-26 ↓
Review Disagreement	Future-defective > Clean*	Future-defective > Clean*	3 ↑	4 ↑
<i>Reviewing Time</i>				
Review Length [†]	-	Future-defective > Clean***	-	18% ↑
Response Delay	-	Future-defective > Clean*	-	11% ↑
Average Review Rate	Future-defective > Clean***	Future-defective > Clean***	42% ↑	16% ↑

[†] The metrics are normalized by patch size.

Statistical significance: * $p < 0.05$, ** $p < 0.01$, *** $p < 0.001$.

Furthermore, Table 6.5 shows that future-defective files tend to have less authors who upload revisions than clean files do ($p < 0.001$). We observe that additional authors often help the original author to improve the proposed changes. For example, in review ID 35360,³ the additional author updated the proposed changes to be consistent with his changes from another patch. This suggests that the proposed change can be improved and complex integration issues can be avoided when it is revised by many developers during code review.

Observation 15 – Future-defective files tend to undergo reviews with a faster rate of code examination than clean files do. Table 6.5 shows that future-defective files tend to be reviewed with a faster average review rate than clean files. A Mann-Whitney U test confirms that the difference is statistically significant ($p < 0.001$). We also observe that in Qt 5.1, the future-defective files tend to receive longer response delay than the clean files do, while we cannot confirm the statistical significance of the difference in Qt 5.0.

Table 6.5 shows that the review length of future-defective files tends to be longer than clean files in Qt 5.1 ($p < 0.001$). This finding contradicts our conjecture that we describe in Table 6.2. We observe that some of the reviews take a longer time due to a lack of reviewer attention. For example, in review ID 32926,⁴ there is little prompt discussion about a proposed change. Since the patch already received a review score of +1 from a reviewer and there were no other comments after waiting for 9 days, the author presumed that the change was clean and self-approved his own change. This lack of reviewer attention may in part be due to long review queues [14]. This finding complements observation 14 — files that are reviewed with little team participation tend to have future defects.

Table 6.5 shows that the metrics with the largest relative impacts are the average review rate for Qt 5.0 (42%) and the number of authors for Qt 5.1 (-27%), while the proportion of review disagreement metric has the smallest relative

³<https://codereview.qt-project.org/#/c/35360>

⁴<https://codereview.qt-project.org/#/c/32926>

impact for both Qt versions (an average of 3.5%). Furthermore, the number of reviewers, authors, and non-author voters also have a large relative impact, which ranges between -33% and -20%.

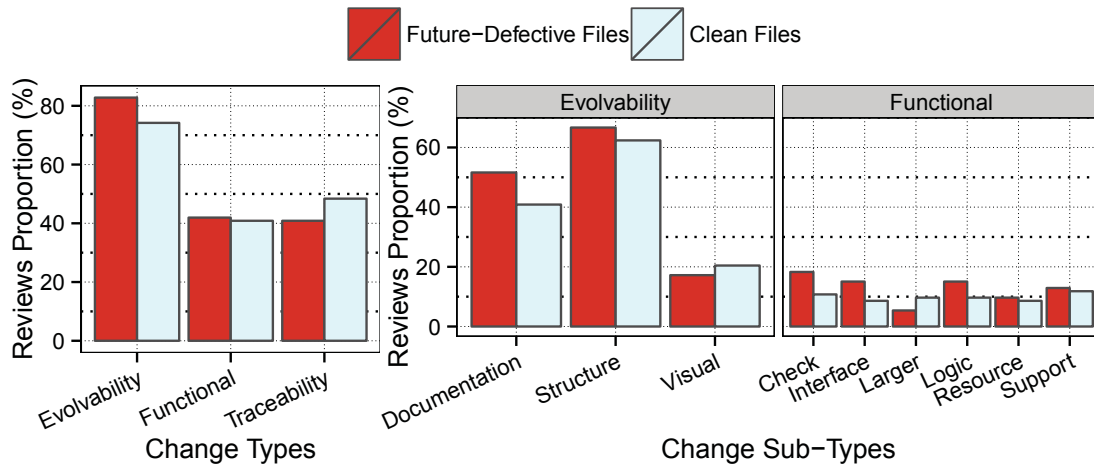
(RQ1-DA2) Qualitative Analysis of Concerns Addressed during Code Review

Observation 16 – For evolvability changes, future-defective files tend to more frequently address documentation and structure concerns than clean files do.

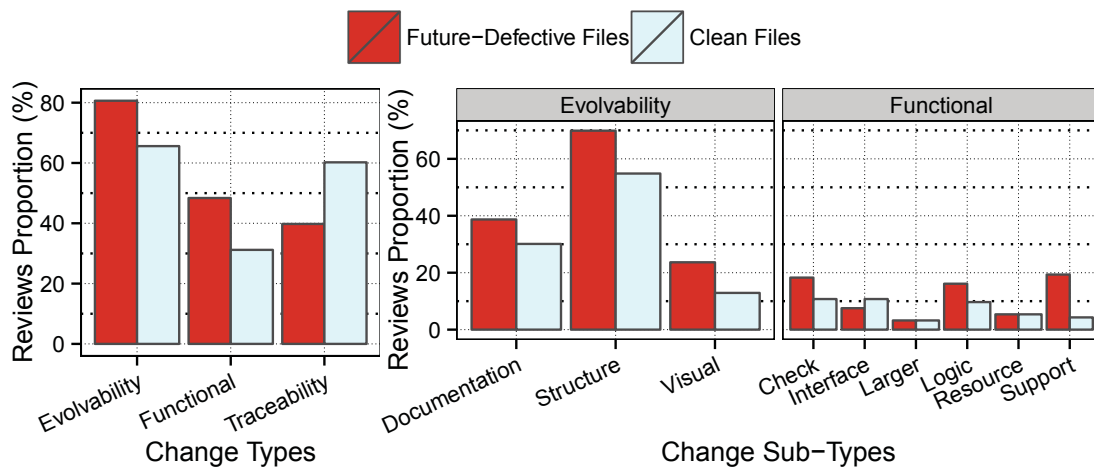
Figure 6.4 shows that the proportion of reviews in future-defective files that make documentation and structure changes is higher than the corresponding proportion in the clean files. There are differences of 11 and 9 percentage points (52%-41% and 39%-30%) in documentation changes, and 5 and 15 percentage points (67%-62% and 70%-55%) in structure changes for Qt 5.0 and Qt 5.1, respectively. We observe that the documentation changes involve updating copyright terms, expanding code comments, and renaming identifiers. The structure changes relate to removing dead code and reusing existing functions rather than implementing new ones. Moreover, we find that the documentation changes were inspired by reviewers in future-defective files (21%) more often than clean files (10%), indicating that reviewers often focus on documentation issues in future-defective files.

Furthermore, we find that evolvability is the most frequently addressed concern during code review of both clean and future-defective files. Figure 6.4 shows that, similar to prior work [16, 70], evolvability changes account for the majority of changes in the reviews, i.e., 82% of the reviews of future-defective files and 70% of the reviews of clean files on average.

Observation 17 – For functionality changes, future-defective files tend to more frequently address check, logic, and support concerns than clean files do. Figure 6.4 shows that the proportion of reviews in future-defective files that make



(a) Qt 5.0



(b) Qt 5.1

Figure 6.4. Distribution of change types that occurred during the code review of future-defective and clean files. The sum of review proportion is higher than 100%, since a review can contain many types of changes.

check, logic, and support changes is higher than the corresponding proportion in the clean files. There are differences of 7 and 6 percentage points (18%-11% and 16%-10%) in check and logic changes, respectively. We observe that many check changes involve validating variable declarations. The logic changes mainly relate to changing comparison expressions. For the support changes, the proportion of reviews in Qt 5.1 shows a clear difference of 15 percentage points (19%-4%) between future-defective and clean files. Moreover, in Qt 5.1, the support changes are addressed by the author in future-defective files more often than clean files. The proportion of reviews is 15% and 2% in future-defective files and clean files, respectively. On the other hand, there are few reviews where reviewers inspire functionality changes. The proportion of reviews ranges between 1% - 9% (average of 5%) in future-defective files and between 1% - 6% (average of 4%) in clean files.

We also find that the reviews of clean files tend to more frequently address traceability concerns than the reviews of future-defective files do. Figure 6.4 shows that the proportion of reviews that address traceability concern in clean files is higher than the corresponding proportion in the future-defective files with the differences of 7 and 21 percentage points (48%-41% and 60%-39%) for Qt 5.0 and Qt 5.1, respectively.

Summary: The reviews of files that will eventually have defects tend to be less rigorous and more frequently address evolvability concerns than the reviews of files without future defects do.

(RQ2) Do developers adapt their code reviewing practices in files that have historically been defective?

To address RQ2, we use post-release defects of Qt 5.0 as prior defects for Qt 5.1 and investigate the reviewing activities of changed files in Qt 5.1. Table 6.6 shows

Table 6.6. An overview of the review database to address RQ2.

	Qt 5.1			
	Risky	Normal	Risky & Future-Defective	Risky & Clean
Studied Files	1,168	11,629	206	962
Related Reviews	2,671	2,868	1,299	1,372
Review	93	93	44	49
Sample	(399 revisions)	(309 revisions)	(205 revisions)	(194 revisions)

an overview of the review database that we use to address RQ2. We conjecture that reviews of risky files should be more intensely scrutinized, have more team participation, and take a longer time to complete than the reviews of normal files.

We now present our empirical observations, followed by our general conclusion.

(RQ2-DA1) Quantitative Analysis of Code Review Activity

Observation 18 – Risky files tend to undergo less intense code reviews than normal files do. Table 6.7 shows that risky files tend to undergo reviews that have fewer iterations, shorter discussions, and more revisions without reviewer feedback than normal files do. Mann-Whitney U tests confirm that the differences are statistically significant ($p < 0.001$ for the number of iterations, discussion length metrics, and $p < 0.01$ for the proportion of revisions without feedback metric).

Table 6.7 also shows that the reviews of risky files tend to churn more during code review than those in normal files. Similar to Observation 13, changes with more churn during code review are likely to be controversial. For example, in review ID 29929,⁵ reviewers provide many suggested fixes and the author needs to revise the proposed changes many times before the change was accepted for integration.

Observation 19 – Risky files tend to be reviewed with less team participation than normal files. Table 6.7 shows that risky files tend to be reviewed by fewer

⁵<https://codereview.qt-project.org/#/c/29929>

Table 6.7. Results of one-tailed Mann-Whitney U tests ($\alpha = 0.05$) for code review activity metrics of risky and normal files.

Metric	Statistical Test	Relative Impact (%)
<i>Review Intensity</i>		
#Iterations [†]	Risky < Normal***	-28 ↓
Discussion Length [†]	Risky < Normal***	-26 ↓
Revisions without Feedback	Risky > Normal**	6 ↑
Churn during Code Review [†]	Risky > Normal***	7 ↑
<i>Review Participation</i>		
#Reviewers [†]	Risky < Normal***	-32 ↓
#Authors [†]	Risky < Normal***	-34 ↓
Non-Author Voters [†]	Risky < Normal***	-33 ↓
Review Disagreement	-	-
<i>Reviewing Time</i>		
Review Length [†]	Risky > Normal*	6 ↑
Response Delay	Risky > Normal***	15 ↑
Average Review Rate	Risky > Normal***	17 ↑

[†] The metrics are normalized by patch size.

Statistical significance: * $p < 0.05$, ** $p < 0.01$, *** $p < 0.001$.

reviewers and non-author voters than normal files do. Mann-Whitney U tests confirm that the differences are statistically significant ($p < 0.001$ for the number of reviewers and the number of non-author voters metrics). Moreover, there tend to be fewer authors in the reviews of risky files than those reviews of normal files. The low number of authors in risky files is also worrisome, since Observation 14 suggests that multiple authors revising the proposed change in a review tend to avoid problems that could lead to future defects.

Observation 20 – Risky files tend to undergo reviews that receive feedback more slowly and have faster review rate than normal files. Although Table 6.7 shows that risky files tend to undergo reviews that have a longer review length than normal files do, its relative impact is only 6%. On the other hand, we find that the reviews of risky files have a longer response delay and faster review rate than the reviews of normal files. Mann-Whitney U tests confirm that the

differences are statistically significant ($p < 0.001$ for the response delay and the average review rate metrics). This result is also worrisome, since observation 15 suggests that files that undergo reviews with longer response delay and faster review rate are likely to have defects in the future.

Table 6.7 shows that the metric with the largest relative impact is the number of authors (-34%), while the proportion of revisions without feedback and the review length metrics have the smallest relative impact (6%). Furthermore, we find that the number of iterations, the discussion length, the number of reviewers, and non-author voters metrics also have large relative impacts, ranging between -33% and -26%.

(RQ2-DA2) Qualitative Analysis of Concerns Raised during Code Review

Observation 21 – Risky files tend to more frequently have evolvability concerns addressed during code review than normal files do. Figure 6.5 shows that evolvability concerns are addressed in the reviews of risky files more often than normal files with a proportion of reviews that is 29 percentage points higher (87%-58%). The proportion of reviews that make structure changes shows an obvious difference of 32 percentage points (78%-46%) in risky and normal files. We observe that structure changes in the reviews of risky files relate to removing dead code, while changes in the reviews of normal files relate to re-implementing solutions using alternative approaches and small fixes for runtime errors.

Observation 22 – Risky files tend to more frequently have functionality concerns addressed during code review than normal files do. Figure 6.5 shows that there is a difference of 14 percentage points (47%-33%) between risky and normal files. The proportion of reviews that make check and logic changes shows a clear difference of 11 percentage points (19%-8%) between risky and normal files. We observe that the check changes relate to validating variable values, and logic

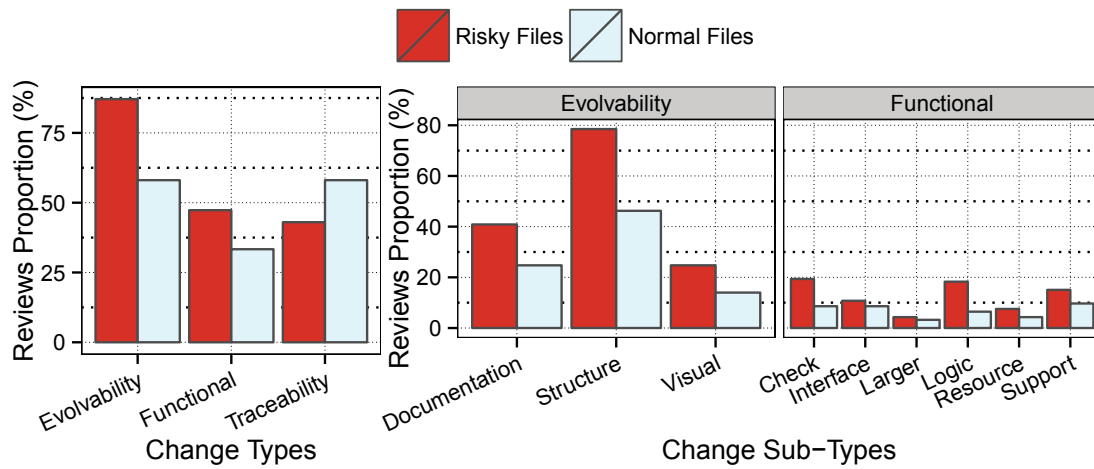


Figure 6.5. Distribution of change types that occurred during the code review of risky and normal files. The sum of review proportion is higher than 100%, since a review can contain many types of changes.

changes relate to updates to comparison expressions.

On the other hand, Figure 6.5 shows that normal files tend to address traceability concerns more often than risky files do. There is a difference of 15 percentage points (58% - 43%) between risky and normal files. Additionally, this concern is often addressed by authors. Moreover, we observe that changes to normal files are more rebased than risky files are.

Summary: Developers are not as careful when they review changes made to files that have historically been defective and often address the concerns of evolvability and functionality.

We study this phenomenon further to investigate the relationship between the code reviewing practices in the risky files and future defects. We use the same data analysis approaches, i.e., DA1 and DA2. We separate the risky files into two groups: 1) risky files that will eventually have defects (called *risky & future-defective files*) and 2) risky files that will eventually be defect-free (called *risky & clean files*) when Qt 5.1 is released. We also separate the randomly selected reviews of risky files to reviews of risky & future-defective files and reviews of

risky & clean files in order to compare concerns addressed during code review. Table 6.6 shows an overview of the review database that we use to perform this study.

Observation 23 – Risky & future-defective files tend to be less carefully reviewed than risky & clean files. Table 6.8 shows that risky & future-defective files tend to undergo less intense code review with less team participation, i.e., fewer iterations, shorter discussions, more revisions without reviewer feedback, fewer reviewers, authors, and non-author voters than risky & clean files. Mann-Whitney U tests confirm that the differences are statistically significant ($p < 0.001$ for all metrics of the review intensity and participation dimensions). Moreover, we find that risky & future-defective files tend to undergo reviews that have a longer response delay and faster review rate than risky & clean files. Mann-Whitney U tests confirm that the differences are statistically significant ($p < 0.001$ for the response delay and the average review rate metrics). These results indicate that changes made to the risky files that are also defective are reviewed with less intensity, less team participation, faster review rate, and receive slower feedback than the risky files that are defect-free.

Table 6.8 shows that the metric with the largest relative impact is the response delay (34%). On the other hand, the churn during code review metric has the smallest relative impact (7%). We also find that the number of iterations, the discussion length, the number of reviewers, authors, and non-author voters metrics have a large negative relative impacts ranging between -31% to -27%, and the average review rate metric has a large positive relative impact (25%).

Observation 24 – Risky & future-defective files tend to have structure, visual representation, and check concerns addressed more often during code reviews than risky & clean files do. Figure 6.6 shows that for evolvability changes, the proportion of reviews in risky & future-defective files that make structure and visual representation changes is higher than the corresponding proportion in risky & clean files. There are differences of 11 and 10 percentage points (84%-

Table 6.8. Results of one-tailed Mann-Whitney U tests ($\alpha = 0.05$) for code review activity metrics of risky & future-defective files and risky & clean files.

Metric	Statistical Test	Relative Impact (%)
<i>Review Intensity</i>		
#Iterations [†]	Risky & Future-defective < Risky & Clean***	-27 ↓
Discussion Length [†]	Risky & Future-defective < Risky & Clean***	-29 ↓
Revisions without Feedback	Risky & Future-defective < Risky & Clean***	10 ↑
Churn during Code Review [†]	Risky & Future-defective > Risky & Clean*	7 ↑
<i>Review Participation</i>		
#Reviewers [†]	Risky & Future-defective < Risky & Clean***	-27 ↓
#Authors [†]	Risky & Future-defective < Risky & Clean***	-31 ↓
Non-Author Voters [†]	Risky & Future-defective < Risky & Clean***	-30 ↓
Review Disagreement	-	-
<i>Reviewing Time</i>		
Review Length [†]	-	-
Response Delay	Risky & Future-defective > Risky & Clean***	34 ↑
Average Review Rate	Risky & Future-defective > Risky & Clean***	25 ↑

[†] The metrics are normalized by patch size.

Statistical significance: * $p < 0.05$, ** $p < 0.01$, *** $p < 0.001$.

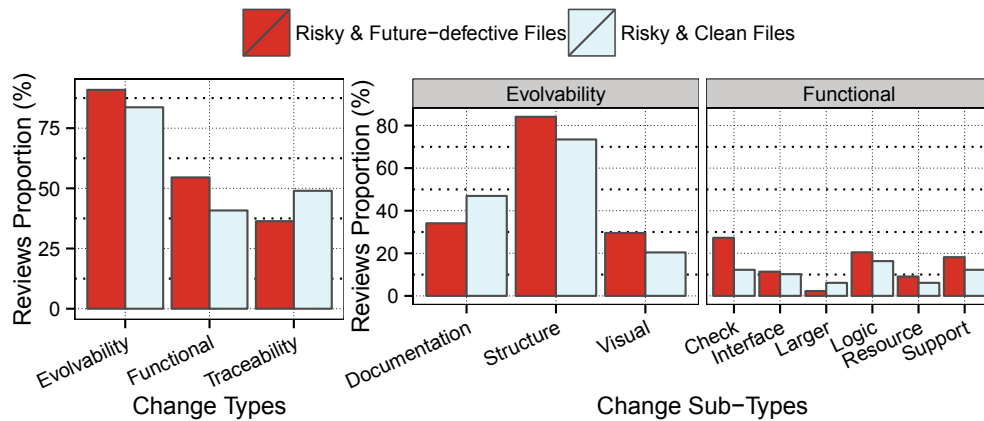


Figure 6.6. Distribution of change types that occurred during the code review of risky & future-defective and risky & clean files. The sum of review proportion is higher than 100%, since a review can contain many types of changes.

73% and 30%-20%) in structure and visual representation changes, respectively. For functionality changes, the proportion of reviews that make check changes shows an obvious difference of 14 percentage points (55%-41%) between risky & future-defective files and risky & clean files. We also observe that reviewers inspire evolvability changes in risky & future-defective files more often than risky & clean files. The proportion of reviews is 43% and 31% in risky & future-defective files and risky & clean files, respectively. However, few functionality changes are inspired by reviewers in the reviews of risky & future-defective files. The proportion of reviews ranges between 0% - 11% (average of 5%) in risky & future-defective files and between 4% - 10% (average of 5%) in risky & clean files for each type of functionality changes. This finding suggests that reviewers do not focus much on functionality during code review of risky files.

Summary: Files that have historically been defective and will eventually have defects tend to undergo less rigorous code reviews that more frequently address evolvability concerns than the files that have historically been defective, but will eventually be defect-free.

6.4. Discussion

In this section, we discuss the broader implications of our empirical observations.

6.4.1. Review Intensity

Observations 13 and 18 have shown that the reviews of changes made to clean and normal files often have longer discussions and more iterations than the reviews of future-defective and risky files do. Prior work reports that the focus of MCR discussion has shifted from defect-hunting to group problem-solving [6, 97, 128]. Examining a patch in multiple review iterations would likely uncover more problems than a single review iteration [91]. Hence, the reviews that have long discussions and many iterations seem to improve the patch and avoid problems that could lead to future defects.

6.4.2. Review Participation

Observations 14 and 19 have shown that the reviews of changes made to clean and normal files often have more participants than the reviews of future-defective and risky files do. Corresponding to Linus' law [96], our findings suggest that code reviews should be performed by multiple reviewers to reduce the likelihood of having future defects.

Several studies also suggest that patches should be reviewed by at least two developers to maximize the number of defects found during the review, while minimizing the reviewing workload on the development team [91, 97, 106]. In this study, we measure the number of reviewers that is normalized by patch size (or the number of reviewers per line). We find that the number of reviewers per line also shares an inverse relationship with defect-proneness. For example, we find that at the median values, the clean files typically undergo reviews that have two reviewers for every 20 lines, while the future-defective files typically

undergo reviews that have one reviewer for every 20 lines. Hence, development teams should take the number of reviewers per line into consideration when making integration decisions. The MCR tools also should be configured to require reviewers according to the patch size and the number of reviewers per line.

6.4.3. Review Speed

Observations 15 and 20 have shown that each review iteration of clean and normal files is performed slower than future-defective and risky files. Similar to the traditional code reviewing practices [35, 60], our findings suggest that reviewers will be able to uncover more problems hidden in a patch if they perform a careful code examination with an appropriate code reading rate.

6.4.4. Reviewing Concerns

Observations 16, 17, 21, and 22 have shown that the reviews of changes made to future-defective files and risky files focus on evolvability concerns rather than functional fixes. Indeed, functionality concerns are still rarely addressed during code review. Although MCR practices seem to focus on improving maintainability, our prior findings suggest that the rigor of the reviewing process that is applied to a source code file throughout a development cycle could help development teams to avoid future defects.

6.4.5. Code Review of Risky Files

Observations 23 and 24 have shown that risky & future-defective files tend to undergo less rigorous code reviews that more frequently address evolvability concerns than the risky & clean files. One contributing reason for the less careful review of risky files could be that it is difficult for practitioners to determine the risk of code changes during code review [117]. Despite the difficulty of estimating

the risk, the Gerrit code review tool did not provide information about the history of those changed files in a review. Moreover, we also observe that risky files were changed more frequently than normal files during the development cycle of Qt 5.1. For example, 425 risky files that are related to the `widgets` components were impacted by six patches on average. According to a large amount of changes must be reviewed before the integration, it could be a tedious task for a developer to accurately estimate risk for every change without additional information. Therefore, a supporting tool could help practitioners notice such risky files to perform code review more rigorously when necessary.

6.5. Threats to Validity

We now discuss potential threats to validity of our study.

6.5.1. Construct validity

We study the code reviewing practices of files derived from a set of reviews instead of studying a single review because uncovering defects is not the sole intent of MCR practices [6] and the number of defects found during code review is not explicitly recorded in an MCR tool [97]. Therefore, we use the collection of code review activity of files during the development of a release to evaluate the impact that the MCR practices have on software quality.

The change classification method that was conducted by the author who is not involved in the code review process of the studied system. The results of manual classification by the team members might be different. However, our classification schema is derived from prior work [16, 70] and the comments that we use to classify code reviews are originally written by team members who participated in the code review process. Furthermore, we repeatedly label changes several times before perform our study to ensure the uniformity of the change classification,

and a subset of change classification results is verified by the second author of our paper [121].

6.5.2. Internal validity

Some of our code review activity metrics are measured based on heuristics. For example, we assume that the review length is the elapsed time between when a patch has been uploaded and when it has been approved for integration. However, there are likely cases where reviewers actually examined a patch for a fraction of this review length. Unfortunately, reviewers do not record the time that they actually spent reviewing a patch. Since there is a limitation of measuring the actual code review activity, we must rely on heuristics to recover this information. Furthermore, our rationales for using metrics are supported by prior work [81, 91, 92, 100, 128].

6.5.3. External validity

Although the Qt system is an open source project that actively assesses software changes through an MCR tool, the analysis of the studied dataset does not allow us to draw conclusions for all open source projects. Since the code review process of MCR is a relatively new initiative, finding systems that satisfy our selection criteria is a challenge (*cf.* Section 6.2.1). Naggapan *et al.* also argue that if care is not taken when selecting which projects to analyze, then increasing the sample size does not actually contribute to the goal of increased generality [83]. Nonetheless, additional replication studies are needed to generalize our results.

6.6. Summary

Although Modern Code Review (MCR) is now widely adopted in both open source and industrial projects, the impact of MCR practices on software quality is still

unclear. In this study, we comparatively study the MCR practices in defective and clean files, i.e., 1) files that will eventually have defects (called future-defective files), and 2) files that have historically been defective (called risky files). Due to the human-intensive nature of code reviewing, we decide to perform an in-depth study on a single system instead of examining a large number of projects. Using data collected from the Qt open source system, we empirically study 11,736 reviews of changes to 24,486 files and manually examine 558 reviews. The results of our study indicate that:

- The code review activity of future-defective files tends to be less intense with less team participation and with a faster rate of code examination than the reviews of clean files (Observations 13 to 15).
- Developers more often address concerns about: 1) documentation and structure to enhance evolvability, and 2) checks, logic, and support to fix functionality issues, in the reviews of future-defective files than the reviews of clean files (Observations 16 to 17).
- Despite their historically defective nature, the code review activity of risky files tends to be less intense with less team participation than files that have historically been defect-free. Reviews of risky files also tend to receive feedback more slowly and have a faster review rate than the reviews of normal files (Observations 18 to 20).
- In the reviews of risky files, developers address concerns about evolvability and functionality more often than the reviews of normal files do (Observations 21 to 22).
- Risky files that will have future defects tend to undergo less careful reviews that more often address concerns about evolvability than the reviews of risky files without future defects do (Observations 23 to 24).

Our results suggest that rigorous code review could lead to a reduced likelihood of future defects. Files that have historically been defective should be given more careful attention during code review, since such files are more likely to have future defects [\[41\]](#).

Identifying Characteristics of Patches with Poor Reviewer Involvement

An earlier version of the work in this chapter appears in the Springer Journal of Empirical Software Engineering (EMSE) [122].

In the previous chapter, we find that the amount of review participation, review intensity, and reviewing time that are applied throughout the development cycle is associated with the likelihood of having defects in the future. However, little is known about which factors influence reviewer involvement in the MCR processes. Hence, in this chapter, we further investigate the characteristics of patches that suffer from poor reviewer involvement. Specifically, we study the poor reviewer involvement in terms of (1) participation, i.e., patches that do not attract reviewers, (2) intensity, i.e., patches that are not discussed, and (3) reviewing time, i.e., patches that receive slow initial feedback. Our case study results demonstrate that the amount of reviewer involvement in the past is a significant indicator of patches that will suffer from poor reviewer involvement. Moreover, we observe that the description length of a patch shares a relationship with the likelihood

of receiving poor review participation or discussion, while patches that introduce new features are likely to receive slow initial feedback. Our findings suggest that the patches with such characteristics should be given more attention in order to increase review participation, which will likely lead to a more effective review process.

7.1. Introduction

Recent research presents reviewer involvement as a key aspect of MCR practices. Rigby *et al.* report that the efficiency and effectiveness of code reviews are most affected by the amount of reviewers [99]. Recent work finds that reviewer involvement metrics (e.g., the number of involved developers and the number of comments in a review) are associated with the quality of the code [65]. McIntosh *et al.* report that a lack of reviewer involvement can have a negative impact on software quality [74]. Our prior work also finds that the amount of reviewer involvement that is applied to files throughout the development cycle is associated with the likelihood of having future defects [121]. Despite the importance of reviewer involvement, little is known about the factors that influence reviewer involvement in MCR processes.

A good understanding of the factors that influence reviewer involvement helps teams create mitigation strategies to avoid such poor involvement which in turn would help them avoid future quality problems. Hence, in this chapter, we set out to investigate the characteristics of patches that suffer from poor reviewer involvement. In particular, we focus on the characteristics of patches that: (1) do not attract reviewers, (2) are not discussed, and (3) receive slow initial feedback. We measure patch characteristics using 20 patch and MCR process metrics that are grouped along five dimensions, i.e., patch properties, reviewer involvement history, past involvement of an author, past involvement of reviewers, and review environment dimensions. To investigate the relationship between the patch

characteristics and the likelihood that a patch will suffer from poor reviewer involvement, we use contemporary regression modelling techniques that relax the requirement of a linear relationship between explanatory variables and the response, enabling a more accurate and robust fit of the data [48]. Through a case study of 196,712 reviews spread across the Android, Qt, and OpenStack open source systems, we address the following three research questions:

(RQ1) Which patch characteristics share a relationship with the likelihood of a patch not attracting reviewers?

Motivation. Recent work has found that patches with low participation of reviewers are undesirable and have a negative impact on code quality [12, 73, 74, 121]. However, patches might be ignored during the MCR process [19, 87]. Moreover, some patches are merged into upstream VCS repositories even though they do not have any participants involved with their reviews apart from the patch author. For example, in the review ID 29921 in the Qt system, no third-party reviewers participated in the review by neither voting a score or posting a message, although the patch author had invited a reviewer.¹ Hence, we set out to better understand the characteristics of the patches that did not attract reviewers.

Result. We find that the number of reviewers of prior patches, the number of days since the last modification of the patched files share a strong increasing relationship with the likelihood that a patch will attract reviewers. The description length is also a strong indicator of a patch that is likely to not attract reviewers.

(RQ2) Which patch characteristics share a relationship with the likelihood of a patch not being discussed?

Motivation. Careful consideration of the implications of changes improves their overall quality prior to integration [6, 121, 128]. Recent studies find

¹<https://codereview.qt-project.org/#/c/29921/>

that the proportion of changes without review discussion shares a positive relationship with the incidence of both post-release defects [74], and software design anti-patterns [81]. A review that simply assigns a review score without any suggestion for improvement nor discussion provides little return on code review investment. However, it is not known whether there are factors that make a review more susceptible to a lackluster discussion.

Result. We find that the description length, churn, and the discussion length of prior patches share an increasing relationship with the likelihood that a patch will be discussed. We also find that the past involvement of reviewers shares an increasing relationship with the likelihood that a patch will be discussed. On the other hand, the past involvement of an author shares an inverse relationship with the likelihood that a patch will be discussed.

(RQ3) Which patch characteristics share a relationship with the likelihood of a patch receiving slow initial feedback?

Motivation. A well-functioning code review process should yield responses to a new review request in a timely manner in order to avoid potential problems in the development process [17]. For example, due to continuous software development practices [36], it is possible that if a patch receives slow initial feedback, it can become outdated, requiring updates to be re-applied (and possibly re-implemented) to the latest version of the system. Prior work also suggest that the earlier that a patch is reviewed, the lower the risk of deeply embedded defects [100,121]. To better understand patches that have a long feedback delay, we investigate the characteristics of patches that received slow initial feedback.

Result. We find that the feedback delay of prior patches shares a strong relationship with the likelihood that a patch will receive slow initial feedback. Furthermore, a patch is likely to receive slow initial feedback if its purpose is to introduces new features.

Our results lead us to conclude that the past reviewer involvement, the description length, the number of days since the last modification of files, the past involvement of an author, and the past involvement of reviewers share a strong relationship with the likelihood that a patch will suffer from poor reviewer involvement. Our results highlight the need for patch submission policies that monitor these factors in order to help development teams improve reviewer involvement in MCR processes.

7.1.1. Chapter Organization

The remainder of the chapter is organized as follows. Section 7.2 describes the design of our empirical study, while Section 7.3 presents the results with respect to our three research questions. Section 7.4 discusses the broader implication of our results. Section 7.5 discloses the threats to the validity of our empirical study. Finally, Section 7.6 summarizes this study.

7.2. Case Study Design

In this section, we describe the studied systems and present the data preparation, model construction, and model analysis approaches that we use to address our research questions.

7.2.1. Studied Systems

In order to address our research questions, we perform an empirical study of software systems that actively use MCR for the code review process, i.e., examine and discuss software changes through a code review tool. We use the review datasets of Android, Qt, and OpenStack systems which are provided by Hamasaki *et al.* [46]. The review datasets describe patch information, reviewer scoring, the involved personnel, and the discussion history. All three systems

Table 7.1. Overview of the studied systems.

System	Overview		
	Period	Total Patches	Avg. #Patches/Yr
Android	2008/10 - 2014/12 (6 Years)	51,721	8,620
Qt	2011/5 - 2014/12 (4 Years)	99,286	33,095
OpenStack	2011/7 - 2014/12 (4 Years)	136,343	45,447

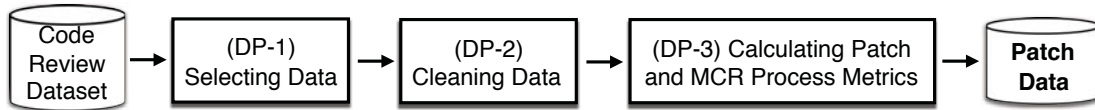


Figure 7.1. An overview of our data preparation approach.

have been performing code reviews through the Gerrit code review tool for an extended period of time, i.e., more than three years (see Table 7.1). Table 7.1 also shows that the Android, Qt, and OpenStack systems have a large number of patches that were reviewed through the Gerrit code review tool in each year, suggesting that our studied systems actively use MCR.

The Android open source system² is an operating system for mobile devices that is developed by Google. Qt³ is a cross-platform application and UI framework that is developed by the Digia corporation. OpenStack⁴ is an open-source software platform for cloud computing that is developed by many well-known companies, e.g., IBM, VMware, and NEC.

7.2.2. Data Preparation

To perform our empirical study, we classify patches based on their reviewer involvement and extract patch metrics. Figure 7.1 provides an overview of our data preparation approach. We describe the details of our data preparation approach below.

²<https://source.android.com/>

³<http://qt-project.org/>

⁴<http://www.openstack.org/>

Selecting Data

To truly understand reviewer involvement, we exclude patches that do not satisfy the following criteria:

1. A patch must be submitted during the period when the studied systems actively uses MCR tools.
2. A patch must not be related to VCS bookkeeping activity, such as branch merging.

For criterion 1, we identify active periods (i.e., years) of MCR usage by computing an active rate, i.e., the number of submitted patches in a year relative to the total number of submitted patches in the whole period that is captured in the datasets. For our analysis, we select the years that have an active rate larger than 10%. Note that we do not have any gaps in our data, i.e., we find that after the first year that the system has an active rate above 10%, each following year also has an active rate above 10%. We focus only on patches that are submitted during the active MCR period because we need to ensure that the low MCR involvement are due to patch characteristics and not initial MCR experimentation (like the activity during initial adoption of MCR tools). For criterion 2, we filter out patches that are related to branch merging because such patches are used to perform VCS bookkeeping for other patches that have already been revised and integrated. Therefore, such patches generally have little reviewer involvement, since the earlier patches have already been reviewed.

Cleaning Data

After selecting patches, we clean the data in order to ensure the accuracy of study results. To do so, we (1) merge the duplicate accounts of a reviewer in the code review systems, and (2) remove auto-generated messages. We describe our approaches below.

Merging the duplicate accounts of a reviewer. Similar to Issue Tracking Systems and email discussion threads, Gerrit uses an email address to uniquely identify users. It is possible that a reviewer may have multiple review accounts in the MCR tool due to email aliases of the reviewer. To merge the duplicate accounts, we identify the email aliases using the approaches of Bird *et al.* [20]. For each reviewer account, we search for the accounts that have a similar name or a similar email name (excluding the email domain) using the generalized Levenshtein edit distance [129]. We then manually inspect potential duplicates, i.e., those with a Levenshtein edit distance below 0.1.

Removing auto-generated messages. Since we will use the messages that are posted in the review discussion thread to measure the involvement of reviewers, we need to remove the messages that are left by automated quality gating tools (e.g., static code analyzers) or written by the patch author. We identify the messages that are posted by tools using the accounts of bots in the studied systems. As suggested by [82], we mark the account named “Deckard Autoverifier” as a bot for the Android system. By studying the MCR processes of the Qt and OpenStack systems, we find that the Qt system has Continuous Integration (CI) and Early Warning System (EWS) systems,⁵ and the OpenStack system has the Jenkins and Zuul automated testing systems.⁶

Calculating Patch and MCR Process Metrics

We use 20 patch and MCR process metrics to examine the patches that will suffer from poor reviewer involvement. Our metrics are grouped into five dimensions: (1) patch properties, (2) history, (3) past involvement of an author, (4) past involvement of reviewers, and (5) review environment. Table 7.2 provides the conjecture and the motivating rationale for each of the studied patch and MCR process metrics. Below, we describe the calculation for each of our metrics.

⁵https://wiki.qt.io/Qt_Contribution_Guidelines

⁶<http://docs.openstack.org/infra/manual/developers.html#peer-review>

Table 7.2. A taxonomy of patch metrics.

Metric	Conjecture	Rationale
<i>Patch Properties Dimension</i>		
Churn	The larger the churn is, the more likely that the patch receives reviewer involvement.	Large patches may need more effort to review [78, 99, 100].
Number of Modified Files	The more files that are changed in this patch, the more likely that the patch will suffer from poor reviewer involvement.	Patches where their changes scatter across a large number of files or directories may need more effort to review. Finding reviewers who have knowledge for such changes is difficult as well. Therefore, it is more likely that the patch will suffer from poor reviewer involvement.
Number of Modified Directories	The more directories that are impacted by this patch, the more likely that the patch will suffer from poor reviewer involvement.	
Entropy	The more scattered the changes in this patch are, the more likely that the patch will suffer from poor reviewer involvement.	
Description Length	The longer description in the patch, the less likely that the patch will suffer from poor reviewer involvement.	Patches with a descriptive subject and a well explained change log message would be able to draw the attention of reviewers [101].
Purpose	A patch that introduces new functionality is more likely to receive slow initial feedback than a patch for another purposes.	Patches that introduce new features may require more effort to examine than patches for other purposes.
<i>History Dimension</i>		
Number of Days since the Last Modification	A patch containing files that have been recently changed is more likely to receive responsive reviewer involvement.	Recently changed files could be the files on which developers are currently working (i.e., more knowledgeable).

Continued on next page

Table 7.2 A taxonomy of patch metrics. *Continued from previous page*

Metric	Conjecture	Rationale
Total Number of Authors	The more developers who have written patches made to the modified files, the more likely that the patch receives active reviewer involvement.	A reviewer is likely to be one of the authors of a frequently changed file.
Number of Prior Defects	A patch containing files that have many defects is more likely to receive active reviewer involvement.	Files that have historically been defective may require additional attention during the code review process [121].
Number of Reviewers of Prior Patches	The more reviewers who have examined prior patches, the more likely the patch receive active involvement.	Files that have been previously examined by many reviewers would have the likelihood that one of those reviewers is a reviewer of this patch.
Discussion Length of Prior Patches	The longer discussion that the modified files have received in prior patches, the more likely that the patch receive active involvement.	Files that have received long discussion in prior patches could be complicated. Hence, they may require additional attention during the code review process.
Feedback Delay of Prior Patches	The longer the feedback delay in prior patches is, the more likely that the patch will suffer from poor reviewer involvement.	Files that often receive slow initial feedback can be those with less priority than other files. Hence, they may receive little reviewer involvement.

Continued on next page

Table 7.2 A taxonomy of patch metrics. *Continued from previous page*

Metric	Conjecture	Rationale
<i>Past Involvement of an Author Dimension</i>		
Number of Prior Patches of an Author	The more prior patches that the author has either written or examined, the more likely that the patch receives reviewer involvement.	Patches written by inexperienced authors are more likely to receive little reviewer involvement, since the authors are not familiar with the system and may not know who should be invited to review the changes [23].
Recent Patches of an Author	The more the recent patches of an author are, the more likely that the patch receives reviewer involvement.	
Number of Directory Patches of an Author	The more the directory patches of an author are, the more likely that the patch receives review involvement.	
<i>Past Involvement of Reviewers Dimension</i>		
Number of Prior Patches of Reviewers	The more prior patches that the reviewers have either written or examined, the more likely that the patch receives reviewer involvement.	Patches reviewed by experienced reviewers are very likely to receive prompt initial feedback and long discussion, since such reviewers have a good understanding and a strong familiarity of the context in which a change is being made [15, 98, 100, 124].
Recent Patches of Reviewers	The more the recent patches of reviewers are, the more likely that the patch receives reviewer involvement.	
Number of Directory Patches of Reviewers	The more the directory patches of reviewers are, the more likely that the patch receives reviewer involvement.	

Continued on next page

Table 7.2 A taxonomy of patch metrics. *Continued from previous page*

Metric	Conjecture	Rationale
<i>Review Environment Dimension</i>		
Overall Workload	The more the overall workload of the system is, the more likely that the patch will suffer from poor reviewer involvement.	Reviewers can be burdened with a large workload. Therefore, it is more likely that patches will receive little reviewer involvement if they are submitted at a time when a system has a large review workload [13, 101].
Directory Workload	The more the directory workload is, the more likely that the patch will suffer from poor reviewer involvement.	

Patch properties. The patch properties dimension measures the change and the information of a patch. To measure the change of a patch, we adopt the change metrics from prior work [58]. Churn measures the number of lines added to and removed from modified files. Number of modified files and directories measure the dispersion of a change. Change entropy measures the distribution of modified code across each modified file. Similar to prior work [51], we measure the entropy of a change C as described below:

$$H(C) = -\frac{1}{\log_2 n} \sum_{k=1}^n (p_k \times \log_2 p_k) \quad (7.1)$$

where n is the number of files included in a patch, p_k is the proportion of change C that impacts file k . The larger the entropy value, the more dispersed that a change is among files.

We also add description length and purpose metrics into this dimension to measure the information that authors provide for the patch. The description length measures how many words an author uses to describe a patch. The purpose indicates the change purpose of a patch. We define the purpose category similar to prior work [50, 80], i.e., documentation, bug fixing, and feature introduction. 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. A similar approach was used to classify patches in prior studies [58, 61, 73].

History. The history dimension measures the activity of prior patches that modified the same files as the patch under examination. Nagappan *et al.* report that the time window of history metrics may have an impact on the prediction models [86]. Therefore, we select a time window based on the development activities of the studied systems. To do so, we study the development cycle by observing the release dates of the studied systems.⁷ We find that the studied systems often

⁷ https://en.wikipedia.org/wiki/Android_version_history, https://en.wikipedia.org/wiki/Android_version_history

release a new version every six months. Hence, we measure the history metrics using the six-month period prior to the patch's date of submission. To measure the history metrics for each patch, we focus our analysis on the reviewing activities of patches that (1) occurred on the same branch as the studied patch, and (2) originated on other branches, but have been merged into the same branch as the studied patch. The number of days since the last modification measures how long it has been since the files that were modified in the patch were last modified. The total number of authors counts how many people have submitted patches that impact the same files as the patch under examination. The number of prior defects counts the number of prior bug-fixing patches that impact the same files as the patch under examination.

Furthermore, we adopt the hypothesis of prior work to estimate past tendencies [115]. We measure the past tendency of reviewer involvement using three metrics, i.e., the number of reviewers, discussion length, and feedback delay of prior patches that have been applied to the same files as the files in the patch under examination. The number of reviewers of prior patches measures the median number of reviewers who have posted messages or a reviewing score in the reviews of prior patches that impact the same files as the patch under examination. The discussion length of prior patches measures the median number of messages that are posted in the reviews of prior patches that impact the same files as the patch under examination. The feedback delay of prior patches measures the median of feedback delays that the reviews of prior patches had received. We use the median value because we find that the distributions of the history data do not follow normal distribution (i.e., the p -values of Shapiro-Wilk tests are lower than 0.05 for all of the studied patches).

Past involvement of an author. The past involvement of an author dimension measures the activity in which an author has been involved before making the patch under examination. To calculate the past involvement of an author metrics,

[org/wiki/List_of_Qt_releases](https://wiki.qt.org/List_of_Qt_releases), <https://wiki.openstack.org/wiki/Releases>

we use the same approach as the history dimension to collect the activity in which an author has been involved. The number of prior patches of an author counts the number of submitted or reviewed patches by the author prior to the patch under examination. The recent patches of an author is a variant of the number of prior patches of an author, which is weighted by the age of the prior patches. The more recent patches are given a higher weight than the less recent ones. Similar to prior work [58], we measure the recent patches for an author using a calculation of $RC = \sum_{m \in M} \frac{N_c}{m}$, where N_c is the number of patches that have been submitted or reviewed by the author in the past month m in the time window M (i.e., six-month period). Similar to prior work [58], we also measure a higher level of experience for an author using the number of directory patches of an author metric. The number of directory patches of an author measures how many prior patches modify code in the same directories as the patch under examination, and were submitted or reviewed by the author.

Past involvement of reviewers. The past involvement of reviewers dimension measures the activity that the patch reviewers have been involved with prior to the patch under examination. Similar to past involvement of an author dimension, we measure the number of prior patches, recent patches, and the number of directory patches of reviewers metrics.

Review environment. The review environment dimension measures the code review activity that occurred during the same period as the patch being submitted. To measure review environment metrics, we use a 7-day period prior to the time that a patch is submitted. We select the 7-days period as our time window because we find that the time from patch submission to review completion is shorter than 3, 4, and 11 days for 75% of the reviews in the Android, Qt, and OpenStack systems, respectively. The overall workload metric counts how many patches are submitted to the code review tool. The directory workload metric counts how many prior patches modify code in the same directories as the patch under examination.

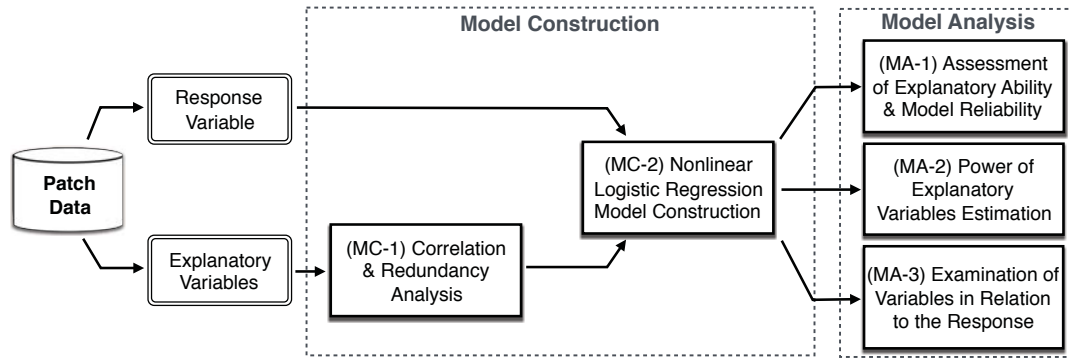


Figure 7.2. An overview of our model construction and analysis approaches.

7.2.3. Model Construction

We build logistic regression models to determine the likelihood of a patch suffering from poor reviewer involvement. Logistic regression models are commonly used to study how explanatory variables are related to a dichotomous response variable. In our study, we use our patch and MCR process metrics as explanatory variables, while the response variable is assigned the value of TRUE if a patch suffered from poor reviewer involvement, and FALSE otherwise.

We adopt the model construction and analysis approaches of Harrell Jr. [48, p. 79] to allow nonlinear relationships between explanatory and response variables to be modelled. These techniques can enable a more accurate and robust fit of the data, while carefully considering the potential for overfitting (i.e., a model is too specifically fit to the training dataset to be applicable to other datasets). An overfit model will overestimate the performance of the model and exaggerate spurious relationships between explanatory and response variables.

Figure 7.2 provides an overview of the three steps in our model construction approach. To facilitate future research and replication study, we provide an example R script of model construction and analysis in Appendix A. We briefly describe each step in our approach below.

(MC-1) Correlation & Redundancy Analysis

We remove highly correlated explanatory variables before constructing our models to reduce the risk that those correlated variables interfere with our interpretation of the models. We measure the correlation between explanatory variables using Spearman rank correlation tests (ρ), which are resilient to data that is not normally distributed. We then use a variable clustering analysis approach [105] to construct a hierarchical overview of the inter-variable correlation and select one explanatory variable from each cluster of highly-correlated variables, i.e., $|\rho| > 0.7$ [66].

We also check for redundant variables (i.e., variables that do not offer a unique signal with respect to the other variables). We use the `redun` function in the `rms` R package [49] to detect redundant variables. However, we find that none of the explanatory variables that survive our correlation analysis are redundant.

(MC-2) Nonlinear Logistic Regression Model Construction

Before constructing our models, we carefully relax the linearity of the modeled relationship between explanatory and response variables, while being mindful of the risk of overfitting. To do so, we only allocate additional degrees of freedom (i.e., the number of regression parameters) to the explanatory variables that have more potential for sharing a nonlinear relationship with the response variable. We measure the potential for nonlinearity in the relationship between explanatory and response variables using a calculation of the Spearman multiple ρ^2 . We then allocate the degrees of freedom to explanatory variables according to their Spearman multiple ρ^2 values, i.e., variables with larger ρ^2 values are allocated more degrees of freedom. Nevertheless, we limit the maximum degrees of freedom that we allocate to any given explanatory variable to five in order to minimize the risk of overfitting [48, p. 23].

After removing the highly correlated and allocating the degrees of freedom to the surviving explanatory variables, we fit our logistic regression models to the data. We use the restricted cubic splines of the `rcs` function in the `rms` R package [49] to fit the allocated degrees of freedom to the explanatory variables.

7.2.4. Model Analysis

Once the logistic regression model has been constructed, we analyze the model in order to understand the relationship between the explanatory variables (i.e., patch and MCR process metrics) and the response variable (i.e., whether a patch had reviewer involvement or not). Figure 7.2 shows our three steps of model analysis. We describe below each step of our model analysis approach.

(MA-1) Assessment of Explanatory Ability & Model Reliability

We measure how well a model can discriminate between the potential response using the Area Under the receiver operating characteristic Curve (AUC) [47]. Furthermore, we evaluate the reliability of our models, since AUC can be too optimistic if the model is overfit to the dataset. Similar to prior work [74], we estimate the optimism of AUC using a bootstrap-derived approach [32]. Small optimism values indicate that the model does not suffer from overfitting.

(MA-2) Power of Explanatory Variables Estimation

We first measure the power of the explanatory variables that contribute to the fit of our models using Wald statistics. We use the `anova` function in the `rms` R package [49] to estimate the explanatory power (Wald χ^2) and the statistical significance (p -value) of each explanatory variable in our models. The larger the Wald χ^2 value, the larger the explanatory power of that variable.

Examination of Variables in Relation to the Response

To better understand the direction and shape of these relationships, we examine the explanatory variables in relation to the odds value produced by our models. We use the `Predict` function in the `rms` R package [49] to plot changes in the estimated odds while varying one explanatory variable under test and holding the other explanatory variables at their median values.

In addition, we estimate the partial effect that explanatory variables have on the response using odds ratio [48, p. 220]. Odds ratio indicates the change to the likelihood of a patch that will suffer from poor reviewer involvement when the value of an explanatory variable under study increases. The larger the odds ratio is, the larger the partial effect that the explanatory variable has on the likelihood of a patch suffering from poor reviewer involvement. We analyze the relative percentage that the odds has changed corresponding to the changed value of the explanatory variable, while holding the other explanatory variables at constant values. Using the `summary` function in the `rms` R package [49], the partial effect is estimated based on the odds difference of the inter-quartile range for continuous variables, and the odds difference between each category value and the mode (i.e., the most frequently occurring category) for categorical variables. A positive partial effect indicates an increasing relationship between the explanatory variable and the response, while a negative partial effect indicates an inverse relationship. The magnitude of a partial effect indicates the amount that the odds value in our models will change according to the shifted value of the explanatory variable.

7.3. Case Study Results

In this section, we present the results of our study with respect to our research questions. Table 7.1 provides a summary of the patch and review data that we selected according to our data preparation approach. Table 7.3 shows distribu-

tions of data for the patch metrics that we measure in the studied datasets. For each research question, we discuss our: (a) model construction procedure and (b) model analysis results.

(RQ1) Which patch characteristics share a relationship with the likelihood of a patch not attracting reviewers?

In the Gerrit process, some patches can be merged into upstream VCS repositories even though these patches do not have any participant apart from the patch author. For example, in the review ID 29921 in the Qt system, no third-party reviewers participated in the review by neither voting a score or posting a message, although the patch author had invited a reviewer.⁸ Furthermore, McIntosh *et al.* have found that the number of patches that do not have reviewers providing feedback to the reviews shares a relationship with defect-proneness [73]. Hence, to address our RQ1, we identify the patches that do not attract reviewers by counting the number of unique reviewers who participated in a review by either posting a message or assigning a reviewing score. We classify patches into two categories — those that attract at least one reviewer and those that do not.

Table 7.4 provides the number of patches from our patch classification. Below, we present and discuss the results of our model construction and analysis.

(RQ1-a) Model Construction

According to our model construction approach (*cf.* Figure 7.2), the response variable is set to TRUE if a patch is not reviewed by another developer and FALSE otherwise. We use our patch and MCR process metrics that are described in Table 7.2 as explanatory variables. However, we did not use the past involvement of reviewers metrics, since past involvement of reviewers cannot be measured in the

⁸<https://codereview.qt-project.org/#/c/29921/>

Table 7.3. Descriptive statistics of the studied patch metrics. Histograms are in a log scale.

		Android	Qt	OpenStack			Android	Qt	OpenStack
Patch Properties Dimension					History Dimension				
Churn	1st Qu.	5.00	4.00	5.00	#Prior defects	1st Qu.	0.00	0.00	0.00
	Median	20.00	15.00	21.00		Median	1.00	1.00	3.00
	Mean	4434.00	603.00	767.00		Mean	5.69	4.01	10.48
	3rd Qu.	96.00	63.00	90.00		3rd Qu.	4.00	4.00	10.00
	Histogram					Histogram			
#Modified files	1st Qu.	1.00	1.00	1.00	#Days since the last modification	1st Qu.	1.47	1.26	1.18
	Median	2.00	2.00	2.00		Median	11.31	8.19	5.45
	Mean	27.02	10.81	4.81		Mean	55.11	41.83	28.62
	3rd Qu.	4.00	4.00	3.00		3rd Qu.	109.24	49.48	22.52
	Histogram					Histogram			
#Modified directories	1st Qu.	1.00	1.00	1.00	#Total authors	1st Qu.	0.00	0.00	1.00
	Median	1.00	1.00	1.00		Median	1.00	2.00	4.00
	Mean	4.10	3.09	2.57		Mean	2.70	3.31	13.40
	3rd Qu.	2.00	2.00	2.00		3rd Qu.	3.00	4.00	13.00
	Histogram					Histogram			
Entropy	1st Qu.	0.00	0.00	0.00	#Reviewers of prior patches	1st Qu.	0.00	1.00	1.50
	Median	0.22	0.10	0.49		Median	1.00	1.00	3.00
	Mean	0.90	0.83	0.85		Mean	0.92	1.29	2.85
	3rd Qu.	1.46	1.31	1.36		3rd Qu.	1.00	2.00	4.00
	Histogram					Histogram			
Description length	1st Qu.	11.00	10.00	13.00	Discussion length of prior patches	1st Qu.	0.00	0.00	1.00
	Median	22.00	19.00	28.00		Median	0.00	0.50	2.00
	Mean	35.61	28.74	36.46		Mean	1.23	1.90	4.15
	3rd Qu.	45.00	36.00	47.00		3rd Qu.	1.00	2.00	5.00
	Histogram					Histogram			
Purpose	BUG-FIX	13360.00	24827.00	39405.00	Feedback delay of prior patches	1st Qu.	0.00	0.35	0.73
	Document	757.00	6830.00	8167.00		Median	0.64	1.53	2.32
	Feature	19852.00	34471.00	48043.00		Mean	31.91	24.17	11.22
						3rd Qu.	3.19	7.87	6.45
	Histogram					Histogram			
Past Involvement of an Author Dimension					Past Involvement of Reviewers Dimension				
#Prior patches of an author	1st Qu.	3.00	13.00	4.00	#Prior patches of reviewers	1st Qu.	0.00	14.00	34.00
	Median	22.00	60.00	24.00		Median	21.00	84.00	171.00
	Mean	83.36	128.50	96.02		Mean	101.80	179.80	331.60
	3rd Qu.	105.00	168.00	114.00		3rd Qu.	139.00	245.00	495.00
	Histogram					Histogram			
Recent patches of an author	1st Qu.	0.33	2.67	1.00	Recent patches of reviewers	1st Qu.	0.00	2.45	13.88
	Median	8.15	23.00	10.17		Median	7.88	33.00	72.05
	Mean	35.27	53.65	40.03		Mean	43.03	73.73	135.41
	3rd Qu.	48.67	74.40	49.75		3rd Qu.	60.50	105.00	200.50
	Histogram					Histogram			
#Directory patches of an author	1st Qu.	0.00	1.00	2.00	#Directory patches of reviewers	1st Qu.	0.00	1.00	9.00
	Median	8.00	13.00	11.00		Median	4.00	15.00	48.00
	Mean	78.34	116.60	68.44		Mean	82.75	104.90	161.80
	3rd Qu.	52.00	76.00	51.00		3rd Qu.	44.00	88.00	164.00
	Histogram					Histogram			
Review Environment Dimension									
Overall work-load	1st Qu.	221.00	511.00	1220.00	Directory work-load	1st Qu.	1.00	1.00	2.00
	Median	394.00	580.00	1524.00		Median	3.00	5.00	8.00
	Mean	369.40	587.20	1430.00		Mean	10.85	10.66	63.79
	3rd Qu.	477.00	674.00	1626.00		3rd Qu.	10.00	12.00	33.00
	Histogram					Histogram			

Table 7.4. Patch data for the study of RQ1.

System	Studied period	Patch data	
		#Patches that did not attract reviewers (TRUE class)	#Total Patches
Android	2012/01 - 2014/12	8,356	33,969
Qt	2012/01 - 2014/12	7,234	66,128
OpenStack	2013/01 - 2014/12	8,360	95,615

patches that do not have reviewers. We then construct logistic regression models, which we describe in detail below.

(MC-1) Correlation & Redundancy Analysis. Before constructing a model, we remove the explanatory variables in Table 7.2 that are highly correlated with one another based on hierarchical clustering analysis. If a cluster of explanatory variables have a Spearman's $|\rho| > 0.7$, we select one variable from the cluster. For example, Figure 7.3 shows the hierarchical clustering of explanatory variables in the Android dataset. There are three clusters of variables that have a Spearman's $|\rho| > 0.7$, i.e., (1) the number of files and the number of directories, (2) the number of prior patches, recent patches, and directory patches of an author, (3) the number of prior defects and the total number of authors, and (4) the number of reviewers and discussion length of prior patches. For the first and second clusters, we select the number of files and the prior patches of an author as the representative variables because they are simpler to calculate than the other variables in their clusters. For the third cluster, we select the number of prior defects and remove the total number of authors since the distribution of the number of prior defects is less skewed than the total number of authors. For the fourth cluster, both explanatory variables (the number of reviewers and discussion length of prior patches) are involvement tendency metrics, which are simple to calculate. We select the number of reviewers of prior patches as the representative variable because the number of reviewers of prior patches shares a more intuitive link with the response (i.e., the likelihood that a patch will attract reviewers) than the discussion length of prior patches.

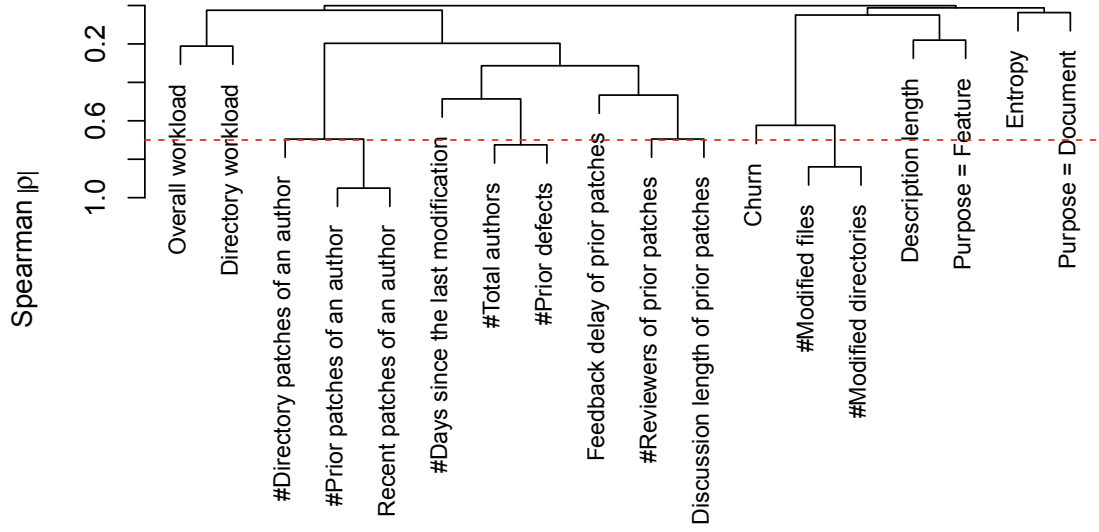


Figure 7.3. Hierarchical clustering of variables according to Spearman’s $|\rho|$ in the Android dataset (RQ1). The dashed line indicates the high correlation threshold (i.e., Spearman’s $|\rho| = 0.7$).

After we remove the highly correlated explanatory variables, we repeat the variables clustering analysis and find that the number of files and churn are also highly correlated, i.e., Spearman’s $|\rho| > 0.7$. Hence, we select the churn and remove the number of files because the distribution of churn data is less skewed than the number of files. For the Qt and OpenStack datasets, we obtain similar results from correlation analysis. Table 7.5 shows the results of our correlation analysis where the variables that were removed during the analysis are marked with a dagger symbol ($\dagger$), while the variables that were removed from all three models are not listed in the table.

For the surviving explanatory variables, we perform a redundancy analysis to detect and remove redundant variables. We find that there are no explanatory variables that have a fit with an R^2 greater than 0.9. Hence, we use all of the surviving explanatory variables to construct our models.

(MC-2) Nonlinear Logistic Regression Model Construction. We allocate the budgeted degrees of freedom to the surviving explanatory variables based on

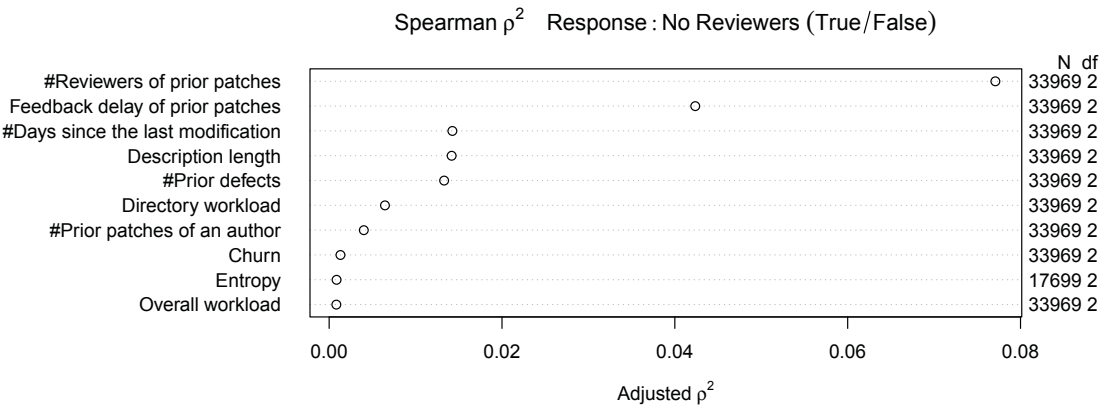


Figure 7.4. Dotplot of the Spearman multiple ρ^2 of each explanatory variable and the response (the likelihood that a patch will not attract reviewers) in the Android dataset. Larger values indicate a higher potential for a nonlinear relationship (RQ1).

their potential for sharing a nonlinear relationship with the response variable. For example, Figure 7.4 shows the potential for nonlinearity in the relationship between explanatory variables and the response variable in the Android dataset. We allocate additional degrees of freedom to the explanatory variables with a large Spearman multiple ρ^2 .

By observing the rough clustering of variables according to the Spearman multiple ρ^2 values, we split the explanatory variables of Figure 7.4 into three groups. We allocate: (1) five degrees of freedom to the number of reviewers of prior patches, (2) three degrees of freedom to feedback delay of prior patches, and (3) one degree of freedom to the remaining variables. We repeat the same process for the Qt and OpenStack datasets.

We then build our logistic regression models to fit our patch data using the surviving explanatory variables with the allocated degrees of freedom. Table 7.5 shows that the number of degrees of freedom that we spent to fit our models did not exceed the budgeted degrees of freedom.

Table 7.5. Statistics of the logistic regression models for identifying patches that do not attract reviewers (RQ1). The explanatory variables that contribute the most significant explanatory power to a model (i.e., accounting for a large proportion of Wald χ^2) are shown in boldface.

		Android		Qt		OpenStack	
AUC (Optimism)		0.72 (0.002)		0.70 (0.001)		0.74 (0.001)	
Wald χ^2		2,218***		2,910***		3,804***	
		Overall	Nonlinear	Overall	Nonlinear	Overall	Nonlinear
<i>Patch Properties Dimension</i>							
Churn	D.F. χ^2	1 0% [°]	—	1 0% [°]	—	1 0%***	—
Entropy	D.F. χ^2	1 0% [°]	—	1 1%***	—	1 1%***	—
Description length	D.F. χ^2	1 0%*	—	2 15%***	1 13%***	2 5%***	1 5%***
Purpose	D.F. χ^2	2 1%***	—	2 1%***	—	2 1%***	—
<i>History Dimension</i>							
#Days since the last modification	D.F. χ^2	1 16%***	—	1 30%***	—	1 15%***	—
#Total authors	D.F. χ^2	†		1 0% [°]	—	†	
#Prior defects	D.F. χ^2	1 0% [°]	—	1 0% [°]	—	1 0%**	—
#Reviewers of prior patches	D.F. χ^2	2 81%***	1 26%***	3 69%***	2 42%***	4 72%***	3 32%***
Feedback delay of prior patches	D.F. χ^2	2 1%***	1 1%***	1 0% [°]	—	2 1%***	1 0%***
<i>Past Involvement of an Author Dimension</i>							
#Prior patches of an author	D.F. χ^2	1 0%**	—	1 0%***	—	1 0%**	—
<i>Review Environment Dimension</i>							
Overall workload	D.F. χ^2	1 0%*	—	1 0% [°]	—	1 0%*	—
Directory workload	D.F. χ^2	1 1%***	—	1 0% [°]	—	1 0%***	—

†: 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:

[°] $p \geq 0.05$; * $p < 0.05$; ** $p < 1\%$; *** $p < 0.001$

(RQ1-b) Model Analysis

In this section, we present and discuss the results of our model analysis approach that is outlined in Figure 7.2 and present our empirical observations.

(MA-1) Assessment of Explanatory Ability & Model Reliability. Table 7.5 shows that our models achieve an AUC of 0.7 to 0.74. Moreover, Table 7.5 also shows that the optimism of AUC is very small ($|\text{Optimism}| = 0.002$ for the Android system and $|\text{Optimism}| = 0.001$ for the Qt and OpenStack systems). These results indicate that our models are stable and can provide a meaningful and robust amount of explanatory power.

(MA-2) Power of Explanatory Variables Estimation. Table 7.5 shows the explanatory power (Wald χ^2) of our explanatory variables that contribute to the fit of our models. In the table, the *Overall* column shows the proportion of the Wald χ^2 of the entire model fit that is attributed to that explanatory variable, and the *Nonlinear* column shows the proportion of the Wald χ^2 of the entire model fit that is attributed to the nonlinear component of that explanatory variable. The larger the proportion of the Wald χ^2 is, the larger the explanatory power that a particular explanatory contributes explanatory power to the fit of the model.

Table 7.5 shows that the number of reviewers of prior patches and the number of days since the last modification account for the largest proportion of Wald χ^2 in our three models. Hence, the number of reviewers of prior patches and the number of days since the last modification contribute the most significant explanatory power to the fit of our models. The description length also contributes a relatively large, significant amount of explanatory power to the Qt model.

On the other hand, Table 7.5 shows that churn did not contribute a significant amount of explanatory power to our three models. Moreover, we observe that entropy, the number of prior defects, feedback delay of prior patches, and the explanatory variables in the past involvement of an author and the review environment dimensions contribute a small explanatory power, although they have a statistically significant impact on our models.

Table 7.5 also shows that four of the seven explanatory variables to which we allocated nonlinear degrees of freedom provide significant boosts to the explanatory power of the model. This result indicates that the nonlinear style of modelling is improving the fit of our models, providing a more in-depth picture of the relationship between explanatory variables and the response.

(MA-3) Examination of Variables in Relation to Response. To study the relationship between the explanatory variables and the response, we plot the odds produced by our models against an explanatory variable while holding the other explanatory variables at their median values. For example, Figure 7.5 shows the nonlinear relationship between the explanatory variables and the response with the 95% confidence interval (gray area) based on models fit to 1,000 bootstrap samples.

To estimate the partial effect that the explanatory variables have on the likelihood that a patch will not attract reviewers, we analyze the relative change in the odds corresponding to a shift in the value of each explanatory variable. Table 7.6 shows the estimated partial effect of each explanatory variable in our models. The *Odds Ratio* column shows the partial effect based on the shifted value shown in the *Observed Value* column. For continuous variables, the observed value is an inter-quartile range of those explanatory variables. For categorical variables, the observed value is a comparison between the observed category and the mode (i.e., the most frequently occurring category).

Below, we present and discuss our empirical observations from the examination of these explanatory variables in relation to the response.

Observation 25 – The number of reviewers of prior patches shares an increasing relationship with the likelihood that a patch will attract reviewer. Table 7.5 shows that there is a nonlinear relationship between the number of reviewers of prior patches and the likelihood that a patch will not attract reviewers. For example, Figure 7.5(a) shows that the likelihood that an Android patch will not attract a reviewer decreases rapidly as the number of reviewers of prior patches

increases from 0 to 1. We observe similar trends in the Qt and OpenStack models. Furthermore, Table 7.6 shows that the likelihood can decrease by 89%, 14%, and 40% when the number of reviewers of prior patches increases from 0 to 1, 1 to 2, and 2 to 4 in the Android, Qt, OpenStack models, respectively. Broadly speaking, our models show that patches are more likely to attract reviewers if past changes to the modified files have a tendency to be reviewed by at least two reviewers. These results indicate that patches that modify files whose prior patches have had few reviewers tend to not attract reviewers.

Observation 26 – The number of days since the last modification shares an increasing relationship with the likelihood that a patch will attract reviewers.

Table 7.6 also shows that the number of days since the last modification consistently shares a strong inverse relationship with the likelihood that a patch will not attract reviewers in the three models for the studied systems. The likelihood decreases by 50%, 45%, and 19% when the number of days since the last modification is changed from 1 to 109, 1 to 49, and 1 to 23 in the Android, Qt, and OpenStack models, respectively. This result indicates that a patch containing files that have been recently modified is not likely to attract reviewers.

Observation 27 – Description length shares an increasing relationship with the likelihood that a patch will attract reviewers. Figure 7.5(b) shows that there is a decreasing trend in the likelihood that a Qt patch will not attract reviewers as the description length increases. On the other hand, there is an increasing trend in the likelihood as the description length increases beyond 50. However, the broadening of the confidence interval (gray area) indicates that there is less data to support this area of the curve. Table 7.6 also shows that the description length shares a relatively strong relationship with the likelihood in the Qt and OpenStack models. When the description length is greater than 10 words, the likelihood decreases by 3%, 57%, and 41% in the Android, Qt, and OpenStack models, respectively. This result indicates that a patch with a short description is unlikely to attract reviewers.

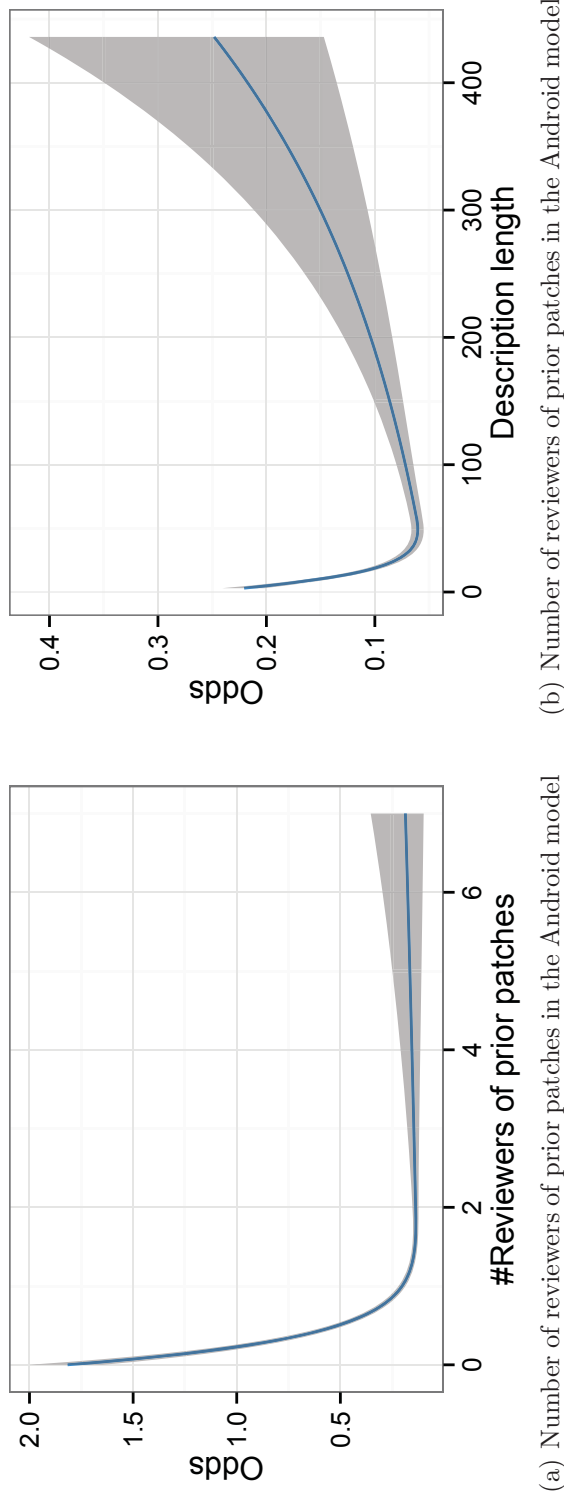


Figure 7.5. The nonlinear relationship between the likelihood that a patch will not attract reviewers (y-axis) and the explanatory variables (x-axis). The larger the odds value is, the higher the likelihood that the patch will not attract reviewers. The gray area shows the 95% confidence interval estimated by using a bootstrap-derived approach.

Table 7.6. Partial effect that our explanatory variables have on the likelihood that a patch will not attract reviewers (RQ1). The larger the magnitude of the odds ratio is, the larger the partial effect that an explanatory variable has on the likelihood that a patch will not attract reviewers.

	Android		Qt		OpenStack	
	Shifted Value	Odds Ratio	Shifted Value	Odds Ratio	Shifted Value	Odds Ratio
<i>Patch Properties Dimension</i>						
Churn	5→96	0%	4→63	0%	5→90	0%
Description length	11→45	-3%↓	10→36	-57%↓	13→47	-41%↓
Entropy	1→1	0%	1→1	-11%↓	1→1	-11%↓
Purpose	Feature→BUG-FIX	-10%↓	Feature→BUG-FIX	0%	Feature→BUG-FIX	-23%↓
	Feature→Document	-37%↓	Feature→Document	-30%↓	Feature→Document	-25%↓
<i>History Dimension</i>						
#Days since the last modification	1→109	-50%↓	1→49	-45%↓	1→23	-19%↓
#Total authors	—	—	0→4	2%↑	—	—
#Prior defects	0→4	-1%↓	0→4	0%	0→10	2%↑
#Reviewers of prior patches	0→1	-89%↓	1→2	-14%↓	2→4	-40%↓
Feedback delay of prior patches	0→3	9%↑	0→8	0%	1→6	16%↑
<i>Past Involvement of an Author Dimension</i>						
#Prior patches of an author	3→105	-4%↓	13→168	-6%↓	4→114	4%↑
<i>Review Environment Dimension</i>						
Overall workload	221→477	9%↑	511→674	-5%↓	1220→1626	-6%↓
Directory workload	1→10	3%↑	1→12	1%↑	2→33	2%↑

Table 7.7. Patch data for the study of RQ2.

System	Studied period	Patch data	
		#Patches that are not discussed (TRUE class)	#Patches with reviewers
Android	2012/01 - 2014/12	12,262	25,613
Qt	2012/01 - 2014/12	29,771	58,894
OpenStack	2013/01 - 2014/12	21,855	87,255

Summary: The number of reviewers of prior patches and the number of days since the last modification share a strong increasing relationship with the likelihood that a patch will have at least one reviewers. Furthermore, a short patch description can also lower the likelihood of attracting reviewers (Observations 25-27).

(RQ2) Which patch characteristics share a relationship with the likelihood of a patch not being discussed?

A review that simply assigns a review score without any suggestion for improvement nor discussion provides little return on code review investment. Hence, to address our RQ2, we perform our analysis on patches that have reviewers but were not discussed. We filter the patches that did not attract any reviewers, since such patches cannot receive any feedback. We then identify the patches that are not discussed by counting the number of messages that are posted in the review discussion thread of each patch. We classify the patches that had no messages as *patches that are not discussed*. Patches that had at least one message are defined as patches that are discussed.

Table 7.7 provides an overview of the studied patch data after we remove patches that did not attract reviewers. Below, we present and discuss the results of our model construction and analysis.

(RQ2-a) Model Construction

Similar to RQ1, the response variable is set to TRUE if a patch is not discussed and FALSE otherwise. We then use all of our patch and MCR process metrics that are described in Table 7.2 to construct logistic regression models for each studied system.

(MC-1) Correlation & Redundancy Analysis. Since we filtered out patches that did not attract any reviewers and we added the past involvement of reviewers metrics into the models, we have to perform correlation & redundancy analysis for the studied patch data again. Figure 7.6 shows that there are four clusters of variables that have a Spearman's $|\rho| > 0.7$: (1) the variables in the past involvement of an author dimension, (2) the number of files and directories, (3) the number of prior patches of reviewers and the number of recent patches of reviewers, (4) the number of prior defects and the number of total authors, and (5) the number of reviewers and the discussion length of prior patches. For the first three clusters, we select the number of prior patches of an author, the number of files, and the number of prior patches of reviewers as the representative variables because they are simpler to calculate than the other variables. For the forth cluster, we select the number of prior defects as the representative variable since the distribution of the number of prior defects is less skewed than the total number of authors. For the fifth cluster, we select the discussion length of prior patches because it shares a more intuitive link with the response (i.e., the likelihood that a patch will not be discussed) than the number of reviewers of prior patches. For the Qt and OpenStack datasets, Table 7.8 shows the results of our correlation analysis.

(MC-2) Nonlinear Logistic Regression Model Construction. Figure 7.7 shows the estimated potential for nonlinear relationships between each explanatory variable and the likelihood that a patch will not be discussed in the Android dataset. We split the explanatory variables for the Android dataset into three groups: (1)

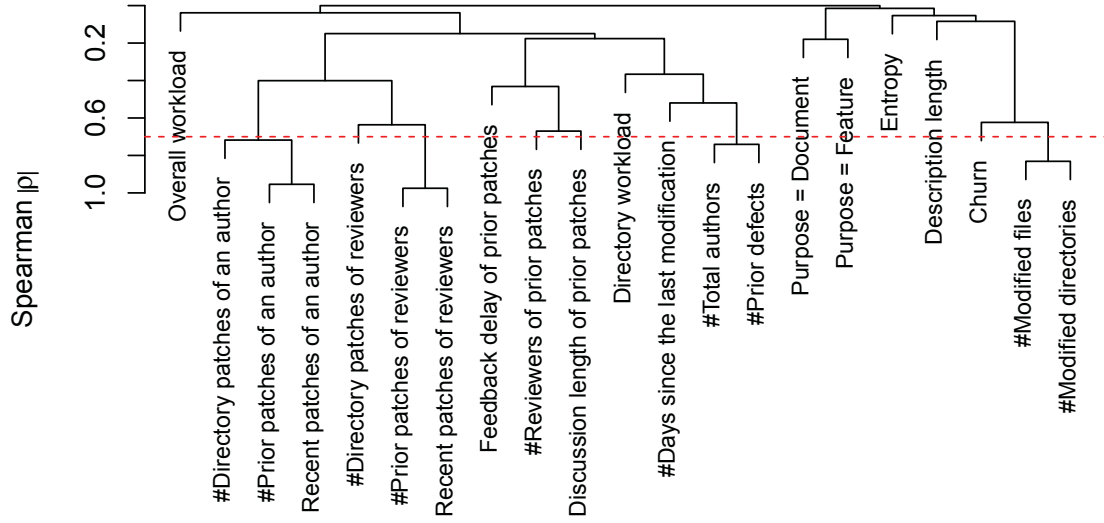


Figure 7.6. Hierarchical clustering of variables according to Spearman’s $|\rho|$ in the Android dataset (RQ2). The dashed line indicates the high correlation threshold (i.e., Spearman’s $|\rho| = 0.7$).

the description length (2) churn, the number of prior defects, discussion length of prior patches, the number of prior patches of an author and the number of prior patches of reviewers, and (3) the remaining explanatory variables. We then allocate five degrees of freedom to the first group, three degrees of freedom to the second group, and one degree of freedom to the third group. We repeat the same process for the Qt and OpenStack datasets.

(RQ2-b) Model Analysis

In this section, we present the results of our model analysis approach that is outlined in Figure 7.2 and present our empirical observations.

(MA-1) Assessment of Explanatory Ability & Model Reliability. Table 7.8 shows that our models achieve an AUC of 0.70 to 0.78 and the optimism of AUC is very small for all of our studied datasets. These results indicate that our models can provide a meaningful and robust amount of explanatory power.

(MA-2) Power of Explanatory Variables Estimation. Table 7.8 shows the pro-

Table 7.8. Statistics of the logistic regression models for identifying patches that are not discussed (RQ2). The explanatory variables that contribute the most significant explanatory power to a model (i.e., accounting for a large proportion of Wald χ^2) are shown in boldface.

		Android		Qt		OpenStack	
AUC (Optimism)		0.70 (0.002)		0.72 (0.001)		0.78 (0.001)	
Wald χ^2		1,312***		3,681***		5,882***	
		Overall	Nonlinear	Overall	Nonlinear	Overall	Nonlinear
<i>Patch Properties Dimension</i>							
Churn	D.F. χ^2	2 12%***	1 12%***	2 19%***	1 19%***	2 8%***	1 8%***
Entropy	D.F. χ^2	1 0%°	—	2 2%***	1 2%***	1 0%***	—
Description length	D.F. χ^2	4 18%***	3 7%***	2 5%***	1 1%***	4 17%***	3 4%***
Purpose	D.F. χ^2	2 0%°	—	2 1%***	—	2 0%*	—
<i>History Dimension</i>							
#Days since the last modification	D.F. χ^2	1 2%***	—	1 1%***	—	1 3%***	—
#Prior defects	D.F. χ^2	2 8%***	1 2%***	1 1%***	—	2 3%***	1 2%***
Discussion length of prior patches	D.F. χ^2	2 14%***	1 12%***	2 23%***	1 21%***	2 27%***	1 17%***
Feedback delay of prior patches	D.F. χ^2	1 1%**	—	1 0%°	—	1 0%°	—
<i>Past Involvement of an Author Dimension</i>							
#Prior patches of an author	D.F. χ^2	1 30%***	—	2 25%***	1 4%***	1 7%***	—
<i>Past Involvement of Reviewers Dimension</i>							
#Prior patches of reviewers	D.F. χ^2	1 10%***	—	1 14%***	—	2 5%***	1 0%°
#Directory patches of reviewers	D.F. χ^2	†		1 1%***	—	2 2%***	1 1%***
<i>Review Environment Dimension</i>							
Overall workload	D.F. χ^2	1 0%*	—	1 0%***	—	1 0%***	—
Directory workload	D.F. χ^2	1 0%°	—	1 1%***	—	1 1%***	—

†: 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:

° $p \geq 0.05$; * $p < 0.05$; ** $p < 1\%$; *** $p < 0.001$

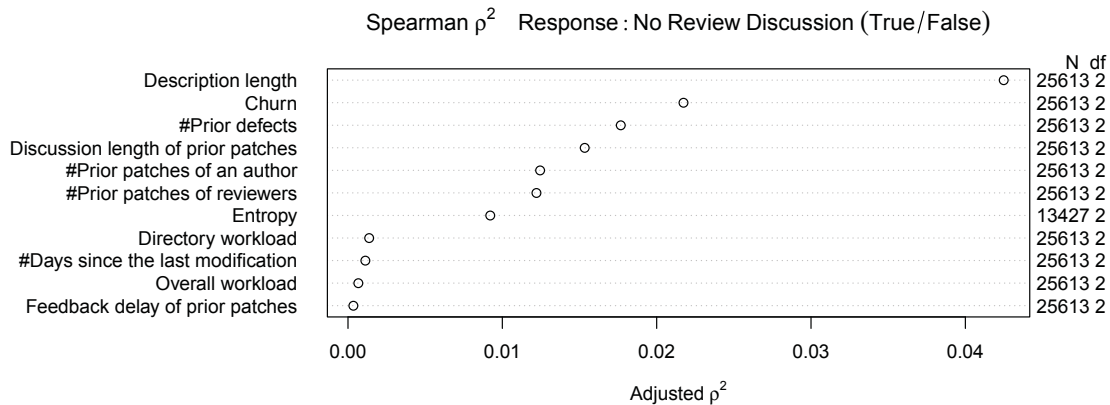


Figure 7.7. Dotplot of the Spearman multiple ρ^2 of each explanatory variable and the response (the likelihood that a patch will not be discussed) in the Android dataset. Larger values indicate a higher potential for a nonlinear relationship (RQ2).

portion of Wald χ^2 of the explanatory variables that contribute to the fit of our models. We find that the discussion length of prior patches has a large proportion of Wald χ^2 in our three models. The churn, the description length, the number of prior patches of an author and reviewers also account for a large proportion of the explanatory power in two of the three models.

Table 7.8 shows that entropy, feedback delay of prior patches, the number of prior patches within the same directory that the reviewer has reviewed, and the explanatory variables in the review environment dimension did not contribute a significant amount of explanatory, although they survive our correlation and redundancy analysis. These results indicate that these explanatory variables in our models share a weaker relationship with the likelihood of a patch being discussed than other metrics.

Table 7.8 also shows that five of the fifteen explanatory variables to which we allocated nonlinear degrees of freedom provide significant boosts to the explanatory power of the model. This result indicates that the nonlinear style of modelling can improve the fit of our models. However, we find that the nonlinear degrees of freedom that we allocate to the number of prior patches of review-

ers does not provide a significant amount of explanatory power to the OpenStack model, suggesting that not all relationships with potential for nonlinearity benefit from nonlinear fits.

(MA-3) Examination of Variables in Relation to Response. Figure 7.8 shows the nonlinear relationships of the high impact explanatory variables and the response. Table 7.9 shows the estimated partial effect of that each explanatory variable has on the likelihood that a patch will not be discussed. Below, we present and discuss our empirical observations from the examination of these explanatory variables in relation to the response.

Observation 28 – Churn shares an increasing relationship with the likelihood that a patch will be discussed. We observe that churn shares a strong relationship with the likelihood that a patch will not be discussed in our three models. Table 7.9 shows that the likelihood decreases by 30%, 46%, and 40% in the Android, Qt, OpenStack models, respectively. Figure 7.8(a) shows that there is a decreasing trend of the odds in the Qt model when the churn increases from 1 to 150 LOC. Then, the odds stabilizes when the churn increases to more than 150 LOC. We also observe similar trend of the odds produced by the Android model. The odds decrease when the churn increases from 1 to 300 LOC, then the odds stabilize when the churn increases to more than 300 LOC. This result indicates that the more lines that were changed in the patch, the more likely the patch will be discussed.

Observation 29 – The description length shares an increasing relationship with the likelihood that a patch will be discussed. Table 7.9 shows that the description length consistently shares an inverse relationship with the likelihood that a patch will not be discussed in our three studied systems. The likelihood decreases by 43%, 24%, and 43% when the description length is greater than 11 words. Our results suggest that the longer the description that an author provides, the higher the likelihood of the patch being discussed.

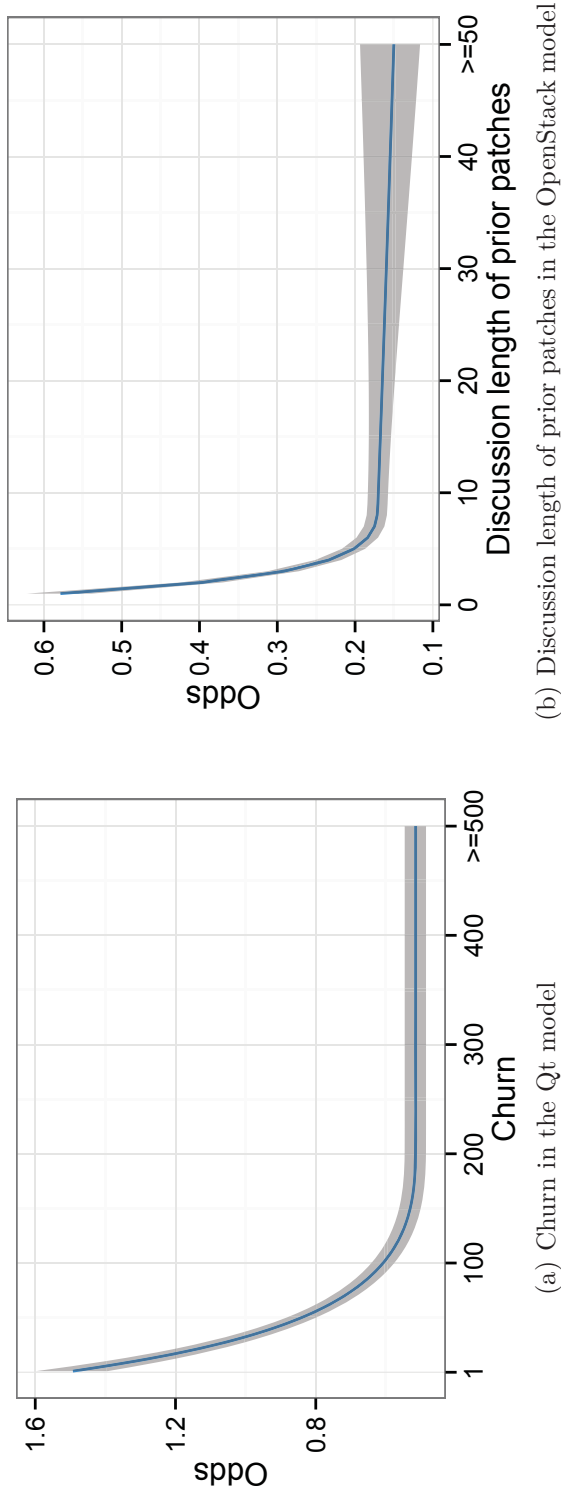


Figure 7.8. The nonlinear relationship between the likelihood that a patch will not be discussed (y-axis) and the explanatory variables (x-axis). The larger the odds value is, the higher the likelihood that the patch will not be discussed. The gray area shows the 95% confidence interval estimated by using a bootstrap-derived approach.

Table 7.9. Partial effect that our explanatory variables have on the likelihood that a patch will not be discussed (RQ2). The larger the magnitude of the odds ratio is, the larger the partial effect that an explanatory variable has on the likelihood that a patch will not be discussed.

	Android		Qt		OpenStack	
	Shifted Value	Odds Ratio	Shifted Value	Odds Ratio	Shifted Value	Odds Ratio
<i>Patch Properties Dimension</i>						
Churn	5→96	-30%↓	4→60	-46%↓	5→91	-40%↓
Description length	11→46	-43%↓	11→38	-24%↓	15→48	-43%↓
Entropy	1→1	-3%↓	1→1	6%↑	1→1	7%↑
Purpose	Feature→BUG-FIX	-1%↓	Feature→BUG-FIX	-19%↓	Feature→BUG-FIX	-7%↓
	Feature→Document	7%↑	Feature→Document	-11%↓	Feature→Document	-9%↓
<i>History Dimension</i>						
#Days since the last modification	2→92	-16%↓	1→52	-7%↓	1→22	-8%↓
#Prior defects	0→5	-27%↓	0→4	-3%↓	1→11	-34%↓
Discussion length of prior patches	0→2	-45%↓	0→2	-58%↓	1→6	-67%↓
Feedback delay of prior patches	0→4	0%	0→8	0%	1→6	0%
<i>Past Involvement of an Author Dimension</i>						
#Prior patches of an author	3→111	41%↑	12→167	104%↑	4→114	21%↑
<i>Past Involvement of Reviewers Dimension</i>						
#Prior patches of reviewers	11→207	-26%↓	31→278	-32%↓	56→532	-37%↓
#Directory patches of reviewers	—	—	3→107	4%↑	16→183	-18%↓
<i>Review Environment Dimension</i>						
Overall workload	221→476	-7%↓	507→673	-6%↓	1225→1626	10%↑
Directory workload	1→10	0%	1→12	6%↑	2→31	-3%↓

Observation 30 – The discussion length of prior patches shares an increasing relationship with the likelihood that a patch will be discussed. Figure 7.8(b) shows that the odds, produced by the OpenStack model, sharply decrease when the discussion length of prior patches increases from 0 to 10 messages. We observe similar trends of the odds in the Android and Qt models. Table 7.9 shows that the likelihood decreases by 45%, 58%, and 67% when the discussion length increases beyond 2 messages in the Android, Qt, OpenStack models, respectively. These results indicate that patches that modify files whose prior patches typically had short review discussions is not likely to be discussed.

Observation 31 – The number of prior patches of an author shares an inverse relationship with the likelihood that a patch will be discussed, while the number of prior patches of reviewers shares an increasing relationship with the likelihood. Table 7.9 shows that the likelihood increases by 41%, 104%, and 21% when the number of prior patches of an author is greater than 3, 12, and 4 in the Android, Qt, OpenStack models, respectively. Furthermore, Table 7.9 shows that the number of prior patches of reviewers shares an inverse relationship with the likelihood that a patch will not be discussed. The likelihood can decrease by 26%, 32%, and 37% in the Android, Qt, and OpenStack models, respectively. Our results indicate that a patch written by an experienced developer or examined by an inexperienced reviewer is not likely to be discussed.

Summary: Churn, the description length, and the discussion length of prior patches share an increasing relationship with the likelihood that a patch will be discussed. Moreover, the past involvement of an author shares an inverse relationship, while the past involvement of reviewers shares an increasing relationship with the likelihood that a patch will be discussed (Observations 28-31).

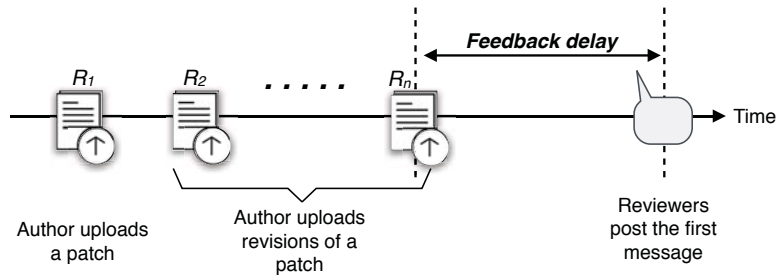


Figure 7.9. An example of a calculation for feedback delay.

Table 7.10. Descriptive statistics of feedback delay (hours).

System	Min	1st Qu.	Median	Mean	3rd Qu.	Max
Android	0.00	0.19	1.20	106.50	11.78	16,470.00
Qt	0.00	0.28	1.49	70.75	15.70	20,260.00
OpenStack	0.00	0.27	2.17	31.82	15.17	7,031.00

(RQ3) Which patch characteristics share a relationship with the likelihood of a patch receiving slow initial feedback?

To address our RQ3, we identify patches that receive slow initial feedback by measuring feedback delay, i.e., a time period between the submission time of the latest patch revision before the first review message is posted and the time that the first review message is posted. Figure 7.9 provides an example of a calculation for feedback delay. We did not use the time that the original patch is submitted because there are likely cases that reviewers are still waiting for the author to complete preliminary revisions, which would incorrectly inflate the feedback delay.

To follow our model construction approach, we classify the patches into two groups, i.e., patches that receive prompt initial feedback and patches that receive slow initial feedback. From the descriptive statistics of feedback delay (see Table 7.10), we classify the patches that have a feedback delay of more than 12 hours as *patches that receive slow initial feedback*. Patches that receive initial feedback within 12 hours are defined as patches that receive prompt initial feedback.

Table 7.11. Patch data for the study of RQ3.

System	Studied period	Patch data	
		#Patches that received slow initial feedback (TRUE class)	#Patches with reviewers & submitted in workdays
Android	2012/01 - 2014/12	5,759	23,287
Qt	2012/01 - 2014/12	15,900	54,783
OpenStack	2013/01 - 2014/12	22,302	79,431

Similar to RQ2, we want to investigate the characteristics of patches that receive slow initial feedback although these patches eventually have reviewers providing feedback. Hence, we filter out patches that did not attract any reviewers, since such patches do not receive any feedback. Moreover, we filter out patches that are submitted on weekends, since our prior study finds that code review activity is often less active on weekend than on weekdays [125]. Hence, such patches have a lower chance to receive initial feedback within 24 hours. Since our studied systems have globally-distributed development teams, the timezone difference could make it difficult to detect weekends consistently. To identify weekends, we observe the number of code review activity (i.e., a patch submission, posting a review message, or voting a review score). For example, Figure 7.10 shows the hourly code review activity of the Qt system, where the periods with small amounts of code review activity are indicated using dark shading. We mark the areas with small amounts of code review activity as weekends for the Qt system, and we remove the patches that are submitted in these periods from our analysis. We repeat the same process for the Android and OpenStack systems.

Table 7.11 provides an overview of the studied patch data after we remove patches that did not attract reviewers and that are submitted during weekends. Below, we present and discuss the results of our model construction and analysis.

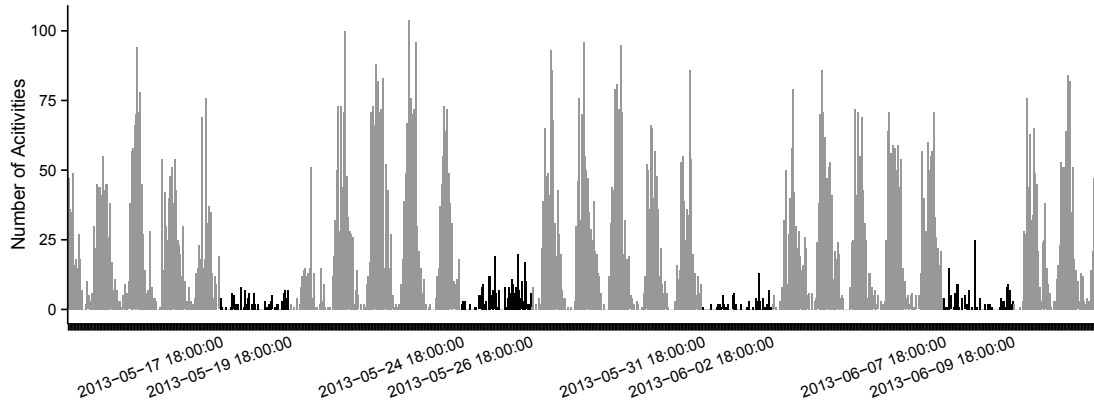


Figure 7.10. The hourly code review activity of the Qt system. The dark areas indicate the periods that are likely to be weekends.

(RQ3-b) Model Construction

Again, we use our patch and MCR process metrics that are described in Table 7.2 as explanatory variables. The response is set to TRUE if a patch receives slow initial feedback, and FALSE otherwise. We then construct a logistic regression model, which we describe in detail below.

(MC-1) Correlation & Redundancy Analysis. We check for highly correlated and redundant variables again, since we filter out patches that did not attract any reviewers, and were submitted on weekends. We find that the hierarchical clustering analysis shows the same results as the analysis in RQ2. Hence, we use the same set of surviving variables. Table 7.12 shows the results of our correlation analysis.

(MC-2) Nonlinear Logistic Regression Model Construction. Again, we allocate additional degrees of freedom to the surviving explanatory variables based on their potential for sharing a nonlinear relationship with the response. By observing the Spearman multiple ρ^2 values in Figure 7.11, we split the explanatory variables into two groups: (1) the feedback delay of prior patches and the number of prior patches of an author, and (2) the remaining explanatory variables. We then allocate five and one degree of freedom to each group respectively. We repeat the

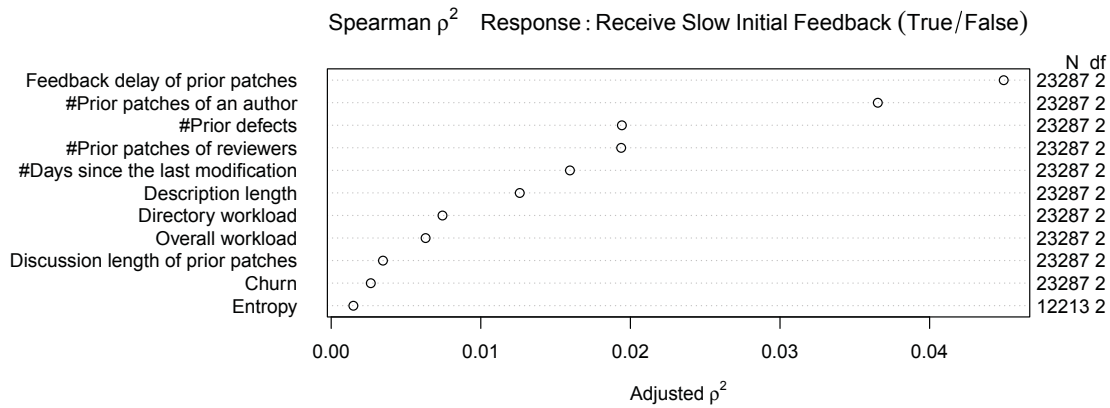


Figure 7.11. Dotplot of the Spearman multiple ρ^2 of each explanatory variable and the response (the likelihood that a patch will receive slow initial feedback) in the Android model. Larger values indicate a higher potential for a nonlinear relationship (RQ3).

same process for Qt and OpenStack datasets. Table 7.12 shows the number of degrees of freedom that we spent to build our logistic regression models.

(RQ3-c) Model Analysis

In this section, we describe the results of our model analysis approach that is outlined in Figure 7.2 and present our empirical observations.

(MA-1) Assessment of Explanatory Ability & Model Reliability. Table 7.12 shows that our models achieve an AUC of 0.61 to 0.66 and the optimism of 0.001 (OpenStack)-0.004 (Android). These results indicate that our models can determine the likelihood that a patch will receive slow initial feedback and provide a meaningful and robust amount of explanatory power.

(MA-2) Power of Explanatory Variables Estimation. We find that the feedback delay of prior patches contribute a significant amount of explanatory power to the fit of our models. Table 7.12 shows that feedback delay of prior patches accounts for a large proportion of the explanatory power in our three models. Furthermore, we observe that churn also accounts for a large proportion of the explanatory power of the OpenStack model, while the number of prior patches of

Table 7.12. Statistics of the logistic regression models for identifying patches receiving slow initial feedback (RQ3). The explanatory variables that contribute the most significant explanatory power to a model (i.e., accounting for a large proportion of Wald χ^2) are shown in boldface.

		Android		Qt		OpenStack	
AUC (Optimism)		0.66 (0.004)		0.61 (0.002)		0.61 (0.001)	
Wald χ^2		722***		940***		1,380***	
		Overall	Nonlinear	Overall	Nonlinear	Overall	Nonlinear
<i>Patch Properties Dimension</i>							
Churn	D.F. χ^2	1 0% ^o	—	2 6%***	1 6%***	2 15%***	1 15%***
Entropy	D.F. χ^2	1 2%***	—	2 7%***	1 4%***	1 1%**	—
Description length	D.F. χ^2	1 8%***	—	1 0% ^o	—	1 1%***	—
Purpose	D.F. χ^2	2 8%***	—	2 4%***	—	2 12%***	—
<i>History Dimension</i>							
#Days since the last modification	D.F. χ^2	2 7%***	1 1%*	1 4%***	—	1 13%***	—
#Prior defects	D.F. χ^2	1 1% ^o	—	1 0% ^o	—	1 1%***	—
Discussion length of prior patches	D.F. χ^2	1 1%**	—	1 1%*	—	1 0% ^o	—
Feedback delay of prior patches	D.F. χ^2	2 33%***	1 27%***	2 51%***	1 51%***	2 61%***	1 54%***
<i>Past Involvement of an Author Dimension</i>							
#Prior patches of an author	D.F. χ^2	4 18%***	3 7%***	1 5%***	—	1 0%*	—
<i>Past Involvement of Reviewers Dimension</i>							
#Prior patches of reviewers	D.F. χ^2	1 0% ^o	—	1 6%***	—	1 0% ^o	—
#Directory patches of reviewers	D.F. χ^2	†		1 0% ^o	—	1 3%***	—
<i>Review Environment Dimension</i>							
Overall workload	D.F. χ^2	1 2%***	—	1 5%***	—	1 0% ^o	—
Directory workload	D.F. χ^2	1 0% ^o	—	1 5%***	—	1 0% ^o	—

†: 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:

$\circ p \geq 0.05$; * $p < 0.05$; ** $p < 1\%$; *** $p < 0.001$

an author accounts for a large proportion of the explanatory power of the Android model.

Table 7.12 shows that the explanatory power of the variables in the past involvement of reviewers dimension account for a relatively small proportion of the explanatory power. This result suggests that when considering the initial feedback delay, the experience of reviewers is not as important as the other studied dimensions.

(MA-3) Examination of Variables in Relation to Response. Figure 7.12 shows the shape of the nonlinear relationship between the likelihood that a patch will receive slow initial feedback and two of the most impactful explanatory variables. Table 7.13 shows the estimated partial effect that each explanatory variable has on the likelihood of receiving slow initial feedback. Below, we present our observations from the examination of these explanatory variables in relation to the response variable.

Observation 32 – The feedback delay of prior patches has an increasing relationship with the likelihood that a patch will receive slow initial feedback. Figure 7.12(a) shows that the shape of the relationship between the likelihood and the feedback delay of prior patches in the Qt model. The plot shows a steeply increasing trend in the likelihood where the feedback delay of prior patches reaches to 25 hours. We observe the similar trends in the Android and OpenStack models. Table 7.13 also shows that feedback delay of prior patches shares a strong relationship with the likelihood. When the feedback delay of prior patches is greater than one hour, the likelihood increases by 35%, 69%, and 76% in the Android, Qt, and OpenStack models, respectively. These results indicate that patch that modify files whose prior patches had received slow initial feedback tend to also receive slow initial feedback.

Observation 33 – The purpose of introducing a new feature shares an increasing relationship with the likelihood that a patch will receive slow initial feedback. Table 7.13 shows that a patch will have a lower likelihood of receiving

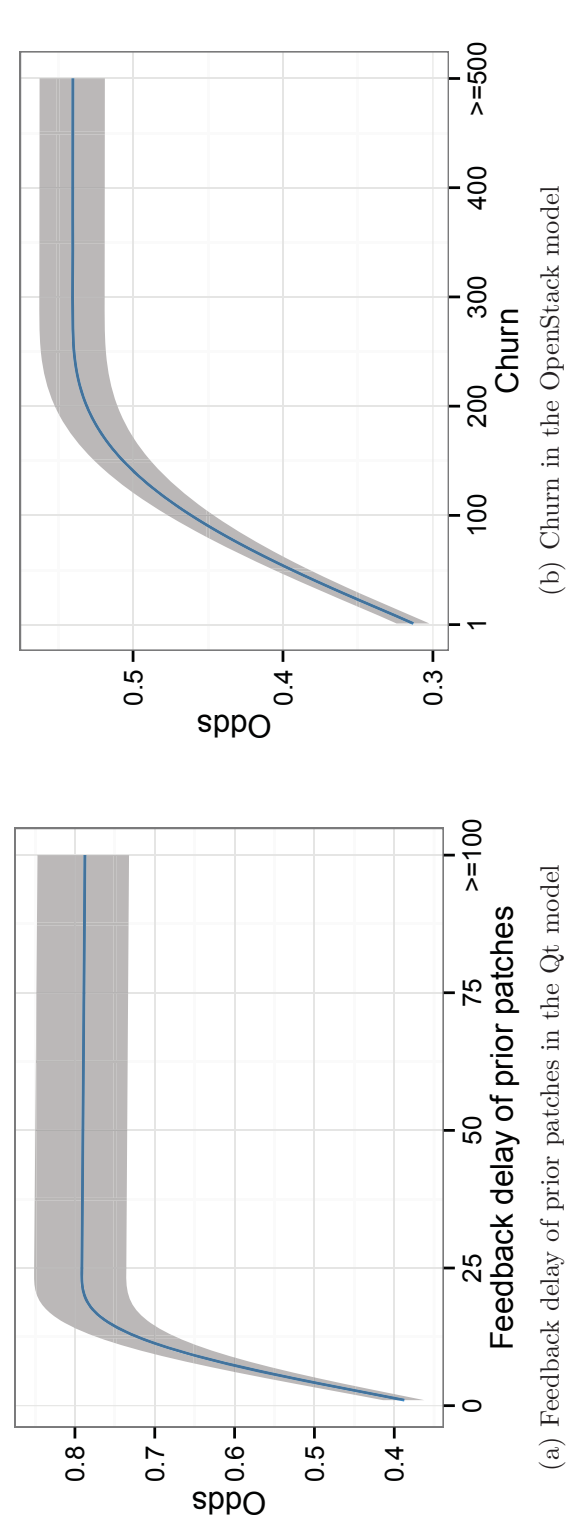


Figure 7.12. The nonlinear relationship between the likelihood that a patch will receive slow initial feedback (y-axis) and the explanatory variables (x-axis). The larger the odds value is, the higher the likelihood that the patch will receive slow initial feedback. The gray area shows the 95% confidence interval estimated by using a bootstrap-derived approach.

Table 7.13. Partial effect that our explanatory variables have on the likelihood that a patch will receive slow initial feedback (RQ3). The larger the magnitude of the odds ratio is, the larger the partial effect that an explanatory variable has on the likelihood that a patch will receive slow initial feedback.

	Android		Qt		OpenStack	
	Shifted Value	Odds Ratio	Shifted Value	Odds Ratio	Shifted Value	Odds Ratio
<i>Patch Properties Dimension</i>						
Churn	5→96	0%	4→59	19%↑	5→90	33%↑
Entropy	1→1	-9%↓	1→1	-12%↓	1→1	4%↑
Description length	11→46	12%↑	11→37	2%↑	15→48	4%↑
Purpose	Feature→BUG-FIX	-29%↓	Feature→BUG-FIX	-16%↓	Feature→BUG-FIX	-25%↓
	Feature→Document	-40%↓	Feature→Document	-12%↓	Feature→Document	-22%↓
<i>History Dimension</i>						
#Days since the last modification	2→93	12%↑	1→51	8%↑	1→22	7%↑
#Prior defects	0→5	8%↑	0→4	0%	1→11	2%↑
Discussion length of prior patches	0→2	-3%↓	0→2	-1%↓	1→6	0%
Feedback delay of prior patches	0→4	35%↑	0→8	69%↑	1→6	76%↑
<i>Past Involvement of an Author Dimension</i>						
#Prior patches of an author	3→109	-42%↓	12→170	9%↑	4→111	-2%↓
<i>Past Involvement of Reviewers Dimension</i>						
#Prior patches of reviewers	11→208	-2%↓	32→275	-11%↓	56→527	1%↑
#Directory patches of reviewers			3→106	0%	15→183	-4%↓
<i>Review Environment Dimension</i>						
Overall workload	221→476	-12%↓	507→674	-11%↓	1229→1626	0%
Directory workload	0→10	-1%↓	1→12	-6%↓	2→31	0%

slow initial feedback when its purpose is changed from feature introduction to other purposes. The likelihood decreases by 16% to 29% when the purpose of the patch is changed from feature introduction to bug fixing, and 12% to 40% when the purpose is changed to documentation. This result indicates that a patch that introduces new functionality is more likely to receive slow initial feedback than patches for other purposes.

In addition to the observations that we have made earlier, we also observe several interesting relationships between characteristics of patches and the likelihood that a patch will receive slow initial feedback. Yet, these relationships were not consistent across our studied systems. We discuss the additional findings below.

Figure 7.12(b) shows that churn shares an increasing relationship with the likelihood that a patch will receive slow initial feedback. The odds, produced by the OpenStack model, increases when churn increases from 1 to 150 LOC. Table 7.13 shows that when the churn increases greater than 5 LOC, the likelihood increases by 33% in the OpenStack models. However, Table 7.12 shows that churn did not contribute a significant amount of explanatory power to the Android and Qt models. Moreover, Table 7.13 shows that the partial effect of churn having on the likelihood in the Android model is 0%. To better understand this relationship, we further investigate with a one-tailed Mann-Whitney U test ($\alpha = 0.05$) comparing the churn of patches that receive slow initial feedback and those that receive prompt feedback in the Android dataset. We find that the churn of patches that receive slow initial feedback is statistically larger than churn of patches that receive prompt initial feedback, indicating that a patch with large churn tends to receive slow initial feedback ($p\text{-value} < 0.001$). Our results suggest that when we control for several confounding factors using prediction model, the churn did not share a strong relationship with the likelihood.

On the other hand, we find that the number of prior patches of an author shares a strong relationship with the speed of initial feedback in the Android model. Table 7.12 shows that the number of prior patches of an author contributes a

large amount of explanatory power, and its nonlinear relationship can boost to the explanatory power of the Android model. Table 7.13 shows that the likelihood decreases by 42% when the number of prior patches of an author increases from 3 to 109. We observe a similar relationship in the OpenStack dataset, i.e., the likelihood decrease by 4% when the number of prior patches of an author increases from 4 to 111. However, Table 7.13 shows that the likelihood increases when the number of prior patches of an author in the Qt dataset increases. One possible reason for the reverse effect of the number of prior patches of an author in the Qt dataset is the low number of developers in the sub-projects. This is especially true for the `qt3d` sub-project. We find that 52% of the Qt patches that are submitted to the `qt3d` sub-project received slow initial feedback (609/1,166). However, there are only 24 developers who author these patches. Hence, many of the `qt3d` patches have a large number of prior patches of an author while they received slow initial feedback.

Summary: The feedback delay of prior patches shares a strong relationship with the likelihood that a patch will receive slow initial feedback. Furthermore, the purpose of introducing new features can also increase the likelihood (Observations 32-33).

7.4. Discussion

In this section, we provide a broader discussion of our empirical observations. The observations are grouped into three main groups, which are the (1) past reviewer involvement, (2) past activities of practitioners, and (3) patch properties.

7.4.1. Past Reviewer involvement

The number of reviewers of prior patches. As we conjecture (see Table 7.2), observation 25 shows that the number of reviewers of prior patches share an in-

creasing relationship the likelihood of having at least one reviewer. We suspect that files have had few reviewers of prior patches in part due to a limited number of developers who are interested or working on related subsystems. Prior work reports that developers usually select patches that are within their area of interest [101]. For example, we find that 42% of the patches in the less popular subsystems (i.e., having fewer than 10 contributing developers) of Android contain files where their prior patches do not attract any reviewers. We also observe a similar proportion of 45% and 53% in the small subsystems of Qt and OpenStack. On the other hand, the subsystems that have a large number of developers (11 to 590 contributing developers) have only 8%, 3%, and 6% of patches containing files where their prior patches did not attract any reviewers in the Android, Qt, and OpenStack systems, respectively.

Moreover, our results indicate that patches are more likely to have at least one reviewer if past changes to the modified files have a tendency to be reviewed by at least two reviewers. This finding complements those of Rigby and Bird who report that two reviewers find an optimal number of defects in MCR processes [97].

Discussion length of prior patches. Observation 30 arrives at our conjecture where the discussion length shares an increasing relationship with the likelihood that a patch will be discussed. In other words, files that had little review discussion in the past will have little discussion in the future. Similar to prior studies, the discussion length is a strong indicator for the review quality. For example, Kononenko *et al.* report that the number of reviewer comments posted shares a significant link to the defect-proneness of that patch [65]. Our prior work also shows that files without post-release defects also undergo many reviews with long discussions [121]. In addition, this observation also complements the study of Bird *et al.* who find that the social status that is derived from the social network of the communication & co-ordination activities shares a strong relationship with the number of messages replied in the email-based discussions [20].

During the correlation analysis, we find that the number of reviewers of prior patches and the discussion length of prior patches are highly correlated with each other. Hence, we experimented by swapping these variables in order to validate our observations 25 and 30. We find that the number of reviewers of prior patches and the discussion length of prior patches share a similar relationship with the likelihood of receiving such poor reviewer involvement. Therefore, for observation 25, we can conclude that patches that modify files whose prior patches had received either few reviewers or short discussion tend to have no reviewers. For observation 30, we can conclude that patches that modify files whose prior patches had received either few reviewers or short discussion tend not to be discussed although there is at least one reviewer participated in. These findings suggest that the poor reviewer involvement in the past (either few reviewers or short discussion) can lead to the poor reviewer involvement in the future.

Feedback delay of prior patches. Observation 32 shows that the feedback delay of prior patches shares an increasing relationship with the likelihood that a patch that will receive slow initial feedback. Prior work shows that reviews often receive prompt initial feedback. Otherwise, patches tend to be ignored if they have not receive initial feedback for a long period of time [100].

Our observations 25, 30, and 32 have shown that the past reviewer involvement shares a link to poor reviewer involvement of a patch. In several studies, this notion of past-is-a-good-predictor drives the analytics research worldwide. For example, Graves *et al.* find that the number of past faults can be used as an indicator of future defects in a module [41]. Kim *et al.* have shown that the recently changed files are more likely to be changed to fix bugs in the future [62]. Tantithamthavorn *et al.* find that bug reporters who have mislabelled bug issue reports in the past tend to mislabel bug issue reports in the future [115]. da Costa *et al.* also use the delay of previously addressed issues to predict the delay of the future issues [30]. Furthermore, our prior study has found that practitioners tend to overlook the history of files, i.e., reviewers did not give much attention to

patches made to files that have been historically defective [121]. Therefore, our findings of past reviewer involvement metrics suggest that practitioners should take the history of files into consideration in order to break the cycle of poor reviewer involvement.

7.4.2. Past Activity

Past Involvement of an author and reviewers. Observation 31 shows that the past involvement of an author shares an inverse relationship with the likelihood that a patch will be discussed, while the past involvement of reviewers shares an increasing relationship with the likelihood that a patch will be discussed. The relationship of the past involvement of reviewers arrives at our conjecture. However, the relationship of the past involvement of authors is counter our conjecture. A potential reason is that the experience developers are less likely to submit defective patches as those developers have written many patches to those files [21].

Rigby *et al.* report that the past involvement of author and reviewers have a small effect on the amount of discussion in the email-based code reviews [99]. Yet, their study did not consider the patches that have no review discussion. Hence, this observation can complement the prior study by showing that the past involvement of an author and reviewers shares a link to the likelihood that a patch that will be discussed. Moreover, Kononenko *et al.* report that reviewer experience is a good indicator of whether the patch will be effectively reviewed [65].

Furthermore, the past involvement of reviewers has been recently used to suggest reviewers for a new patch in several studies [9, 124, 135], and also in commercial MCR tools such as CodeCollaborator [95]. Hence, observation 31 can support their approaches that inviting reviewers who have been involved in many reviews of the modified files can increase the likelihood that a patch will be discussed.

The number of days since the last modification. Our observation 26 shows that the number of days since the last modification shares an increasing relationship

with the likelihood that a patch will have at least one reviewer. This finding does not match our expectations. One possible reason for this relationship is that the last patch prior to the current patch is incomplete. For example, the Android patch of review ID 39850 was approved by a reviewer.⁹ Then, two hours later, the patch author submitted a new patch in order to make a change that was suggested in review ID 39850.¹⁰ We find similar examples in the Qt and OpenStack systems.¹¹ Although such patches can be minor changes or may not need much involvement from reviewers, prior work suggests that the patches still should be examined by at least one reviewers to decrease the likelihood of having defects in the future [12].

7.4.3. Patch Properties

Patch Description. As we conjecture, observation 27 shows that the description length shares an increasing relationship with the likelihood that a patch will have at least one reviewer. Similarly, observation 29 shows that a short description can lower the likelihood that a patch will be discussed. We find that most of the patches with short descriptions provide neither the details that are necessary to understand the proposed changes nor a link for additional information. For example, the patch author of OpenStack review ID 29856 did not describe the detail of a change.¹² This finding is consistent with prior studies of email-based code reviews, i.e., a descriptive subject and a well-explained change log message are very important information for developers to select patches to review [101]. Moreover, Tao *et al.* report that one of the most important pieces of information for reviewers is a description of the rationale of a change [117].

⁹<https://android-review.googlesource.com/#/c/39850>

¹⁰<https://android-review.googlesource.com/#/c/39881>

¹¹An example in the Qt system: <https://codereview.qt-project.org/#/c/27218> and <https://codereview.qt-project.org/#/c/30591>. An example in the OpenStack system: <https://review.openstack.org/#/c/36808> and <https://review.openstack.org/#/c/36832>.

¹²<https://review.openstack.org/#/c/36901/>

Observation 33 shows that as we conjecture, a patch that introduces new functionality is more likely to receive slow initial feedback than a patch with another purposes. A potential reason for this delayed feedback in patches that introduce new features could be that such patches require more effort to understand. Intuitively, a documentation patch would be easy to understand, and could receive prompt feedback. We also observe that the reviewers of the bug-fixing patches often are reporters in the Issue Tracking System (ITS). For example, Qt review ID 29856 is made to address the bug ID 22625.¹³ We find that the reviewer of this patch is the one who reports the bug.¹⁴ Hence, it is more likely that the bug fix reviewers would already understand the problem and could provide prompt feedback to the author. On the other hand, the purpose of patches that introduce new functionality must be entirely built from the patch description and source code. For example, the patch author in Qt review ID 101316 implements a new function to `QGeoShape`.¹⁵ Hence, there are likely cases that reviewers will require longer time to understand before providing initial feedback. These observations suggest that to increase reviewer involvement, an author should provide a detailed description of their proposed changes in order to help reviewers to understand which problem it fixes or how the new feature is supposed to work. Then, reviewers can either provide feedback if they have the expertise to do so or suggest appropriate reviewers.

Patch Size. Observation 28 shows that as we conjecture, churn shares an increasing relationship with the likelihood that a patch will be discussed. Recent studies also report that the patch size is a good indicator of patch acceptance [57, 131]. Intuitively, the large patches are likely to be discussed since they can contain more problems than the small patches. For example, OpenStack review ID 35074 shows a review where an author proposes a change of 737 LOC, then the review-

¹³<https://codereview.qt-project.org/#/c/29856>

¹⁴<https://bugreports.qt.io/browse/QTBUG-22625>

¹⁵<https://codereview.qt-project.org/#/c/101316>

ers raised several issues.¹⁶ On the other hand, smaller patches have less code to critique, and thus are less likely to receive comments from reviewers.¹⁷ This observation is also consistent with the findings of Baysal *et al.* who find that large patches tend to have more revisions than small patches in the MCR processes of the WebKit and Blink systems [15].

7.5. Threats to Validity

We now discuss threats to the validity of our study.

7.5.1. External Validity

We focus our study on three open source systems, which may limit the generalizability of our results. Nagappan *et al.* argue that increasing the sample size without careful selection cannot contribute to the goal of increased generality [83]. Hence, it is a challenge to carefully identify systems that satisfy our selection criteria (*cf.* Section 7.2.1), since the code review process of MCR is a relatively new development. To aid in future work, we make our datasets publicly available.¹⁸ Nonetheless, additional replication studies are needed to generalize our results.

7.5.2. Construct Validity

We identify the purpose of a patch by extracting keywords from its commit message. Although modern Issue Tracking Systems (ITSs) provide a field for practitioners to denote the purpose of a change, we find that our studied systems have a small proportion of patches that can be linked to records in ITSs. Indeed, only 9%, 1%, and 19% of the studied patches can be linked to issues in the ITS records of our studied systems. Hence, we must rely on heuristics to recover this

¹⁶<https://review.openstack.org/#/c/35074/>

¹⁷<https://review.openstack.org/#/c/36448/>

¹⁸http://sailhome.cs.queensu.ca/replication/review_participation/

information. Nevertheless, we measure the accuracy of our purpose identification by manually examining samples of patches. From a sample of 50 patches for each type of purpose, we find that on average, 89% of patches are correctly identified as feature introduction, 91% of patches are correctly identified as bug-fixing, and 75% of patches are correctly identified as documentation.

We measure feedback delay based on a heuristic that reviewers will promptly review a new patch at the time that it is submitted. However, there are likely cases where reviewers actually examined a patch for a fraction of this timeframe. Unfortunately, the Gerrit code review tool does not record the timestamps when reviewers start to examine a patch nor the timezone information. To reduce such measurement errors, we use the elapsed time between the latest revision before receiving the initial feedback and the posting of the initial feedback. In addition, the threshold of feedback delay may also be impacted by timezone difference. Hence, we check whether the threshold is impacted by timezone difference (see Appendix D). We find that timezone difference is less likely to have an impact on the threshold of feedback delay.

7.5.3. Internal Validity

We assume that our studied systems perform code reviews using the MCR tools. There are likely cases where the reviewing activities are missing. For example, Mukadam *et al.* report that many reviews in the Android system are missing since August 2011 until early January 2012 [82]. It is also possible that reviewers may provide feedback using other communication media, such as in-person discussion [16], a group IRC [108], or mailing list [45, 101]. Unfortunately, recovering these reviewing activities is a non-trivial problem [7, 19]. However, our analysis focuses on reviews that were submitted during the period when the studied systems actively use MCR tools (*cf.* Section 7.2.2). Hence, we rely on the information that is recorded by these tools.

Since our observations are based on the surviving explanatory variables, there are likely cases that our variable selection may influence our conclusions. Hence, for the sake of completeness, we change each surviving variable to the other variables that were removed and refit our models. Our model analysis results indicate that these alternate models achieve similar AUC values, i.e., the AUC differences are ranging between -0.01 to 0.04 for models in RQ1, -0.01 to 0.02 for models in RQ2, and -0.01 to 0.01 for models in RQ3. Furthermore, we find that if the surviving variables have a large effect on the likelihood, the alternate variables will have a large effect as well. For example, Table 7.6 shows that the number of reviewers of prior patches has the largest negative effect on the likelihood that a patch will not attract reviewer. We also find that the discussion length of prior patches has the largest negative effect on the likelihood in the alternate model (i.e., the variable that is highly correlated with the number of reviewers of prior patches). Moreover, the variables that are highly correlated in our study measure similar characteristic of a patch, e.g., the discussion length of prior patches and the number of reviewers measure the typical level of reviewer involvement in prior patches. Therefore, we believe that our variable selection did not muddle our conclusions.

We assume that the review processes are consistent across all subsystems in a large system. Future work should closely examine whether there are differences in review processes across subsystems.

7.6. Summary

Due to the human-intensive nature of code reviewing, reviewer involvement plays an important role in Modern Code Review (MCR) practices. Despite the importance of reviewer involvement [73, 81, 121], little is known about the factors that influence reviewer involvement in the MCR process.

In this chapter, we investigate the characteristics of patches that: do not attract

reviewers, are not discussed, or receive slow initial feedback in the MCR process. We measure 20 patch and MCR process metrics that are grouped along five dimensions. We use contemporary regression modelling techniques to investigate the relationship that our metrics share with the likelihood that a patch will suffer from poor reviewer involvement. Using data collected from the Android, Qt, and OpenStack open source systems, we empirically study 196,712 code reviews. The results of our study show that our models can identify patches that fill suffer from poor reviewer involvement with an AUC ranging 0.61-0.76. Moreover, we make the following observations:

- The number of reviewers of prior patches, and the description length share a strong inverse relationship with the likelihood that a patch will not attract any reviewers. Counter-intuitively, the number of days since the last modification of files also share an inverse relationship. (Observations [25](#) to [27](#)).
- The description length, the discussion length of prior patches, churn, and past involvement of reviewers share an inverse relationship with the likelihood that a patch will not be discussed, while past involvement of an author shares an increasing relationship with that likelihood (Observations [28](#) to [31](#)).
- The feedback delay of prior patches shares a strong increasing relationship with the likelihood that a patch will receive slow initial feedback. Moreover, patches that introduce new features tend to receive slower feedback than patches that fix bugs or address documentation issues (Observations [32](#) to [33](#)).

We believe that our results and empirical observations help to support the management of the MCR process and adherence to our recommendations will lead to a more responsive review process.

Part IV.

Conclusion and Future Work

Conclusion

Code review is a well-established software quality practice. Lately, the lightweight variant of code review process called Modern Code Review (MCR) has been widely adopted in many modern software organizations. Unlike the formal software inspection of the past, MCR enables asynchronous collaboration among developers at the globally-distributed teams of modern software organizations. Moreover, MCR becomes a final gate of quality control as MCR process is based on collaborative tools which tightly integrate with the project's Version Control System (VCS)

Despite the flexibility of code reviews that MCR provides, MCR lacks mechanisms for ensuring a base level of review quality. Lax reviewing practices may creep into MCR processes, which can impact software quality. In this thesis, we empirically study the suboptimal reviewing practices in MCR processes. In the remainder of this chapter, we outline the contributions of this thesis and lay out opportunities for future research.

8.1. Contribution and Findings

The overarching goal of this thesis is to better understand reviewing practices and provide MCR process support. To achieve this goal, we perform a series of

empirical studies on historical data in order to understand how teams can better perform code reviews in an MCR context. Broadly speaking, we find that:

Thesis Statement: The lightweight Modern Code Review process lacks mechanisms for preventing lax code reviewing practices, which can allow poor quality code to slip through to official software releases. A deeper understanding of these processes and their impact on software quality is needed as MCR continues to gain popularity and be adopted by projects worldwide.

We perform four empirical studies to investigate the suboptimal reviewing practices in the review preparation and executions steps of MCR processes. Below, we reiterate the empirical observations of this thesis:

1. Part I: The Modern Code Review Processes

- (a) Developers who only review patches of other developers account for the largest proportion of developers who contribute to that module (Observations 1 and 2). Moreover, many developers who were considered as minor contributors are actually major contributors who review a large proportion of patches (Observation 3).

Suggestion. Code review activity should be considered in order to thoroughly model expertise of developers.

- (b) Modules without post-release defects tend to have a larger proportion of developers with major reviewing expertise than modules with post-release defects do (Observations 4 and 5). On the other hand, the proportion of developers who have neither authored nor reviewed many patches to a module share a strong increasing relationship with defect-proneness for the module (Observations 6 and 7).

Suggestion. Teams should consider reviewing expertise in order to mitigate the risk of future defects.

- (c) Patch authors often have difficulties selecting appropriate reviewers for their patches. Moreover, patches where an author could not initially find appropriate reviewers tend to take longer time in MCR processes than patches without such a problem (Observations 8 and 9).

Implication. Developers are suffering from selecting an appropriate reviewer for a new patch.

- (d) REVFinder, which leverages the similarities between the path of a newly changed file and the paths of previously reviewed files to recommend a reviewer, provides a high rank of correct reviewers, which is less likely to involve unrelated reviewers (Observations 10 and 11). Moreover, REVFinder can accurately recommend appropriate reviewers within its top-5 recommendations (Observation 12).

Suggestion. REVFinder should be included into MCR tools in order to help developers find appropriate reviewers and speed up the overall process.

2. Part II: Reviewer Selection in Modern Code Review Processes.

- (a) The code review activity of files that will be defective in the future, i.e., future-defective files tends to be less intense with less team participation and with a faster rate of code examination than the reviews of clean files (Observations 13 to 15). Moreover, developers more often address concerns about: (1) documentation and structure to enhance evolvability, and (2) checks, logic, and support to fix functionality issues, in the reviews of future-defective files than the reviews of clean files (Observations 16 to 17).

Suggestion. Reviewers should actively be involved in their assigned code reviews in order to reduce the likelihood of having future defects.

- (b) The code review activity of files that have been historically defective in the past, i.e., risky files tends to be less intense, with less team

participation, and receive feedback more slowly than files that have historically been defect-free, i.e., normal files (Observations 18 to 20). Developers address concerns about evolvability and functionality in the reviews of risky files more often than the reviews of normal files do (Observations 21 to 22). Moreover, risky files that will have future defects tend to undergo less careful reviews that more often address concerns about evolvability than the reviews of risky files without future defects (Observations 23 to 24).

Suggestion. Risky files are more likely to have future defects [41]. Hence, a more careful attention is needed for such files during the code review process.

- (c) Past reviewer involvement (e.g., the number of reviewers, discussion length of prior patches), the description length, the number of days since the last modification of files, the past involvement of an author, and the past involvement of reviewers are strong indicators that a patch will suffer from poor reviewer involvement (Observations 25 to 33).

Suggestion. Patch submission policies that monitor these factors are urgently needed in order to help improve the reviewer involvement in MCR processes.

8.2. Opportunities for Future Research

We believe that this thesis makes a major contribution towards the goal of improving the reviewing practices in an MCR context. However, there are many open challenges for future research. Below, we outline several opportunities for future work.

8.2.1. More Advanced Reviewing Expertise Heuristics

In the studies of Chapters 4 and 5, we estimate the reviewing expertise based on the review participation in the past. In other words, the more patches that a developer participated in the past, the more reviewing expertise the developer has. However, there is a likely case that some of the identified reviewers may only leave superficial or unrelated reviewer comments [26,88]. Although we attempted to mitigate this risk by allocating less expertise for reviewers who provide less feedback, the results of the study in Chapter 4 show that this heuristic did not show different results from the simpler heuristic. Thus, future work should explore more advanced reviewer contribution heuristics in order to more accurately model the reviewing expertise of developers.

8.2.2. A Reviewer Recommendation with Workload Balancing

In Chapter 5, we proposed REVFinder to identify appropriate reviewers based on their reviewing expertise. Although the evaluation results show that REVFinder accurately recommends reviewers for a new patch, there is a likely case that reviewers may be burdened with a large number of assigned reviews according to REVFinder's recommendations. Therefore, future work of a reviewer recommendation should consider the workload of reviewers in order to avoid the risk that reviewers will be burdened by review tasks and that the patches will receive slow review feedback.

8.2.3. Automated MCR Practices Detection

In Chapter 5, we manually identify patches which have reviewer selection problems. Similarly, in Chapter 6, we manually identify concerns that were addressed during MCR reviews. However, we used small sets of statistically representative sample for our analyses. An automatic detection for these patches could ease

future studies to use a large set of dataset and to draw a more generalized conclusion. Moreover, develop teams can use this tool to monitor an overview of MCR practices as well.

8.2.4. Understanding Source Code Changes under MCR processes

In Chapter 6, we focus on code review activity and reviewing concerns that were addressed during MCR processes. Yet, it is unclear how is a proposed patch semantically changes under an MCR process. A better understanding of semantical changes under MCR processes may help improve the quality of code reviews as well as software quality.

8.2.5. Understanding Reviewer Factors in MCR

In Chapter 7, we uncovered several patch characteristics like past tendencies and patch description that lead to a poor reviewer involvement. However, the reviewer factors that may also have an impact on reviewer involvement remains largely explored. Although we explored the past reviewer involvement, the other reviewer factors like review workload of reviewers and the size of team may also influence the reviewer involvement. Furthermore, developers who are hired by third-party companies may perform different code reviewing from volunteer developers. To better understand the MCR practices, future research should explore more factors related to reviewers.

8.2.6. Revisiting Code Review Studies when Considering the Importance of Software Modules

In Chapters 4 and 6, we assume that each post-release defect is of the same weight, while in reality, some post-release defects are more severe than others. Similarly,

in Chapter 7, we did not consider the importance of software modules that were impacted by studied patches. However, there is a likely case that patches that modify important software modules will receive more responsive and rigorous reviewer involvement than other patches. Hence, future work should consider the severity of post-release defects and the importance of software modules in order to draw a more clear relationship between developer expertise and software quality, and a more clear relationship between patch characteristics and reviewer involvement.

8.2.7. Revisiting Code Review Studies when using different MCR tools

The case studies in Chapters 4, 5, 6, and 7 are based on Gerrit. Although we used several open source systems as our case studies, our observations may not be generalized to all software systems and MCR tools. Each software organization may use different policies which lead to different reviewing practices. For example, in Chapter 6, we manually identify concerns that were addressed through both in-line comments and discussion threads. However, some MCR tools may provide a different feature other than in-line commenting. Moreover, the reviewing practices in industrial settings may be more rigorous than the reviewing practices in open source projects. To better generalize the observations in this thesis, future research should revisit code review studies that are used in other systems with different MCR tools.

8.2.8. Revisiting Code Review Studies by using Qualitative Analysis

The observations in this thesis are derived from the results of quantitative analysis, i.e., statistical analysis approach. However, the reasons behind these observa-

tions remain largely unexplored. Although we further investigated these findings by manually examine code reviews of patches, this investigation did not involve with developers who are in the studied projects. To fulfill the gap in knowledge, future research should revisit code review studies by using qualitative analysis, e.g., developer survey.

8.2.9. Studying the Impact of MCR on Code Quality

In this thesis, we study the impact that MCR can have on software quality. However, code reviews may also improve the code quality. For example, in Chapter 6, we found that future maintenance is one of the main concerns that were addressed during code reviews. Hence, a better understand of benefits of MCR in terms of code quality may help team to chart a better quality improvement plan.

References

- [1] U. Abelein and B. Paech. Understanding the Influence of User Participation and Involvement on System Success – a Systematic Mapping Study. *Empirical Software Engineering (EMSE)*, 20(1):28–31, 2013.
- [2] A. F. Ackerman, L. S. Buchwald, and F. H. Lewski. Software Inspections: An Effective Verification Process. *IEEE Software*, 6(3):31–36, 1989.
- [3] J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In *Proceedings of the 28th International Conference on Software Engineering (ICSE)*, pages 361–370, 2006.
- [4] J. Asundi and R. Jayant. Patch Review Processes in Open Source Software Development Communities: A Comparative Case Study. In *Proceedings of the 40th Annual Hawaii International Conference on System Sciences (HICSS)*, page 166c. Ieee, 2007.
- [5] A. Aurum, H. Petersson, and C. Wohlin. State-of-the-Art: Software Inspections After 25 Years. *Software Testing, Verification and Reliability*, 12(3):133–154, sep 2002.
- [6] A. Bacchelli and C. Bird. Expectations, Outcomes, and Challenges Of Modern Code Review. In *Proceedings of the 35th International Conference on Software Engineering (ICSE)*, pages 712–721, 2013.

- [7] A. Bacchelli, M. Lanza, and R. Robbes. Linking E-Mails and Source Code Artifacts. In *Proceedings of the 32nd International Conference on Software Engineering (ICSE)*, pages 375–384, 2010.
- [8] R. A. Baker. Code Reviews Enhance Software Quality. In *Proceedings of the 19th international conference on Software engineering (ICSE)*, pages 570–571, 1997.
- [9] V. Balachandran. Reducing Human Effort and Improving Quality in Peer Code Reviews using Automatic Static Analysis and Reviewer Recommendation. In *Proceedings of the 35th International Conference on Software Engineering (ICSE)*, pages 931–940, 2013.
- [10] M. Barnett, C. Bird, J. Brunet, and S. K. Lahiri. Helping Developers Help Themselves: Automatic Decomposition of Code Review Changesets. In *Proceedings of the 37th International Conference on Software Engineering (ICSE)*, pages 134–144, 2015.
- [11] E. T. Barr, C. Bird, P. C. Rigby, A. Hindle, D. M. German, and P. Devanbu. Cohesive and Isolated Development with Branches. In *Proceedings of the 20th International Symposium on the Foundations of Software Engineering (FASE)*, pages 316–331, 2012.
- [12] G. Bavota and B. Russo. Four Eyes Are Better Than Two: On the Impact of Code Reviews on Software Quality. In *Proceedings of the 31st International Conference on Software Maintenance and Evolution (ICSME)*, pages 81–90, 2015.
- [13] O. Baysal, O. Kononenko, R. Holmes, and M. W. Godfrey. The Secret Life of Patches: A Firefox Case Study. In *Proceedings of the 19th Working Conference on Reverse Engineering (WCRE)*, pages 447–455, oct 2012.

- [14] O. Baysal, O. Kononenko, R. Holmes, and M. W. Godfrey. The Influence of Non-Technical Factors on Code Review. In *Proceeding of 20th Working Conference on Reverse Engineering (WCRE)*, pages 122–131, 2013.
- [15] O. Baysal, O. Kononenko, R. Holmes, and M. W. Godfrey. Investigating Technical and Non-Technical Factors Influencing Modern Code Review. *Empirical Software Engineering (EMSE)*, page to appear, 2015.
- [16] M. Beller, A. Bacchelli, A. Zaidman, and E. Juergens. Modern Code Reviews in Open-Source Projects: Which Problems Do They Fix? In *Proceedings of the 11th International Working Conference on Mining Software Repositories (MSR)*, pages 202–211, 2014.
- [17] N. Bettenburg, A. E. Hassan, B. Adams, and D. M. German. Management of community contributions. *Empirical Software Engineering (EMSE)*, 20(1):252–289, 2015.
- [18] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu. Fair and Balanced? Bias in Bug-Fix Datasets. In *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the International Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 121–130, 2009.
- [19] C. Bird, A. Gourley, and P. Devanbu. Detecting Patch Submission and Acceptance in OSS Projects. In *Proceedings of the 4th International Working Conference on Mining Software Repositories (MSR)*, pages 26–29, 2007.
- [20] C. Bird, A. Gourley, P. Devanbu, M. Gertz, and A. Swaminathan. Mining Email Social Networks. In *Proceedings of the 3rd International Workshop on Mining Software Repositories (MSR)*, pages 137–143, 2006.
- [21] C. Bird, N. Nagappan, B. Murphy, H. Gall, and P. Devanbu. Don’t Touch My Code! Examining the Effects of Ownership on Software Quality. In

- Proceedings of the 8th joint meeting of the European Software Engineering Conference and the International Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 4–14, 2011.
- [22] B. Boehm and V. R. Basili. Software Defect Reduction Top 10 List. *Computer*, 34(1):135–137, 2001.
- [23] A. Bosu and J. C. Carver. Impact of Peer Code Review on Peer Impression Formation: A Survey. In *Proceedings of the 7th International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 133–142. Ieee, oct 2013.
- [24] A. Bosu and J. C. Carver. Impact of Developer Reputation on Code Review Outcomes in OSS Projects : An Empirical Investigation. In *Proceedings of the 8th International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 33–42, 2014.
- [25] A. Bosu, J. C. Carver, M. Hafiz, P. Hilley, and D. Janni. Identifying the Characteristics of Vulnerable Code Changes: An Empirical. In *Proceedings of the 22nd International Symposium on the Foundations of Software Engineering (FSE)*, pages 257–268, 2014.
- [26] A. Bosu, M. Greiler, and C. Bird. Characteristics of Useful Code Reviews: An Empirical Study at Microsoft. In *Proceedings of the 12th International Working Conference on Mining Software Repositories (MSR)*, pages 146–156, 2015.
- [27] I. T. Bowman, R. C. Holt, and N. V. Brewster. Linux as a case study: Its extracted software architecture. In *Proceedings of the 21st International Conference on Software Engineering (ICSE)*, pages 555–563, 1999.
- [28] B. Caglayan, B. Turhan, A. Bener, M. Habayeb, A. Miransky, and E. Cialini. Merits of Organizational Metrics in Defect Prediction: An In-

- dustrial Replication. In *Proceedings of the 37th International Conference on Software Engineering (ICSE)*, pages 89–98, 2015.
- [29] M. Cataldo, A. Mockus, J. A. Roberts, and J. D. Herbsleb. Software Dependencies, Work Dependencies, and Their Impact on Failures. *Transactions on Software Engineering (TSE)*, 35(6):864–878, 2009.
- [30] D. A. da Costa, S. L. Abebe, S. McIntosh, U. Kulesza, and A. E. Hassan. An Empirical Study of Delays in the Integration of Addressed Issues. In *Proceedings of the 30th International Conference on Software Maintenance and Evolution (ICSME)*, pages 281–290, 2014.
- [31] A. W. F. Edwards. The Measure of Association in a 2 x 2 Table. *Journal of the Royal Statistical Society. Series A (General)*, 126(1), 1963.
- [32] B. Efron. How Biased is the Apparent Error Rate of a Prediction Rule? *Journal of the American Statistical Association*, 81(394):461–470, 1986.
- [33] M. E. Fagan. Advances in software inspections. *Transactions on Software Engineering (TSE)*, 12(7):744–751, 1986.
- [34] M. E. Fagan. Design and Code Inspections to Reduce Errors in Program Development. *IBM System Journal*, 38(2-3):258–287, jun 1999.
- [35] A. L. Ferreira, R. J. Machado, L. Costa, J. G. Silva, R. F. Batista, and M. C. Paulk. An Apporach to Improving Software Inspections Performance. In *Proceeding of the International Conference on Software Maintenance (ICSM)*, pages 1–8, 2010.
- [36] M. Fowler. Continuous Integration, 2006.
- [37] T. Fritz, G. C. Murphy, and E. Hill. Does a Programmer’s Activity Indicate Knowledge of Code? In *Proceedings of the 6th joint meeting of the European*

- Software Engineering Conference and the International Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 341–350, 2007.
- [38] T. Fritz, J. Ou, G. C. Murphy, and E. Murphy-Hill. A Degree-of-Knowledge Model to Capture Source Code Familiarity. In *Proceedings of the 32nd International Conference on Software Engineering (ICSE)*, pages 385–394, 2010.
- [39] A. Gabadinho, G. Ritschard, N. S. Muller, and M. Studer. Analyzing and Visualizing State Sequences in R with TraMineR. *Journal of Statistical Software*, 40(4):1–37, 2011.
- [40] G. Gousios, M. Pinzger, and A. van Deursen. An Exploratory Study of the Pull-based Software Development Model. In *Proceedings of the 36th International Conference on Software Engineering (ICSE)*, pages 345–355, 2014.
- [41] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy. Predicting Fault Incidence Using Software Change History. *Transactions on Software Engineering (TSE)*, 26(7):653–661, 2000.
- [42] M. Greiler, K. Herzig, and J. Czerwonka. Code Ownership and Software Quality: A Replication Study. In *Proceedings of the 12th International Working Conference on Mining Software Repositories (MSR)*, pages 2–12, 2015.
- [43] Z. Guan and E. Cutrell. An Eye Tracking Study of the Effect of Target Rank on Web Search. In *Proceedings of the 25th Conference on Human Factors in Computing Systems (CHI)*, pages 417–420, 2007.
- [44] D. Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. 1997.

-
- [45] A. Guzzi, A. Bacchelli, M. Lanza, M. Pinzger, and A. Van Deursen. Communication in Open Source Software Development Mailing Lists. In *Proceedings of the 10th International Working Conference on Mining Software Repositories (MSR)*, pages 277–286, 2013.
 - [46] K. Hamasaki, R. G. Kula, N. Yoshida, C. C. A. Erika, K. Fujiwara, and H. Iida. Who does what during a Code Review? An extraction of an OSS Peer Review Repository. In *Proceedings of the 10th International Working Conference on Mining Software Repositories (MSR)*, pages 49–52, 2013.
 - [47] J. A. Hanley and B. J. McNeil. The Meaning and Use of the Area under a Receiver Operating Characteristic (ROC) Curve. *Radiological Society of North America*, 143(1):29–36, 1982.
 - [48] F. E. Harrell Jr. *Regression Modeling Strategies: With Application to Linear Models, Logistic Regression, and Survival Analysis*. Springer, 1st edition, 2002.
 - [49] F. E. Harrell Jr. rms: Regression Modeling Strategies, 2015.
 - [50] A. E. Hassan. Automated Classification of Change Messages in Open Source Projects. In *Proceedings of the 23rd Symposium on Applied Computing (SAC)*, pages 837–841, 2008.
 - [51] A. E. Hassan. Predicting Faults Using the Complexity of Code Changes. In *Proceedings of the 31st International Conference on Software Engineering (ICSE)*, pages 78–88, 2009.
 - [52] L. P. Hattori, M. Lanza, and R. Robbes. Refining code ownership with synchronous changes. *Empirical Software Engineering (EMSE)*, 17(4-5):467–499, 2012.

-
- [53] A. Hindle, D. M. German, M. W. Godfrey, and R. C. Holt. Automatic Classification of Large Changes into Maintenance Categories. In *Proceedings of the 17th International Conference on Program Comprehension (ICPC)*, pages 30–39, 2009.
 - [54] D. E. Hinkle, W. Wiersma, and S. G. Jurs. *Applied statistics for the behavioral sciences*. Houghton Mifflin Boston, MA, 4th edition, 1998.
 - [55] T. K. Ho, J. J. Hull, and S. N. Srihari. Decision Combination in Multiple Classifier Systems. *Transactions on Pattern Analysis and Machine Intelligence*, 16(1):66–75, 1994.
 - [56] J. Jiang, J. H. He, and X. Y. Chen. CoreDevRec: Automatic Core Member Recommendation for Contribution Evaluation. *Journal of Computer Science and Technology*, 30(5):998–1016, 2015.
 - [57] Y. Jiang, B. Adams, and D. M. German. Will My Patch Make It? And How Fast? Case Study on the Linux Kernel. In *Proceeding of the 10th IEEE Working Conference on Mining Software Repositories (MSR)*, pages 101–110, 2013.
 - [58] 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. *Transactions on Software Engineering (TSE)*, 39(6):757–773, 2013.
 - [59] P. Kampstra. Beanplot: A Boxplot Alternative for Visual Comparison of Distributions. *Journal of Statistical Software*, 28(1):1–9, 2008.
 - [60] C. F. Kemerer, I. C. Society, M. C. Paulk, and S. Member. The Impact of Design and Code Reviews on Software Quality: An Empirical Study Based on PSP Data. *Transactions on Software Engineering (TSE)*, 35(4):534–550, 2009.

-
- [61] S. Kim, J. Whitehead Jr., and Y. Zhang. Classifying Software Changes: Clean or Buggy? *Transactions on Software Engineering (TSE)*, 34(2):181–196, 2008.
 - [62] S. Kim, T. Zimmermann, J. Whitehead Jr., and A. Zeller. Predicting Fault from Cached History. In *Proceedings of the 29th International Conference on Software Engineering (ICSE)*, pages 489–498, 2007.
 - [63] L. Kish. *Survey sampling*. John Wiley and Sons, 1965.
 - [64] J. Kittler, M. Hater, and R. P. W. Duin. Combining classifiers. *Transactions on Pattern Analysis and Machine Intelligence*, 20(3):226–239, 1998.
 - [65] O. Kononenko, O. Baysal, L. Guerrouj, Y. Cao, and M. W. Godfrey. Investigating Code Review Quality: Do People and Participation Matter? In *Proceedings of the 31st International Conference on Software Maintenance and Evolution (ICSME)*, pages 111–120, 2015.
 - [66] H. C. Kraemer, G. A. Morgan, N. L. Leech, J. A. Gliner, J. J. Vaske, and R. J. Harmon. Measures of Clinical Significance. *Journal of the American Academy of Child & Adolescent Psychiatry*, 42(12):1534–1529, 2003.
 - [67] S. K. Lwanga and S. Lemeshow. *Sample size determination in health studies: A Practical Manual*. Geneva: World Health Organization, 1991.
 - [68] D. Ma, D. Schuler, T. Zimmermann, and J. Sillito. Expert Recommendation with Usage Expertise. In *Proceeding of the 25th International Conference on Software Maintenance (ICSM)*, pages 535–538, 2009.
 - [69] G. Macbeth, E. Razumiejczyk, and R. D. Ledesma. Cliff’s Delta Calculator: A Non-parametric Effect Size Program for Two Groups of Observations. *Universitas Psychologica*, 10:545–555, 2011.

- [70] M. V. Mäntylä and C. Lassenius. What Types of Defects Are Really Discovered in Code Reviews? *Transactions on Software Engineering (TSE)*, 35(3):430–448, 2009.
- [71] V. Mashayekhi, J. M. Drake, W.-T. Tsai, and J. Riedl. Distributed, Collaborative Software Inspection. *IEEE Software*, 10(5):66–75, 1993.
- [72] M. Masudur, R. Chanchal, and J. 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 SEIP)*, pages 222–231, 2016.
- [73] S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan. The Impact of Code Review Coverage and Code Review Participation on Software Quality. In *Proceedings of the 11th International Working Conference on Mining Software Repositories (MSR)*, pages 192–201, 2014.
- [74] S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan. An Empirical Study of the Impact of Modern Code Review Practices on Software Quality. *Empirical Software Engineering (EMSE)*, page to appear, 2015.
- [75] A. Meneely, A. C. R. Tejada, B. Spates, S. Trudeau, D. Neuberger, K. Whitlock, C. Ketant, and K. Davis. An Empirical Investigation of Socio-technical Code Review Metrics and Security Vulnerabilities. In *Proceedings of the 6th International Workshop on Social Software Engineering (SSE)*, pages 37–44, 2014.
- [76] A. Meneely and L. Williams. Secure Open Source Collaboration: An Empirical Study of Linus’ Law. In *Proceedings of the 16th Conference on Computer and Communications Security (CCS)*, pages 453–462, 2009.
- [77] B. Meyer. Design and code reviews in the age of the internet. *Communications of the ACM*, 51(9):66–71, 2008.

- [78] R. Mishra and A. Sureka. Mining Peer Code Review System for Computing Effort and Contribution Metrics for Patch Reviewers. In *Proceedings of the 4th Workshop on Mining Unstructured Data (MUD)*, pages 11–15, 2014.
- [79] A. Mockus and J. D. Herbsleb. Expertise Browser: A Quantitative Approach to Identify Expertise. In *Proceedings of the 24th International Conference on Software Engineering (ICSE)*, pages 503–512, 2002.
- [80] A. Mockus and L. G. Votta. Identifying Reasons for Software Changes using Historic Databases. In *Proceedings of the 16th International Conference on Software Maintainance (ICSM)*, pages 120–130, 2000.
- [81] 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 *Proceedings of the 22nd International Conference on Software Analysis, Evolution, and Reengineering (SAnER)*, pages 171–180, 2015.
- [82] M. Mukadam, C. Bird, and P. C. Rigby. Gerrit Software Code Review Data from Android. In *Proceedings of the 10th Working Conference on Mining Software Repositories (MSR)*, pages 45–48, 2013.
- [83] M. Nagappan, T. Zimmermann, and C. Bird. Diversity in Software Engineering Research. In *Proceedings of the 9th joint meeting of the European Software Engineering Conference and the International Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 466–476, 2013.
- [84] N. Nagappan and T. Ball. Use of Relative Code Churn Measures to Predict System Defect Density. In *Proceeding of the 27th International Conference on Software Engineering (ICSE)*, pages 284–292, 2005.
- [85] N. Nagappan, B. Murphy, and V. R. Basili. The Influence of Organizational Structure on Software Quality: An Empirical Case Study. In *Proceedings*

- of the 30th International Conference on Software Engineering (ICSE)*, pages 521–530, 2008.
- [86] N. Nagappan, A. Zeller, T. Zimmermann, K. Herzig, and B. Murphy. Change Bursts as Defect Predictors. In *Proceedings of the 21st International Symposium on Software Reliability Engineering (ISSRE)*, pages 309–318. Ieee, nov 2010.
- [87] M. Nurolahzade, S. M. Nasehi, S. H. Khandkar, and S. Rawal. The Role of Patch Review in Software Evolution: An Analysis of the Mozilla Firefox. In *Proceedings of the join International and Annual ERCIM Workshops on Principles of Software Evolution and Software Evolution Workshop (IWPSE-Evol)*, pages 9–17, 2009.
- [88] T. Pangsakulyanont, P. Thongtanunam, D. Port, and H. Iida. Assessing MCR Discussion Usefulness using Semantic Similarity. In *Proceedings of the 6th International Workshop on Empirical Software Engineering in Practice (IWESEP)*, pages 49–54, 2014.
- [89] S. Panichella, V. Arnaoudova, M. D. Penta, and G. Antoniol. Would Static Analysis Tools Help Developers with Code Reviews? In *Proceeding of the 22nd International Conference on Software Analysis, Evolution, and Reengineering (SAnER)*, page to be appear, 2015.
- [90] M. Pinzger, N. Nagappan, and B. Murphy. Can Developer-Module Networks Predict Failures? In *Proceedings of the 16th International Symposium on Foundations of Software Engineering (FSE)*, pages 2–12, 2008.
- [91] A. Porter, H. Siy, A. Mockus, and L. Votta. Understanding the Sources of Variation in Software Inspections. *Transactions on Software Engineering and Methodology (TOSEM)*, 7(1):41–79, jan 1998.

-
- [92] A. A. Porter, H. P. Siy, C. A. Toman, and L. G. Votta. An Experiment to Assess the Cost-Benefits of Code Inspections in Large Scale Software Development. *Transactions on Software Engineering (TSE)*, 23(6):329–346, 1997.
 - [93] F. Rahman and P. Devanbu. Ownership, Experience and Defects: A Fine-Grained Study of Authorship. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE)*, pages 491–500, 2011.
 - [94] R. Ranawana and V. Palade. Multi-Classifer Systems - Review and a Roadmap for Developers. *International Journal of Hybrid Intelligent Systems*, 3(1):1–41, 2006.
 - [95] J. G. Ratcliffe. Moving Software Quality Upstream: The Positive Impact of Lightweight Peer Code Review. In *Pacific NW Software Quality Conference*, pages 1–10, 2009.
 - [96] E. S. Raymond. The Cathedral and the Bazaar. *Knowledge, Technology & Policy*, 12(3):23–49, 1999.
 - [97] P. C. Rigby and C. Bird. Convergent Contemporary Software Peer Review Practices. In *Proceedings of the 9th joint meeting of the European Software Engineering Conference and the International Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 202–212, 2013.
 - [98] P. C. Rigby, B. Cleary, F. Painchaud, M.-A. Storey, and D. M. German. Contemporary Peer Review in Action: Lessons from Open Source Development. *IEEE Software*, 29(6):56–61, 2012.
 - [99] P. C. Rigby, D. M. German, L. Cowen, and M.-a. Storey. Peer Review on Open-Source Software Projects: Parameters, Statistical Models, and Theory. *Transactions on Software Engineering and Methodology (TOSEM)*, 23(4):35:1–35:33, 2014.

- [100] P. C. Rigby, D. M. German, and M.-A. 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)*, pages 541–550, 2008.
- [101] P. C. Rigby and M.-A. Storey. Understanding Broadcast Based Peer Review on Open Source Software Projects. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE)*, pages 541–550, 2011.
- [102] R. Robbes and D. Röthlisberger. Using Developer Interaction Data to Compare Expertise Metrics. In *Proceedings of the 10th International Working Conference on Mining Software Repositories (MSR)*, pages 297–300, 2013.
- [103] J. Romano, J. D. Kromrey, J. Coraggio, and J. Skowronek. Appropriate Statistics for Ordinal Level Data: Should We Really Be Using T-Test and Cohen’s d for Evaluating Group Differences on the NSSE and Other Surveys? In *the annual meeting of the Florida Association of Institutional Research (FAIR)*, pages 1–33, 2006.
- [104] G. W. Russell. Experience with Inspection in Ultralarge-Scale Developments. *IEEE Software*, 8(1):25–31, 1991.
- [105] W. S. Sarle. The VARCLUS Procedure. In *SAS/STAT User’s Guide*. SAS Institute, Inc, 4th edition, 1990.
- [106] C. Sauer, D. R. Jeffery, L. Land, and P. Yetton. The Effectiveness of Software Development Technical Reviews: A Behaviorally Motivated Program of Research. *Transactions on Software Engineering (TSE)*, 26(1):1–14, 2000.
- [107] D. Schuler and T. Zimmermann. Mining Usage Expertise from Version Archives. In *Proceedings of the 5th International Working Conference on Mining Software Repositories (MSR)*, pages 121–124, 2008.

- [108] E. Shihab, Z. M. Jiang, and A. E. Hassan. Studying the Use of Developer IRC Meetings in Open Source Projects. In *Proceedings of the 25th International Conference on Software Maintenance (ICSM)*, pages 147–156, 2009.
- [109] E. Shihab, A. Mockus, Y. Kamei, B. Adams, and A. E. Hassan. High-Impact Defects: A Study of Breakage and Surprise Defects. In *Proceedings of the 8th joint meeting of the European Software Engineering Conference and the International Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 300–310, 2011.
- [110] R. Shokripour, J. Anvik, Z. M. Kasirun, and S. Zamani. Why So Complicated ? Simple Term Filtering and Weighting for Location-Based Bug Report Assignment Recommendation. In *Proceeding of the 10th Working Conference on Mining Software Repositories (MSR)*, pages 2–11, 2013.
- [111] F. Shull, V. Basili, B. Boehm, A. W. Brown, P. Costa, M. Lindvall, D. Port, I. Rus, R. Tesoriero, and M. Zelkowitz. What We Have Learned About Fighting Defects. In *Proceedings of the 8th International Software Metrics Symposium (METRICS)*, pages 249–258, 2002.
- [112] F. Shull and C. Seaman. Inspecting the History of Inspections: An Example of Evidence-Based Technology Diffusion. *IEEE Software*, 25(1):88–90, 2008.
- [113] D. Surian, N. Liu, D. Lo, H. Tong, E. P. Lim, and C. Faloutsos. Recommending People in Developers’ Collaboration Network. In *Proceedings of the 18th Working Conference on Reverse Engineering (WCRE)*, pages 379–388, 2011.
- [114] C. Tantithamthavorn, A. Ihara, and K. I. Matsumoto. Using Co-Change Histories to Improve Bug Localization Performance. In *Proceedings of the International Conference on Software Engineering, Artificial Intelligence,*

- Networking and Parallel/Distributed Computing (SNPD)*, pages 543–548, 2013.
- [115] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, A. Ihara, and K. Matsumoto. The Impact of Mislabelling on the Performance and Interpretation of Defect Prediction Models. In *Proceedings of the 37th International Conference on Software Engineering (ICSE)*, pages 812–823, 2015.
- [116] C. Tantithamthavorn, R. Teekavanich, A. Ihara, and K. I. Matsumoto. Mining A Change History to Quickly Identify Bug Locations: A Case Study of the Eclipse Project. In *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*, pages 108–113, 2013.
- [117] Y. Tao, Y. Dang, T. Xie, D. Zhang, and S. Kim. How Do Software Engineers Understand Code Changes?: An Exploratory Study in Industry. In *Proceedings of the 20th International Symposium on the Foundations of Software Engineering (FSE)*, pages 51:1–51:11, 2012.
- [118] Y. Tao, D. Han, and S. Kim. Writing Acceptable Patches: An Empirical Study of Open Source Project Patches. In *Proceedings of the 30th International Conference on Software Maintenance and Evolution (ICSME)*, pages 271–280, 2014.
- [119] Y. Tao and S. Kim. Partitioning Composite Code Changes to Facilitate Code Review. In *Proceedings of the 12th Working Conference on Mining Software Repositories (MSR)*, pages 180–190, 2015.
- [120] P. Thongtanunam, R. G. Kula, C. C. A. Erika, N. Yoshida, K. Ichikawa, and H. Iida. Mining History of Gamification Towards Finding Expertise in Question and Answering Communities: Experience and Practice with Stack Exchange. *The Review of Socionetwork Strategies*, 7(2):115–130, 2013.

- [121] P. Thongtanunam, S. McIntosh, A. E. Hassan, and H. Iida. Investigating Code Review Practices in Defective Files: An Empirical Study of the Qt System. In *Proceedings of the 12th International Working Conference on Mining Software Repositories (MSR)*, pages 168–179, 2015.
- [122] P. Thongtanunam, S. McIntosh, A. E. Hassan, and H. Iida. Review Participation in Modern Code Review. *Empirical Software Engineering (EMSE)*, page to appear, 2016.
- [123] P. Thongtanunam, S. McIntosh, A. E. Hassan, and H. 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)*, pages 1039–1050, 2016.
- [124] P. Thongtanunam, C. Tantithamthavorn, R. G. Kula, N. Yoshida, H. Iida, and K.-i. Matsumoto. Who Should Review My Code? A File Location-Based Code-Reviewer Recommendation Approach for Modern Code Review. In *Proceedings of the the 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SAnER)*, pages 141–150, 2015.
- [125] P. Thongtanunam, X. Yang, N. Yoshida, R. G. Kula, C. C. Ana Erika, K. Fujiwara, and H. Iida. ReDA: A Web-based Visualization Tool for Analyzing Modern Code Review Dataset. In *The proceeding of the 30th International Conference on Software Maintenance and Evolution (ICSME)*, pages 606–609, 2014.
- [126] Y. Tian, P. S. Kochhar, E.-p. Lim, F. Zhu, and D. Lo. Predicting Best Answerers for New Questions: An Approach Leveraging Topic Modeling and Collaborative Voting. *Social Informatics*, pages 55–68, 2014.

- [127] J. Tsay, L. Dabbish, and J. Herbsleb. Influence of social and technical factors for evaluating contribution in GitHub. In *Proceedings of the 36th International Conference on Software Engineering (ICSE)*, pages 356–366, New York, New York, USA, 2014. ACM Press.
- [128] J. Tsay, L. Dabbish, and J. Herbsleb. Let’s Talk About It: Evaluating Contributions through Discussion in GitHub. In *Proceedings of the 22nd International Symposium on Foundations of Software Engineering (FSE)*, pages 144–154, 2014.
- [129] E. Ukkonen. Algorithms for Approximate String Matching. *Information and Control*, 64(1-3):100–118, 1985.
- [130] L. G. Votta. Does Every Inspection Need a Meeting? In *Proceedings of the 1st Symposium on Foundations of Software Engineering (FSE)*, pages 107–114, 1993.
- [131] P. Weißgerber, D. Neu, and S. Diehl. Small Patches Get In! In *Proceedings of the 5th International Working Conference on Mining Software Repositories (MSR)*, pages 67–75, 2008.
- [132] X. Xia, D. Lo, X. Wang, and X. Yang. Who Should Review This Change? Putting Text and File Location Analyses Together for More Accurate Recommendations. In *Proceedings of the 31st International Conference on Software Maintenance and Evolution (ICSME)*, pages 261–270, 2015.
- [133] X. Xia, D. Lo, X. Wang, and B. Zhou. Accurate Developer Recommendation for Bug Resolution. In *Proceedings of the 20th Working Conference on Reverse Engineering (WCRE)*, pages 72–81, 2013.
- [134] Y. Yu, H. Wang, G. Yin, and T. Wang. Reviewer recommendation for pull-requests in GitHub: What can we learn from code review and bug assignment? *Information and Software Technology*, 74:204–218, 2015.

- [135] M. Zanjani, H. Kagdi, and C. Bird. Automatically Recommending Peer Reviewers in Modern Code Review. *Transactions on Software Engineering (TSE)*, page to appear, 2015.
- [136] T. Zimmermann, N. Nagappan, P. J. Guo, and B. Murphy. Characterizing and Predicting Which Bugs Get Reopened. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, pages 1074–1083, 2012.

An Example R Script for Non-linear Logistic Regression Models

A.1. Install and Load a Necessary R Library

```
1 install.packages("rms")
2 library(rms)
```

A.2. Model Construction

In this section, we provide R scripts for our model construction. For this example, we use the Android dataset that is used in Chapter 7 and publicly available online.¹

```
1 #Download the Android dataset
2 data <- read.csv('http://sailhome.cs.queensu.ca/replication/review_
  participation/data/Android.csv')
3
4 #Set dependent variable (for this example)
5 data$y = data$NumberOfReviewers == 0
6
```

¹http://sailhome.cs.queensu.ca/replication/review_participation/

```

7 ind_vars = c('Churn', 'NumberOfFiles', 'NumberOfDirectories', 'Entropy',
  'DescriptionLength', 'Purpose', 'DaysSinceTheLastModification',
  'NumberOfAuthors', 'NumberOfPriorDefects',
  'NumberOfReviewersOfPriorPatches', 'DiscussionLengthOfPriorPatches',
  'FeedbackDelayOfPriorPatches', 'PriorPatchesOfAnAuthor',
  'RecentPatchesOfAnAuthor', 'DirectoryPriorPatchesOfAnAuthor',
  'OverallWorkload', 'DirectoryWorkload')
8
9 #RMS package requires a data distribution when building a model
10 dd <- datadist(data[, c("y", ind_vars)])
11 options(datadist = "dd")

```

(MC-1.a) Correlation Analysis

```

1 #Calculate Spearman's correlation between independent variables
2 vc <- varclus(~ ., data=data[, ind_vars], trans="abs")
3 #Plot hierarchical clusters and the spearman's correlation threshold
  of 0.7
4 plot(vc)
5 threshold <- 0.7
6 abline(h=1-threshold, col = "red", lty = 2)

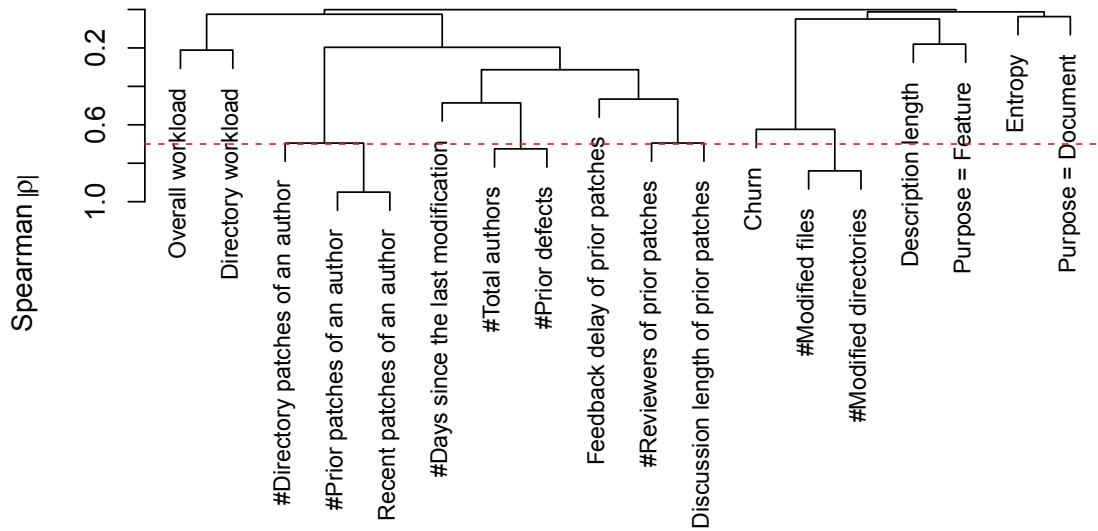
```

Listing A.1 Constructing hierarchical clusters of explanatory variables based on Spearman's correlation analysis.

```

1 #Remove the highly correlated variable from the hierarchical
  clusters
2 reject_vars <- c('DirectoryPriorPatchesOfAnAuthor',
  'RecentPatchesOfAnAuthor', 'NumberOfAuthors',
  'DiscussionLengthOfPriorPatches', 'NumberOfDirectories')
3 ind_vars <- ind_vars[!(ind_vars %in% reject_vars)]
4
5 #Re-calculate spearman's correlation between independent variables
6 vc <- varclus(~ ., data=data[, ind_vars], trans="abs")

```



```

7 #Re-plot hierarchical clusters and the spearman's correlation
  threshold of 0.7
8 plot(vc)
9 threshold <- 0.7
10 abline(h=1-threshold, col = "red", lty = 2)
11
12 #Churn and NumberOfFiles are still highly correlated. Therefore, we
  remove the NumberOfFiles variable out.
13 reject_vars <- c('NumberOfFiles')
14 ind_vars <- ind_vars[!(ind_vars %in% reject_vars)]

```

Listing A.2 Removing the highly correlated variables from and check for highly correlated variable again.

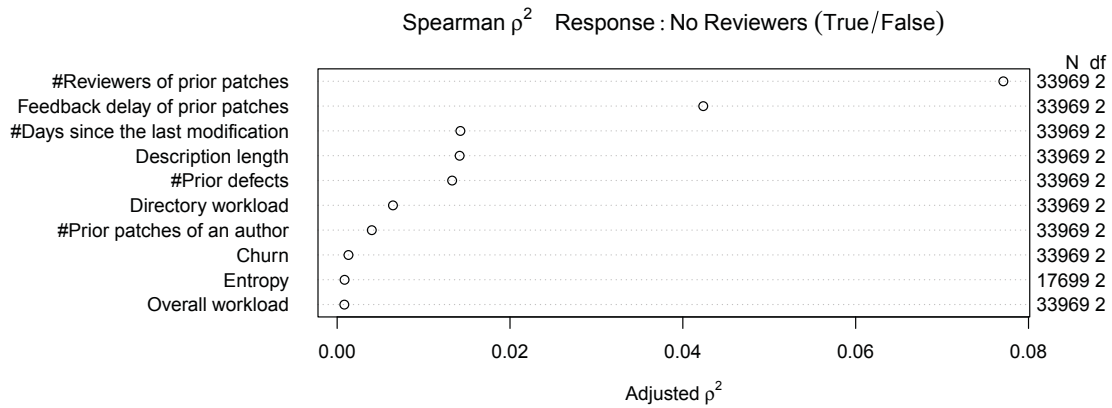
(MC-1.b) Redundancy Analysis

```

1 red <- redun(~., data=data[,ind_vars], nk=0)
2 print(red)
3 reject_vars <- red$Out
4 ind_vars <- ind_vars[!(ind_vars %in% reject_vars)]

```

Listing A.3 Removing the explanatory variables where an R^2 value greater than 0.9.



(MC-2) Degree of Freedom Allocation

```
1 sp <- spearman2(formula(paste("y", "~ ", paste0(ind_vars, collapse="
+ "))), data=data, p=2)
2 plot(sp)
```

Listing A.4 Estimating the potential of explanatory variables for sharing a nonlinear relationship with the response variable

(MC-3) Nonlinear Logistic Regression Model Construction

```
1 fit <- lrm(y ~ rcs(NumberOfReviewersOfPriorPatches,3) + rcs(
FeedbackDelayOfPriorPatches,3) + DaysSinceTheLastModification +
DescriptionLength + NumberOfPriorDefects + DirectoryWorkload +
PriorPatchesOfAnAuthor + Purpose + Churn + Entropy +
OverallWorkload, data=data, x=T, y=T)
```

Listing A.5 Fitting a nonlinear logistic regression model to the data

A.3. Model Analysis

In this section, we describe our model analysis approach.

(MA-1) Assessment of Explanatory Ability & Model Reliability

```

1 val <- validate(fit , B=1000)
2 AUC = 0.5 + val[1,1]/2
3 AUC_optimism_reduced = (0.5 + val[1,5]/2)
4 AUC_optimism = AUC - AUC_optimism_reduced
5 print(c("AUC"=AUC, "AUC_optimism"=AUC_optimism))

```

Listing A.6 Measuring AUC and AUC optimism

```

##           AUC  AUC_optimism
## 0.719464852 0.001431571

```

(MA-2) Power of Explanatory Variables Estimation

```

1 explanatory_power = anova(fit , test='Chisq')
2 print(explanatory_power)

```

Listing A.7 Estimate power of explanatory variables to the fit of the model

```

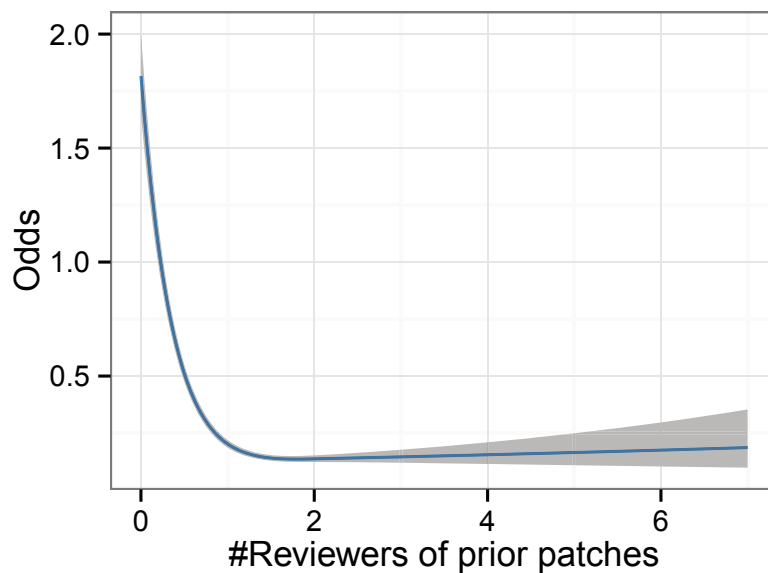
##           Wald Statistics           Response: y
##
## Factor           Chi-Square d.f. P
## NumberOfReviewersOfPriorPatches 1794.57      2    <.0001
##   Nonlinear                    572.33      1    <.0001
## FeedbackDelayOfPriorPatches      20.91      2    <.0001
##   Nonlinear                    20.66      1    <.0001
## DaysSinceTheLastModification    348.90      1    <.0001
## DescriptionLength                5.29      1    0.0215
## NumberOfPriorDefects             2.23      1    0.1352
## DirectoryWorkload                12.36      1    0.0004
## PriorPatchesOfAnAuthor           6.65      1    0.0099
## Purpose                        15.84      2    0.0004
## Churn                          0.25      1    0.6150
## Entropy                        0.04      1    0.8397

```

```
## OverallWorkload                6.61      1    0.0101
## TOTAL NONLINEAR                576.27     2    <.0001
## TOTAL                          2217.85    14    <.0001
```

(MA-3) Examination of Variables in Relation to the Response

```
1  predict <- Predict(fit, NumberOfReviewersOfPriorPatches, fun=
  function(x) exp(x))
2  plot(predict, ylab='Odds')
```



```
3  patial_effect = summary(fit)
4  print(patial_effect)
```

```
##           Effects           Response : y
##
## Factor           Low      High      Diff.      Effect
## NumberOfReviewersOfPriorPatches  0.00  1.00      1.00 -2.1960e+00
## Odds Ratio           0.00  1.00      1.00  1.1125e-01
## FeedbackDelayOfPriorPatches      0.00  3.19      3.19  8.2429e-02
```

##	Odds Ratio	0.00	3.19	3.19	1.0859e+00
##	DaysSinceTheLastModification	1.47	109.24	107.77	-6.9927e-01
##	Odds Ratio	1.47	109.24	107.77	4.9695e-01
##	DescriptionLength	11.00	45.00	34.00	-3.1415e-02
##	Odds Ratio	11.00	45.00	34.00	9.6907e-01
##	NumberOfPriorDefects	0.00	4.00	4.00	-7.1453e-03
##	Odds Ratio	0.00	4.00	4.00	9.9288e-01
##	DirectoryWorkload	1.00	10.00	9.00	3.2369e-02
##	Odds Ratio	1.00	10.00	9.00	1.0329e+00
##	PriorPatchesOfAnAuthor	3.00	105.00	102.00	-4.5098e-02
##	Odds Ratio	3.00	105.00	102.00	9.5590e-01
##	Churn	5.00	96.00	91.00	2.9869e-06
##	Odds Ratio	5.00	96.00	91.00	1.0000e+00
##	Entropy	0.61	0.91	0.30	4.8200e-03
##	Odds Ratio	0.61	0.91	0.30	1.0048e+00
##	OverallWorkload	221.00	477.00	256.00	8.4197e-02
##	Odds Ratio	221.00	477.00	256.00	1.0878e+00
##	Purpose - BUG-FIX:Feature	3.00	1.00	NA	-1.0940e-01
##	Odds Ratio	3.00	1.00	NA	8.9637e-01
##	Purpose - Document:Feature	3.00	2.00	NA	-4.5418e-01
##	Odds Ratio	3.00	2.00	NA	6.3497e-01

Additional Results for Chapter 4

B.1. A Sensitivity Analysis of RSO thresholds

We check whether the selection of RSO thresholds has an impact on our results. To do so, we compute the proportion of developers in each expertise category of the review-aware ownership heuristic (see Table 4.1) with the RSO threshold values of 2%, 5%, 7%, and 10%. Then, we measure the magnitude of the difference between the proportion of developers in each of the expertise categories of defective and clean modules using Cliff's δ [69]. Cliff's δ is considered as negligible for $|\delta| < 0.147$, small for $0.147 \leq |\delta| < 0.33$, medium for $0.33 \leq |\delta| < 0.474$ and large for $|\delta| \geq 0.474$ [103]. Figure B.1 shows the magnitude of the differences for each expertise category when using RSO_{Even} . A positive Cliff's δ value indicates that the proportion of developer in defective modules is larger than that in clean modules, whereas a negative Cliff's δ value indicates that the proportion of developer in defective modules is less than that in clean modules.

Figure B.1 shows that across the different RSO thresholds, the association between developer expertise and defect-proneness is similar, e.g., defective modules tend to have more developers in the minor author & minor reviewer category than clean modules do. However, we observe that when using a large RSO threshold value, the proportion of developers between defective and clean modules tend

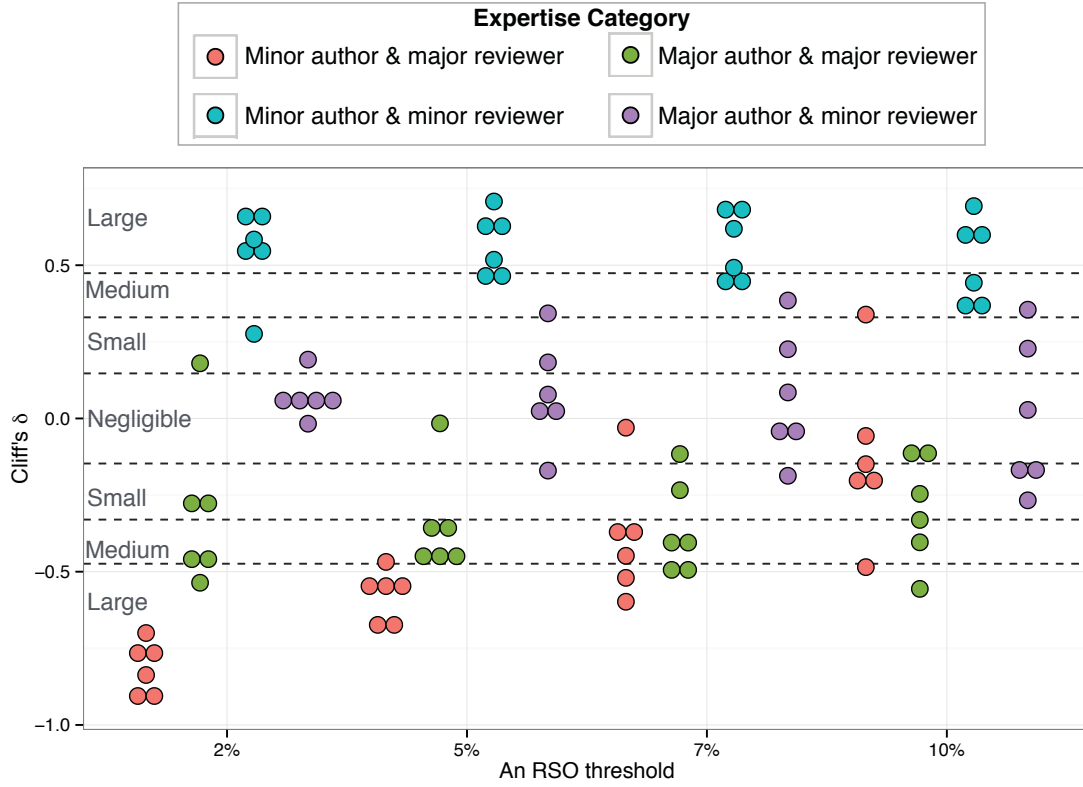


Figure B.1. A magnitude of difference between the proportion of developers in each of expertise categories of defective and clean modules when using RSO_{Even} . Each dot in each expertise category represents the result of one studied release.

to converge on a smaller difference. For example, Figure B.1 shows that when using an RSO threshold of 2%, a magnitude of differences in the minor author & minor reviewer category is large for five of the six studied releases. On the other hands, when using an RSO threshold of 10%, a magnitude of difference in the minor author & minor reviewer category is large for only three of the six studied releases. We observe similar results when using $\text{RSO}_{\text{Proportional}}$ (see Figure B.2).

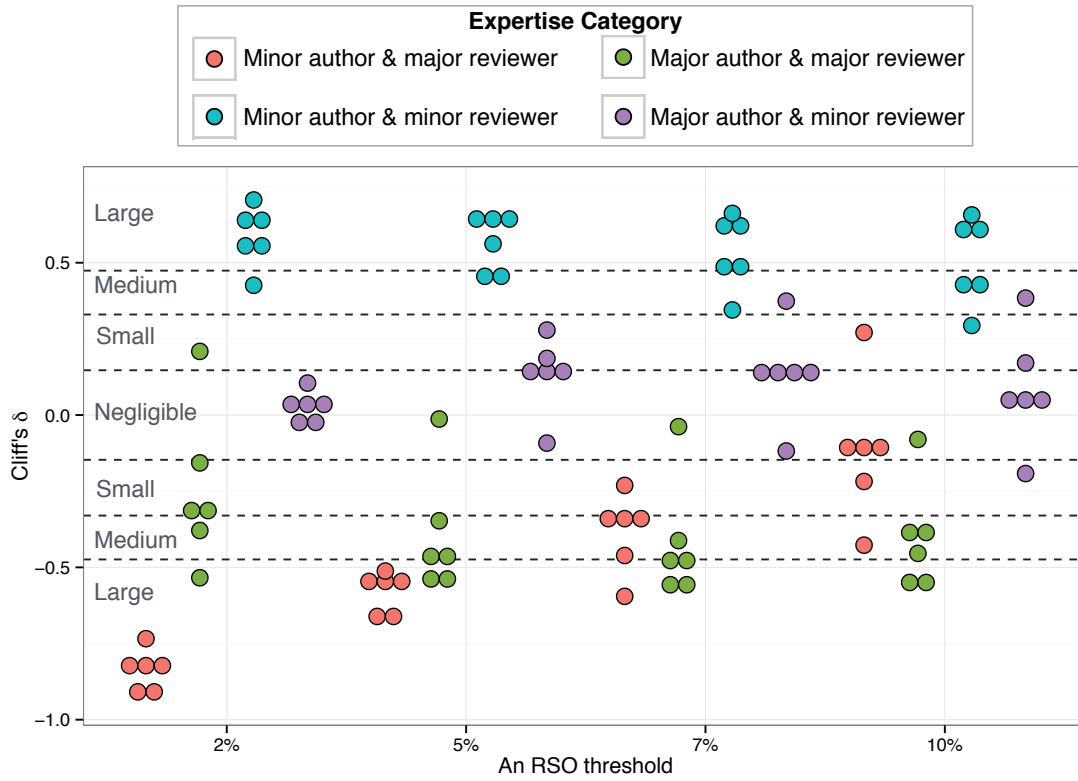


Figure B.2. A magnitude of difference between the proportion of developers in each of expertise categories of defective and clean modules when using $RSO_{Proportional}$. Each dot in each expertise category represents the result of one studied release.

B.2. Additional Model Statistics

Table B.1 shows the model analysis result when the review-specific and review-aware ownership are estimated using $RSO_{Proportional}$. Table B.2 shows the model analysis result when including the proportion of core developers. Table B.3 shows the model analysis result when we use the proportion of minor author & major reviewer instead of the proportion of minor author & minor reviewer to build our models. Figure B.3 shows a relationship between the proportion of developers in the minor author & major reviewer category and defect-proneness.

Table B.1. Statistics of defect models where review-specific and review-aware ownership are estimated using $RSO_{Proportional}$.

	Qt 5.0		Qt 5.1		OpenStack Folsom		OpenStack Grizzly		OpenStack Havana		OpenStack Icehouse	
AUC	0.81		0.87		0.88		0.83		0.87		0.84	
AUC optimism	0.03		0.02		0.02		0.02		0.01		0.02	
Wald χ^2	47***		85***		60***		69***		104***		114***	
	Overall	Nonlinear	Overall	Nonlinear	Overall	Nonlinear	Overall	Nonlinear	Overall	Nonlinear	Overall	Nonlinear
Size	D.F. χ^2	2 19%*	1 0%	2 5% 1%	1 24%***	1 13%***	1 12%*	1 0%	2 6%*	1 0%	2 7%*	1 1%
Churn	D.F. χ^2	1 4%	—	1 1%	1 0%	—	1 1%	—	1 13%***	—	1 11%***	—
Entropy	D.F. χ^2	1 1%	—	1 2%	1 4%	—	1 2%	—	1 0%	—	1 1%	—
Top TCO	D.F. χ^2	†		†	1 11%*	—	4 10%	3 2%	†		†	
Top $RSO_{Proportional}$	D.F. χ^2	1 0%	—	†	1 5%	—	2 9%*	1 2%	1 3%	—	1 1%	—
Major author & major reviewer	D.F. χ^2	1 6%	—	2 7%*	1 16%***	1 0%	2 1%	1 1%	2 0%	1 0%	2 7%*	1 2%
Minor author & minor reviewer	D.F. χ^2	1 44%***	—	4 50%***	2 2%	1 0%	2 5%	1 2%	4 22%***	3 5%	2 27%***	1 3%*
Major author & minor reviewer	D.F. χ^2	1 10%*	—	1 3%	1 10%*	—	1 1%	—	1 4%*	—	1 9%*	—

†: 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: $op \geq 0.05$; * $p < 0.05$; ** $p < 0.01$; *** $p < 0.001$

Table B.2. Statistics of defect models where the proportion of core developers is considered and review-specific and review-aware ownership are estimated using RSO_{Even} .

	Qt 5.0		Qt 5.1		OpenStack Folsom		OpenStack Grizzly		OpenStack Havana		OpenStack Icehouse	
	Overall	Nonlinear	Overall	Nonlinear	Overall	Nonlinear	Overall	Nonlinear	Overall	Nonlinear	Overall	Nonlinear
AUC	0.81		0.86		0.89		0.84		0.89		0.82	
AUC Optimism	0.05		0.03		0.03		0.02		0.01		0.03	
Wald χ^2	48***		81***		57***		70***		114***		96***	
Size	D.F. χ^2	2 12%*	1 0%	2 3%	1 1%	1 11%*	2 10%*	1 0%	2 4%	1 0%	2 27%***	1 1%
Churn	D.F. χ^2	1 4%	—	1 0%	1 0%	—	1 2%	—	1 16%***	—	1 12%***	—
Entropy	D.F. χ^2	1 1%	—	1 1%	1 5%	—	1 2%	—	1 0%	—	1 3%	—
Proportion of Core Developers	D.F. χ^2	1 0%	—	1 3%	1 0%	—	1 6%*	—	1 1%	—	1 5%*	—
Top TCO	D.F. χ^2	†		†	1 6%	—	4 11%	3 1%	2 0%	1 0%	†	
Top RSO_{Even}	D.F. χ^2	1 0%	—	1 5%*	1 1%	—	1 10%***	—	1 0%	—	1 5%*	—
Major author & major reviewer	D.F. χ^2	1 11%*	—	2 5%	2 14%*	1 6%	2 1%	1 1%	1 3%	—	1 3%	—
Minor author & minor reviewer	D.F. χ^2	2 46%***	1 4%	4 39%***	3 6%	1 5%	2 14%***	1 7%*	4 31%***	3 7%	2 14%***	1 1%
Major author & minor reviewer	D.F. χ^2	1 6%	—	2 1%	1 4%	—	1 2%	—	1 1%	—	1 3%	—

†: Discarded during variable clustering analysis ($|\rho| \geq 0.7$)

The number of contributors, authors, reviewers, and the proportion of minor author & major reviewer are also discarded during variable clustering analysis.

—: Nonlinear degrees of freedom not allocated.

Statistical significance of explanatory power according to Wald χ^2 likelihood ratio test: $op \geq 0.05$; * $p < 0.05$; ** $p < 0.01$; *** $p < 0.001$

Table B.3. Statistics of defect models where review-specific and review-aware ownership are estimated using RSO_{Even} and using the proportion of minor author & major reviewer instead of the proportion of minor author & minor reviewer.

	Qt 5.0		Qt 5.1		OpenStack Folsom		OpenStack Grizzly		OpenStack Havana		OpenStack Icehouse	
AUC		0.81		0.87		0.88		0.83		0.88		0.81
AUC Optimism		0.04		0.02		0.03		0.02		0.01		0.03
Wald χ^2		48***		90***		58***		67***		103***		92***
	Overall	Nonlinear	Overall	Nonlinear	Overall	Nonlinear	Overall	Nonlinear	Overall	Nonlinear	Overall	Nonlinear
Size	D.F. χ^2	2 17%*	1 5% $^\circ$	1 1% $^\circ$	2 27%***	1 13%**	2 16%**	1 0% $^\circ$	2 10%**	1 0% $^\circ$	2 29%***	1 1% $^\circ$
Churn	D.F. χ^2	1 3% $^\circ$	—	—	1 0% $^\circ$	—	1 1% $^\circ$	—	1 16%***	—	1 12%**	—
Entropy	D.F. χ^2	1 0% $^\circ$	—	—	1 4% $^\circ$	—	1 3% $^\circ$	—	1 1% $^\circ$	—	1 2% $^\circ$	—
Top TCO	D.F. χ^2	†	†	†	1 10%*	—	4 17%*	3 2% $^\circ$	2 2% $^\circ$	1 2% $^\circ$	†	†
Top RSO_{Even}	D.F. χ^2	1 0% $^\circ$	—	—	1 4% $^\circ$	—	1 6% $^\circ$	—	1 0% $^\circ$	—	1 4%*	—
Minor author & major reviewer	D.F. χ^2	2 48%***	1 3% $^\circ$	3 3% $^\circ$	2 1% $^\circ$	1 0% $^\circ$	2 6% $^\circ$	1 1% $^\circ$	4 28%***	3 1% $^\circ$	2 12%**	1 0% $^\circ$
Major author & major reviewer	D.F. χ^2	1 0% $^\circ$	—	1 1% $^\circ$	2 25%***	1 3% $^\circ$	2 2% $^\circ$	1 1% $^\circ$	1 0% $^\circ$	—	1 4%*	—
Major author & minor reviewer	D.F. χ^2	1 1% $^\circ$	—	1 6% $^\circ$	1 22%***	—	1 0% $^\circ$	—	1 8%***	—	1 0% $^\circ$	—

Discarded during: †Variable clustering analysis ($|\rho| \geq 0.7$)

Statistical significance of explanatory power according to Wald χ^2 likelihood ratio test: $op \geq 0.05$; * $p < 0.05$; ** $p < 0.01$; *** $p < 0.001$

— Nonlinear degrees of freedom not allocated.

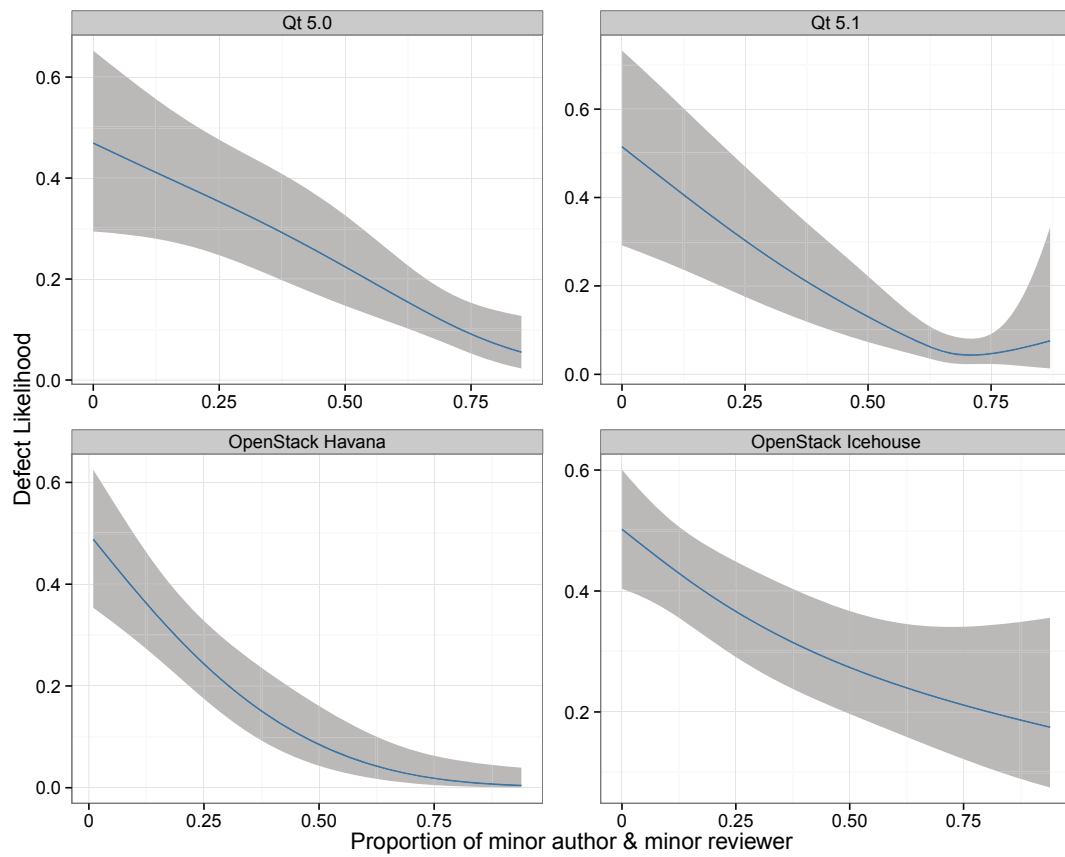


Figure B.3. The estimated probability in a typical module for the proportion of developers in the minor author & major reviewer category ranging. The gray area shows the 95% confidence interval.

Core Developers of the Studied Projects

C.1. The OpenStack Project

The list of core developers is from the OpenStack development documentation.¹

Table C.1. A list of core developers of the OpenStack project.

Name	Email Address
Aaron Rosen	aaronrosen@gmail.com
Akihiro Motoki	amotoki@gmail.com
Alex Meade	mr.alex.meade@gmail.com
Alexander Tivelkov	ativelkov@mirantis.com
Alistair Coles	alistair.coles@hp.com
Andrew Laski	andrew@lascii.com
Ann Kamyshnikova	akamyshnikova@mirantis.com
Armando Migliaccio	armamig@gmail.com
Arnaud Legendre	arnaudleg@gmail.com
Assaf Muller	amuller@redhat.com
Brandon Logan	brandon.logan@rackspace.com
Brian Haley	brian.haley@hp.com
Carl Baldwin	carl@ecbaldwin.net
Christian Schwede	cschwede@redhat.com

Continued on next page

¹https://wiki.openstack.org/wiki/Project_Resources (Last Access on June 2015)

Table C.1 A list of core developers of the OpenStack project. (*Cont.*)

Name	Email Address
Clay Gerrard	clay.gerrard@gmail.com
Dan Smith	dms@danplanet.com
Daniel Berrange	berrange@redhat.com
Darrell Bishop	darrell@swiftstack.com
David Goetz	david.goetz@rackspace.com
Doug Wiegley	dougwig@parkside.io
Duncan Thomas	duncan.thomas@gmail.com
Edgar Magana	emagana@gmail.com
Eric Harney	eharney@redhat.com
Erno Kuvaja	jokke@usr.fi
Fei Long Wang	flwang@catalyst.net.nz
Flavio Percoco	fpercoco@redhat.com
Greg Lange	greglange@gmail.com
Hemanth Makkapati	hemanth.makkapati@rackspace.com
Henry Gessau	gessau@cisco.com
Huang Zhiteng	winston.d@gmail.com
Ian Cordasco	ian.cordasco@rackspace.com
Ihar Hrachyshka	ihrachys@redhat.com
Ivan Kolodyazhny	e0ne@e0ne.info
Jay Bryant	jsbryant@us.ibm.com
Jay Pipes	jaypipes@gmail.com
Joe Gordon	joe.gordon0@gmail.com
John Dickinson	me@not.mn
John Griffith	john.griffith8@gmail.com
John Garbutt	john@johngarbutt.com
Ken'ichi Ohmichi	ken1ohmichi@gmail.com
Kevin Benton	kevinbenton@buttewifi.com
Kevin L. Mitchell	kevin.mitchell@rackspace.com
Kota Tsuyuzaki	tsuyuzaki.kota@lab.ntt.co.jp
Kyle Mestery	mestery@mestery.com
Louis Taylor	louis@kragniz.eu
Mark Washenberger	mark.washenberger@markwash.net
Matt Dietz	matt.dietz@rackspace.com
Matt Riedemann	mriedem@us.ibm.com
Matthew Oliver	matt@oliver.net.au
Michael Barton	mike@weirdlooking.com
Michael Still	mikal@stillhq.com

Continued on next page

Table C.1 A list of core developers of the OpenStack project. (*Cont.*)

Name	Email Address
Miguel Angel Ajo	mangelajo@redhat.com
Mike Fedosin	mfedosin@mirantis.com
Nikhil Komawar	nik.komawar@gmail.com
Nikola Dipanov	ndipanov@redhat.com
Oleg Bondarev	obondarev@mirantis.com
OpenStack Hudson	hudson@openstack.org
Pete Zaitcev	zaitcev@kotori.zaitcev.us
Rossella Sblendido	rsblendido@suse.com
Samuel Merritt	sam@swiftstack.com
Sean McGinnis	sean.mcginnis@gmail.com
Sean Dague	sean@daguer.net
Stuart McLaren	stuart.mclaren@hp.com
Thiago da Silva	thiago@redhat.com
Walter A. Boring IV (hemna)	walter.boring@hp.com
YAMAMOTO Takashi	yamamoto@midokura.com
Zhi Yan Liu	lzy.dev@gmail.com
Garyk	gkotton@vmware.com
Ichha-sethi	iccha.sethi@rackspace.com
Ike Perez	thingee@gmail.com
Mark McClain	mark@mcclain.xyz
Melanie Witt	melwitt@yahoo-inc.com
Paul Luse	paul.e.luse@intel.com
Xing-yang	xing.yang@emc.com

C.2. The Qt Project

The list of core developers is from the Qt development documentation.²

Table C.2. A list of core developers of the Qt project.

Name	Email Address
Aaron McCarthy	mccarthy.aaron@gmail.com
Alan Alpert	aalpert@blackberry.com
Alex Blasche	alexander.blasche@theqtcompany.com

Continued on next page

²<http://wiki.qt.io/Maintainers> (Last Access on June 2015)

Table C.2 A list of core developers of the Qt project. (*Cont.*)

Name	Email Address
Andrew Knight	andrew.knight@intopalo.com
André Pönitz	andre.poenitz@theqtcompany.com
Andy Nichols	andy.nichols@theqtcompany.com
Bernd Weimer	bernd.weimer@pelagicore.com and
Björn Breitmeyer	björn.breitmeyer@kdab.com
Bogdan Vatra	bogdan@kdab.com
Christian Strømme	christian.stromme@theqtcompany.com
David Faure	david.faure@kdab.com
David Faure	faure@kde.org
Denis Shienkov	denis.shienkov@gmail.com
Eirik Aavitsland	eirik.aavitsland@theqtcompany.com
Eskil Abrahamsen	eskil.abrahamsen- blomfeldt@theqtcompany.com
Frederik Gladhorn	frederik.gladhorn@theqtcompany.com
Friedemann Kleint	friedemann.kleint@theqtcompany.com
Giulio Camuffo	giulio.camuffo@jollamobile.com
Gunnar Sletta	gunnar@sletta.org
Jason McDonald	macadder1@gmail.com
Jens Bache Wiig	jens.bache-wiig@theqtcompany.com
Jens Bache-Wiig	jens.bache-wiig@theqtcompany.com
John Layt	jlayt@kde.org
Jørgen Lind	jorgen.lind@theqtcompany.com
Jędrzej Nowacki	jedrzej.nowacki@theqtcompany.com
Karsten Heimrich	karsten.heimrich@theqtcompany.com
Kurt Pattyn	pattyn.kurt@gmail.com
Lars Knoll	lars.knoll@theqtcompany.com
Laszlo Agocs	laszlo.agocs@theqtcompany.com
Lorn Potter	lorn.potter@gmail.com
Louai Al-Khanji	louai.al-khanji@theqtcompany.com
Mark Brand	mabrand@mabrand.nl
Martin Smith	martin.smith@theqtcompany.com
Milian Wolff	milian.wolff@kdab.com
Morten Sørvig	morten.sorvig@theqtcompany.com
Olivier Goffart	ogoffart@woboq.com
Oswald Buddenhagen	oswald.buddenhagen@theqtcompany.com
Pasi Keranen	pasi.keranen@theqtcompany.com
Rafael Roquette	rafael.roquette@kdab.com

Continued on next page

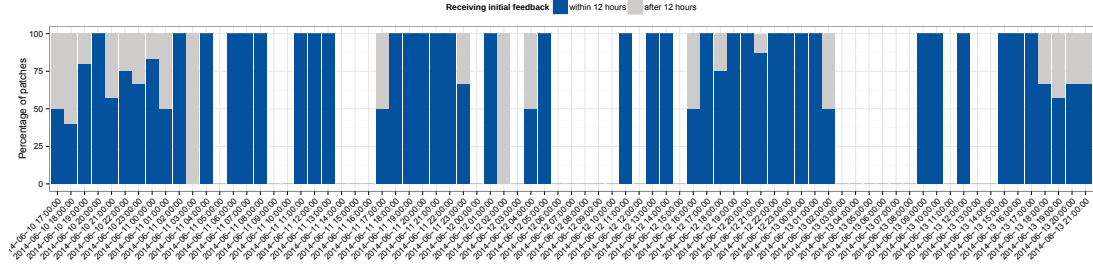
Table C.2 A list of core developers of the Qt project. (*Cont.*)

Name	Email Address
Sean Harmer	sean.harmer@kdab.com
Stephen Kelly	stephen.kelly@kdab.com
Thiago Macieira	thiago.macieira@intel.com
Tor Arne Vestbø	tor.arne.vestbo@theqtcompany.com

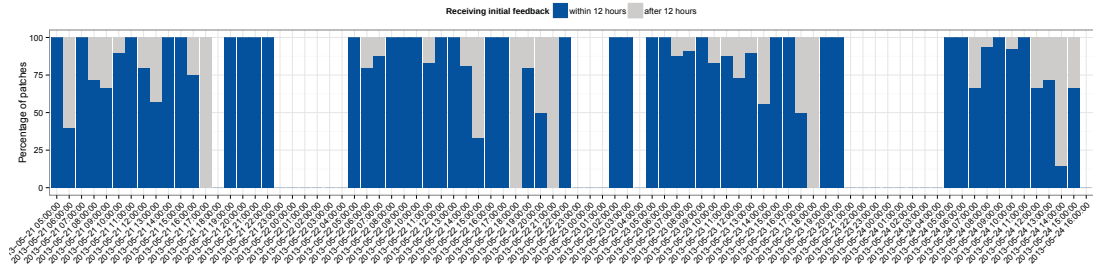
Additional Analysis for Chapter 7

D.1. An Analysis of Threshold Sensitivity

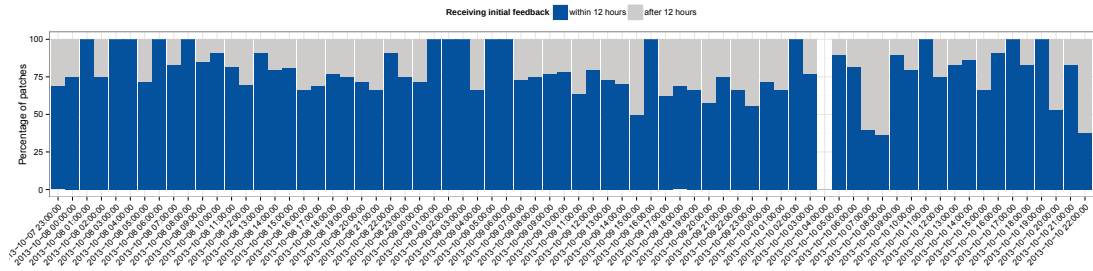
In RQ3 of Chapter 7, we study the characteristics of patches that will receive slow initial feedback. We use a threshold of 12 hour to identify patches that will receive slow initial feedback. However, there is a likely case that this threshold is impacted by timezone difference since our studied systems have globally-distributed development teams. Hence, we count the number of patches that receive slow and prompt initial feedback. Figure D.1 shows the hourly percentage of patches that receive slow and prompt initial feedback, where the x-axis shows submission dates of the patches. Figure D.1(a) and D.1(b) show that in Android and Qt, the patch submission periodically occurs, suggesting that development teams of Android and Qt tend to locate in the similar timezones. Hence, our threshold should not be impacted by timezone difference for Android and Qt. On the other hand, Figure D.1(c) shows that the patch submission occurs almost every hour, suggesting that development teams of OpenStack tend to distribute over different timezones. However, we do not observe a specific pattern of the percentage of patches that receive slow initial feedback over the time. Therefore, we believe that the timezone difference may not have a large impact on our threshold for OpenStack.



(a) Android



(b) Qt



(c) OpenStack

Figure D.1. The percentages of patches that receive prompt (blue) and slow (gray) initial feedback. The x-axis shows the patch submission dates.