INVESTIGATING PAST AND PRESENT CODE REVIEWER RECOMMENDATION SYSTEMS

PALAK HALVADIA
Bachelor of Computer Engineering, Gujarat Technological University, 2017

A thesis submitted
in partial fulfilment of the requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

Department of Mathematics and Computer Science
University of Lethbridge
LETHBRIDGE, ALBERTA, CANADA

© Palak Halvadia, 2021

INVESTIGATING PAST AND PRESENT CODE REVIEWER RECOMMENDATION SYSTEMS

PALAK HALVADIA

Date of Defence: April 14, 2021

Dr. J. Anvik Thesis Supervisor	Associate Professor	Ph.D
Dr. W. Osborn Thesis Examination Committee Member	Associate Professor	Ph.D
Dr. Y. Chali Thesis Examination Committee Member	Professor	Ph.D
Dr. J. Sheriff Chair, Thesis Examination Committee	Assistant Professor	Ph.D

Abstract

Context: Selecting a code reviewer is an important aspect of software development and depends on several factors.

Objectives: The aim is to understand existing solutions for code reviewer recommendation systems (CRRSs), factors to be considered when building them and various dimensions based on which they can be categorised. Our goal is to understand important features of CRRSs and what can be improved in existing CRRSs.

Methods: A literature review study was conducted to understand the existing CRRSs. A survey of software development project members was conducted to understand the important and missing features in CRRSs.

Results: We categorized the selected papers into two categories: based on the data type used to make recommendations and the kind of project used for evaluation. The survey helped us understand the features missing in CRRSs and observe some trends and patterns.

Acknowledgments

First and foremost, I would like to thank my parents for their continuous love and support throughout my life and especially during this journey of my masters. Thank you for encouraging me to pursue my dreams and giving me strength to pursue my dreams. I would also like to thank my brother for giving all the moral support.

I would like to sincerely thank my supervisor Dr. John Anvik for his guidance and support throughout this study and having the confidence in me. Thank you for giving me the opportunity to explore the fields I wanted to study during my thesis. I would also like to thank the Sibyl Lab for all their support throughout my graduate journey.

I would also like to thank all my friends for your understanding, encouragement and support during my moments of crisis. The friendship we have shared has made my journey really amazing and though I cannot list all the names here, I would always be thankful that you all were there for me in times of need. The friendship I share with you all has taught me a lot and made me a better person than I was.

Contents

Contents v

List of Tables vii

List of Figures viii

1 Introduction 1

1.1 Overview of work 4

1.2 Why this work is needed 4

1.3 Contributions 6

2 Related Work 7

2.1 Recommender Systems Literature Reviews 8

2.2 Literature Reviews in Software Engineering 10

2.3 Literature Reviews in Data Mining 12

2.4 Summary 12

3 Code Review Practices and Tools 14

3.1 Methodology 15

3.2 Results 16

3.3 Systems Found 18

3.3.1 REVFINDER 18

3.3.2 cHRev 20

3.3.3 CoRReCT 22

3.3.4 TIE 23
3.3.5 CodeFlow 26
3.3.6 CRITIQUE 30
3.3.7 Profile-based CRRS 35
3.3.8 Categorization of systems 36
3.3.8.1 Project Used for Evaluation 37
3.4 Summary 41

4 Information Needs of Code Reviewers 42
4.1 Methodology 42
4.1.1 Screening Survey 42
4.1.2 Demographic and Code Reviewer Recommendation Systems/Tools Usage Survey Questions 44
4.2 Results 50
4.3 Some observed trends and patterns 62
4.4 Summary 67

5 Discussion 68
5.1 Proposal for an Improved Code Review Recommender System 72

6 Conclusion 75
6.1 Contributions: 75
6.2 Future Work 76

7 Appendix 78
7.1 Ethical Review Certificate 78

List of Tables

3.1 Search Strings 16

3.2 Extracted Research Papers and Data Used. 17

3.3 Data Sources for Code Review Recommendation Systems 37

3.4 Project used for evaluation 38

4.1 Job roles of the participants 51

4.2 Geographic location of the participants 52

4.3 Size of the project team 52

4.4 Distribution of the team 53

4.5 Familiarity of the CRRS amongst the participants 54

List of Figures

1.1 Research Questions and Methodology 5

3.1 A calculation example of the Code-Reviewers Ranking Algorithm[12] . . . 19

3.2 Architecture of TIE [13] 24

3.3 Three stage research method[11] 27

3.4 Decision tree model to classify useful comments[11] 29

3.5 Relationship diagram that describes the themes of review expectations appearing primarily in a particular author/review context[22] 33

4.1 Reported usefulness of CRRS features 55

4.2 Criteria for selecting a code reviewer 56

4.3 UI features of CRRS 60

4.4 Preference of when to have the code review recommendation 61

4.5 Kind of code reviews 62

4.6 Type of CRRS and job role 63

4.7 CRRS features and job role 65

Chapter 1

Introduction

Code review is a systematic examination of computer source code and is most often done as a peer review. Code review aims at identifying and rectifying the mistakes in the source code as well as improving the quality of code and a software developer's skills. Also, it does not only aim at code quality improvement or finding defects in the source code; it also increases the team awareness as well as help in knowledge distribution. It also encourages the shared code ownership. There are four types of code reviews:

1. **Pair programming:** In this type of code review two developers produce source code simultaneously, as well as reviewing at the same time.

2. **Tool-assisted code review:** For this kind of code review the authors and developers use tools for peer code reviews.

3. **Walk-through code review:** Here, the developer walks the reviewer through a set of code changes.

4. **Formal code review:** This kind of code review involves a careful and

detailed inspection of code with the involvement of multiple number of participants and in multiple phases. It is a traditional method of code review which involves attending a number of meetings and reviewing the code line-by-line.

Code review can be considered as a manual inspection of the changes in source code [21]. There are a number of tools and recommendation systems developed for the purpose of code review by a number of different organizations [22].

There are number of fields where the contribution of recommendation systems has proved to be useful to software development project members. To assist with the task of code review, a significant amount of research has been conducted on recommendation systems which aim at providing recommendations of code reviewers based on various aspects. There are various reasons as to why a code reviewer is needed in addition to finding code defects. This is because code reviewers also focus on code improvement, finding alternative solutions to a problem, knowledge transfer, improvisation in the development process, avoiding build breaks [1], sharing of ownership of code, as well as team assessment [9]. For example Rahman, Roy, and Collins [16], proposed a code review system in which the expertise of a code reviewer is based on the information obtained from a cross-project work

[1]When a developer adds changes to the source code repository that result in the failure of a subsequent build process, the developer has "broken the build."

history, as well as the specialization of a code reviewer in a particular field based on their pull requests.

Examples of more general recommendation systems in software engineering include graphical code recommender systems [15] and the `Hipikat` system [2]. Lee and Kang [15] conducted a study on ‘graphical code recommender systems’ to understand to what extent software visualization tools have helped developers to comprehend the code. The authors learned that developers spend a significant amount of time on understanding the code bases. To make this easier, a number of graphical code recommenders were created for them. These recommender systems used two abstractions: the designing and explanation of code and the documentation of the software system, as well as its analysis.

A recommender system was developed called `Hipikat` which provides the developers with access to a group memory that contains project-related artifacts created during the development of the project [2]. This helps the developers to save time in overcoming the technical and sociological difficulties. This tool creates the group memory automatically with very few or no changes to the existing work practices. This recommendation system helped in sharing the information related to a project from each and every perspective to all of the members of the development team, thereby saving time on the explanation of concepts to the existing and new members of the

development team.

1.1 Overview of work

1. Types of code review. This work will focus on tool-supported code review.

Our research aim is two-fold-first, to find the answers to the 'past' questions by conducting a 'systematic literature review' and second to find answers to the 'present' questions by conducting a survey of software project members. A 'Systematic Literature Review' will help in finding the details about the existing code reviewers recommendation systems whereas the survey will help in finding what changes the software engineers think are needed or what is lacking in the existing code reviewer recommendation systems.

1.2 Why this work is needed

The aim of this research is to document the learnings obtained from the 'systematic literature review study' as well as the 'survey' conducted on a broad range of software project members. This work was done as there has been very less research done on Code Reviewer Recommendation Systems (CRRS) and more research done on code review practices and procedures. The objectives that will help us in finding answers to the required outcome are mentioned below:

- To analyze the solutions provided by currently existing 'code reviewer

recommendation systems'.

- To understand the solutions/features missing in the existing 'code reviewer recommendation systems'.
- To analyze the different means of bifurcation of existing 'code reviewer recommendation systems' based on their way of implementation.

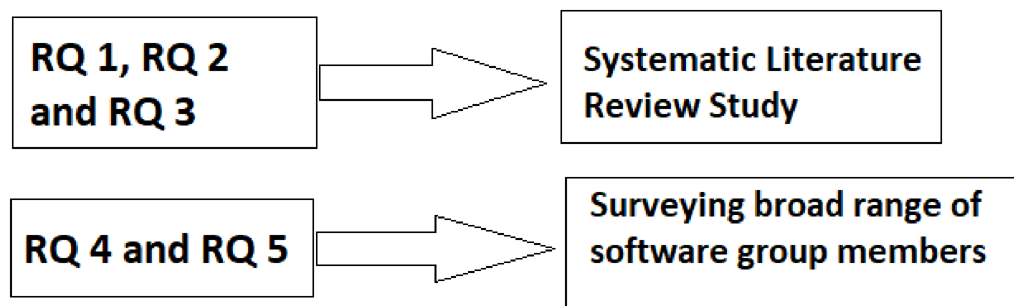

Figure 1.1: Research Questions and Methodology

The research questions for this work are as follows:

RQ1. What are the existing solutions for recommendation systems for code reviewers?

RQ2. What are the factors that need to be taken into consideration when creating a recommendation system for code reviewers?

RQ3. How can existing recommendation systems for code reviewers in the literature be categorized?

RQ4. What are the important features for a recommendation system for code reviewers?

RQ5. How can existing recommendation systems for code reviewers be improved? In other words, what features are missing from existing implementations for recommendation systems of code reviewers?

RQ1, RQ2 and RQ3 will be answered using the systematic literature review whereas RQ4 and RQ5 will be answered by surveying broad range of software project members.

1.3 Contributions

This research makes the following contributions:

C1: We identified a number of features present in the existing Code Review Recommendation Systems (CRRS) and ranked those features based on their usefulness.

C2: We categorized the existing CRRSs based on different dimensions.

C3: We identified the features that can be considered important when selecting a code reviewer.

C4: We identified possible improvements to existing CRRSs to facilitate the finding of appropriate code reviewers.

Chapter 2 presents the related work. The results of the literature review study is presented in Chapter 3. Chapter 4 is the survey of results. Chapter 5 consists of discussion. The thesis has been concluded in Chapter 6.

Chapter 2

Related Work

This chapter presents summaries of previous literature reviews done in software engineering, data mining, and recommender systems. These studies are presented to show how literature review studies have been conducted in the past and used to guide the methodology for our literature review.

The literature review conducted on recommender systems includes recommender systems which aim at extracting relevant information from a huge amount of knowledge and recommendation systems for software engineering that presents the features in the existing systems, research gaps and possible future work. Similarly, a literature review study was done in the field of software engineering regarding fault prediction studies and the agile software development methodology. The literature review done in data mining discovered the two most used models for data mining in CRM (Customer Relationship Management).

2.1 Recommender Systems Literature Reviews

A literature review was conducted by Haruna, Ismail, Suhendroyono, *et al.* [18] on Context-Aware Recommender Systems (CARS) that aim at extracting the relevant information required from a huge amount of knowledge. These kind of recommender systems aim at giving contextual and relevant information based on the 'user searches' and providing more personalized user recommendations. Haruna, Ismail, Suhendroyono, *et al.*'s [18] approach consists of three main steps. The first step includes the in-depth review and classification of literature based on various domains of the application models, filtering, extraction, as well as evaluation approaches. The second step involves presenting the results of the review with the advantages and disadvantages of review. The third and final step involves highlighting the possibles challenges/opportunities or the future work or research that can be done. This include helping the novice and new researchers understand the prerequisites for the development of CARS as well as provide this review as a benchmark to develop CARS for expert users.

A recommendation system is a type of software application which aims at providing/recommending relevant information to the users based on the user requirements. For this area, a systematic literature review study was conducted similar to Gasparic and Janes [14], which examines the results of the functionality that existing RSSEs (Recommendation System for Software

Engineering) provide, the research gaps, as well as the possible research directions. They followed a methodological approach that included filtering out the gathered, relevant research papers based on various criteria. Their exclusion criteria included papers that were irrelevant to the research field, papers that described unimplemented solutions or papers that were not fully accessible. For extracting relevant papers, the papers were filtered and divided based on the content described in the abstract of the paper or sometimes the title. After following their methodological approach, the authors obtained answers to their four research questions which include: the output given by the existing RSSEs, the benefits that these RSSEs provide to the software engineers, the types of input that these RSSEs require and what efforts does a software engineer require to put in for using these RSSEs. It was seen that some of the outputs given by the existing RSSEs include binary source code files, changes in the deployment environment, design patterns and digital documents that might be interesting for the software engineer. The existing RSSEs mainly support reuse, debugging, implementation, maintenance phases/activities and support the improvement of system quality to benefit the software engineers. Some of the inputs that these existing RSSEs use include a log files, communication between software engineers, source code, user input (e.g., search terms, a query, settings, preferences), test artifacts and software development process. Also, the efforts that a software engineer requires to put in for using the existing RSSEs are categorized as extensive

efforts, low efforts and no efforts.

Another literature review was conducted for RSSEs by Park, Kim, Choi, *et al.*[8] where the authors categorized the research papers based on eight application fields and eight data mining techniques. The aim of the authors was to provide information regarding the trends in the field of research for recommender systems, as well as defining the possible future research direction on recommender systems.

2.2 Literature Reviews in Software Engineering

Accurately predicting faults in code can reduce the cost of testing, to a great extent, as well as increase the quality of the software product. For this purpose, a literature review study was done by Hall, Beecham, Bowes, *et al.* [7] which concentrated on fault prediction studies. The authors followed the systematic literature review approach as proposed by Kitchenham and Charters [4] where the initial steps consist of including the relevant papers and studies and excluding the repeated studies. Various aspects are taken into consideration when excluding and including the papers, such as the papers were extracted from various resources, such as journals, conferences and databases, and sorted based on the content of their title and abstract. The authors ended up with around 208 papers. The findings show that most of the studies report insufficient contextual and methodological information to enable full understanding of a model. The authors present a set of criteria

that identify the set of essential contextual and methodological details that fault prediction studies should report.

The Agile software development methodology is a common software development methodology used by many software development projects. The methodology aims to assure a good delivery of product as per the user's requirements and a suitable User Experience (UX). In order to deliver a quality product, the involvement of stakeholders and users is necessary, along with feedback loops from both sides. A literature review study was conducted on the Agile methodology by Schön, Thomaschewski, and Escalona [19] in order to capture the current state of work in this field and possible future enhancements to address any aspects that are lacking in the current state. They conducted the study in three main phases: planning, conducting and reporting. The 'planning' phase included finding the identification need for a review, framing the research questions and developing and evaluating the review protocol. The 'conducting' phase aimed at searching for the research papers, selecting the relevant papers for study, qualitative assessment and data extraction and analysis. The last phase, 'reporting' aimed at extracting and discussing the results obtained from the previous phase and then writing, evaluating and formatting the final report for the study. Similar to other studies, the authors followed the methodology posed by Kitchenham and Charters [4].

2.3 Literature Reviews in Data Mining

The literature review conducted by Ngai, Xiu, and Chau [6] provides another example of a methodology for a systematic literature review in the area of data mining.

Data mining techniques are applied to Customer Relationship Management (CRM) and Ngai, Xiu, and Chau [6] provides a thorough insight on this with the help of a literature review they conducted. The authors gathered about 87 relevant research papers for this purpose which were split based on four CRM dimensions which include Customer Development, Customer Identification, Customer Attraction and Customer Retention and seven data mining techniques; Association, Classification, Clustering, Forecasting, Regression, Sequence Discovery and Visualization. Apart from this, for more clarity, the CRM dimensions were further classified into nine sub-categories of CRM elements falling under the data mining techniques. Based on the study, it was found that Classification and Association were the two most used models for data mining in CRM. Also, from the four CRM dimensions, Customer Retention is the most researched one, though most of them were related to one-to-one marketing and loyalty programs.

2.4 Summary

Literature review studies were presented for recommender systems, and from the fields of software engineering and data mining. For recommenda-

tion systems, the literature reviews focused on context-aware recommender systems and recommendation systems in software engineering. In the field of software engineering, the presented study was conducted on the area of fault prediction, as well as on the Agile methodology. In the field of data mining, a literature review study was conducted where the concept of data mining applied to Customer Relationship Management (CRM) was targeted.

Chapter 3

Code Review Practices and Tools

There have been a number of code reviewer recommendation systems/tools developed in the past. For example, a tool named *Review Bot* is one such other tool developed by Balachandran [10] which was used in the VMware project. Review Bot consisted of an algorithm that examines the code changes done in a line of code in a way that is quite similar to the *git blame* command. Each and every author who has worked on the code change in the source code is awarded points but authors with more recent changes gain more points than the authors with older changes. At the end, the summation of each individual author is used to decide the top-*k* authors and then recommend them to become code reviewers.

We conducted a systematic literature review study in order to find answers to our first three research questions. First, we define our methodology before presenting the results of our study.

3.1 Methodology

A Systematic Literature Review Study is a methodology where the available literature that is related to the research is determined, then assessed and finally comprehended. For our research, we have followed the approach adopted by Kitchenham and Charters [4] which consists of the following steps:

1. All of the possible keywords related to the research are identified. We identified the following words: code, reviewer, recommendation, systems, tools and recommender. These keywords were identified based on our research topic.

2. Use the identified keywords to form search strings. The search strings are used to obtain research papers from online databases. We used a search string of all of the possible keywords and their synonyms. We created two search strings. See Table 3.1 for the search strings.

3. The obtained research papers are then filtered based on various exclusion and inclusion criteria. The papers are first filtered by reading the titles and the abstract of the research papers.

   The filtering of the search results was carried out in three main steps:

   (a) Filtering out the research papers based on reading the title of the research paper,

(b) Filtering out research papers based on the abstract of the research paper and

(c) Filtering out the research papers based on full-text reading

4. Filtered papers are fully read, assessed and interpreted to obtain relevant information.

Table 3.1: Search Strings

1.	(Code) AND (Reviewers) AND ((Recommender) OR (Recommendation)) AND ((Systems) OR (Tools))
2.	((Recommendation) OR (Recommender)) AND ((Systems) OR (Tools)) AND (Code) AND ((Reviews) OR (Reviewers))

At each step of filtration, a number of research papers were filtered out. After reading the paper titles 19 papers were filtered out. From these obtained papers, 21 papers were filtered out after reading the abstract of the papers. Lastly, 7 papers were filtered out after reading the full-text of the research papers. In the end 14 papers were obtained.

3.2 Results

After reading the full-text of the filtered research papers, nine code review recommender systems were identified. Table 3.2 lists these systems and

the data used by the system for making a recommendation. The rest of this section presents descriptions of each of these systems.

Table 3.2: Extracted Research Papers and Data Used.

Data Type	**Research Paper Title**
Code Review Histories	Automatically Recommending Peer Reviewers in Modern Code Review [17]
Commits	Modern Code Review: A Case Study at Google [22]
Tracks state of each reviewer or author	Characteristics of Useful Code Reviews: An Empirical Study at Microsoft [11]
File-Path Similarity (FPS)	A large-scale study on source code reviewer recommendation [12]
Relevant cross-project and Technology Experience	CoRReCT: code reviewer recommendation in GitHub based on cross-project and technology experience[16]
Text Mining and file location	Who should review this change?[13]
Profile based	Profile based recommendation of code reviewers [20]

3.3 Systems Found

3.3.1 REVFINDER

There have been a number of CRRSs proposed based on the File Path Systems (FPS) or File Location-Based approach. Thongtanunam, Tantithamthavorn, Kula, *et al.*[12] proposed *REVFINDER* which follows the approach of file location-based code-reviewer recommendation. The intuition behind this approach is that multiple files with a similar location/file path would be reviewed and managed by similar experienced code-reviewers.

Thongtanunam, Tantithamthavorn, Kula, *et al.*[12] also conducted an exploratory study about how the code-reviewer assignment impacts reviewing time. The exploratory study showed that about 4%-30% of code reviews face the problem of determining the correct code reviewer and it takes around 12 days longer to approve a code change. Based on the results of this study, the authors proposed *REVFINDER*.

REVFINDER consists of two parts: the Code Reviewers Ranking Algorithm and the Combination Technique. The authors used the Code Reviewers Ranking Algorithm (as shown in the Figure 3.1 [12]) to evaluate the scores of the code reviewers based on the similarity of files paths previously reviewed. Given a new review *R3* and two past reviews *R1* and *R2*, the algorithm calculates the review similarity score for each of the past reviews *(R1,R2)* by comparing the file paths with the new review *R3*. Hence, there were two review similarity scores of size two: *(R3, R1)* and *(R3, R2)*. From the figure

it can be seen that review *R3* and *R2* share more common keywords as compared to *R3* and *R1* which means that Reviewer A can be considered as the potential reviewer for review *R3*. In order to compute the file path similarity the authors used four state-of-the-art [1] string comparison techniques:

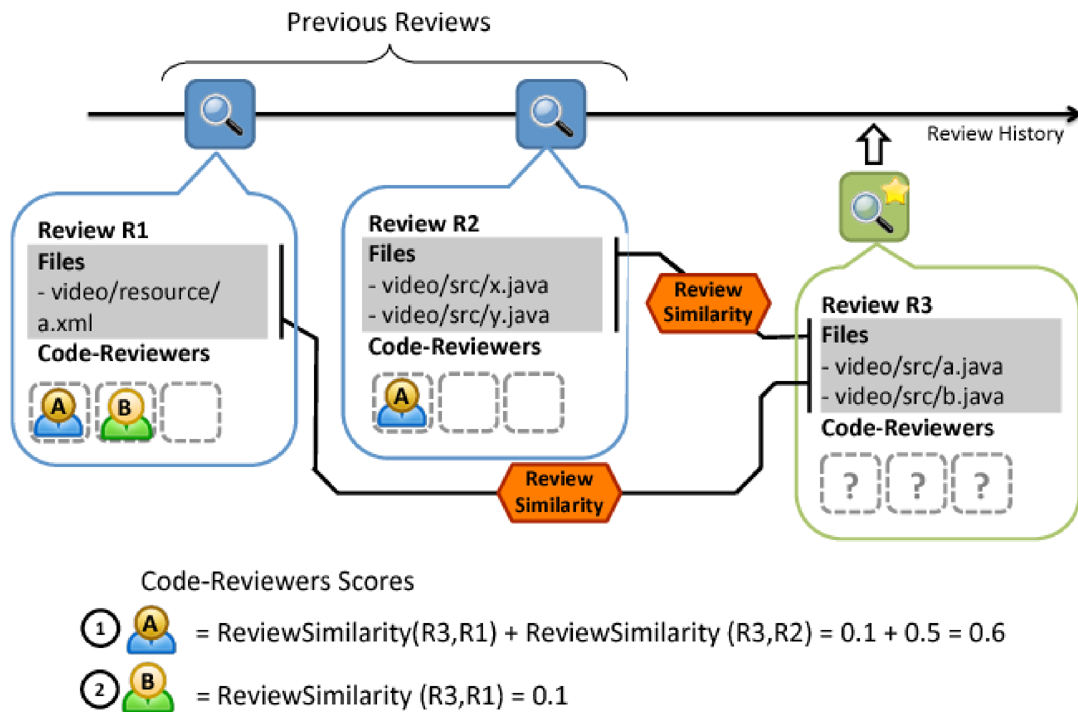

Figure 3.1: A calculation example of the Code-Reviewers Ranking Algorithm[12]

1. **Longest Common Prefix (LCP)**

   LCP calculates the common file path components that appear in both of file paths from start to end.

2. **Longest Common Suffix (LCS)**

   LCS calculates the common file path components that appear in both of file paths from the end of both of the file paths.

3. **Longest Common Substring (LCSubstr)**

   LCSubstr calculates the common file path components that appear in both of the file paths consecutively but also appear at any position in the file paths.

4. **Longest Common Subsequence (LCSubseq)**

   LCSubseq calculates the common file path components that appear in both of the files paths in the same relative order.

Now, In order to calculate the file path similarity between file f_n and file f_p, the filePathSimilarity(f_n, f_p) function is calculated as follows:

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

The file path is split into tokens using the slash character (”/”) as a delimiter. The StringComparison(f_n, f_p) function is then used to compare the file path components of f_n and f_p which returns the common file components that appear in both file paths.

3.3.2 cHRev

A number of CRRS have been built based on past reviews and Zanjani, Kagdi, and Bird [17] built one such recommendation system called cHRev. cHRev automatically recommends code reviewers based on their past contributions made in their prior reviews. cHRev stands for **c**ode review **H**istories

over other types of past information to recommend **Re**viewers. This recommendation system has two key features:

1. The code reviewers recommended by cHRev may not be necessarily involved in developing the part of source code that they are reviewing but might have worked on source code that is indirectly dependent on the source code they are reviewing.

2. The *expertise* changes over time and hence *recency* and *frequency* must be accounted for when searching for the most appropriate code reviewer.

The process used by cHRev consists of three steps:

1. Extract the source code that needs to be reviewed.

2. Formulate reviewer expertise based on various details such as who, how many and when were reviews performed in the past.

3. Obtain a ranked list of candidate reviewers based on the source code files in step 1 and the cumulative contributions of reviewers from step 2 and then recommend the top *m* number of candidates from the obtained list using a user-defined parameter.

In order to test the effectiveness of their approach, Zanjani, Kagdi, and Bird [17] compared their approach with REVFINDER[12], *x*Finder [5] and *RevCom*. It was found that *cHRev* makes more accurate reviewer recommendations in terms of *precision* and *recall*. Also, it was seen that *cHRev*

performed better than *REVFINDER*, in terms of reviewers based on the files having similar names and paths and *xFinder*, which depends on the source code repository data. cHRev was found to be statistically equivalent to *RevCom* [17], which requires both past reviews and commits.

3.3.3 CoRReCT

Rahman, Roy, and Collins [16] proposed a code reviewer recommendation system called CoRReCT (Code Reviewer Recommendation based on Cross-project and Technology experience) which aimed at recommending coder reviewers based on relevant cross-project work history, as well as the experience of developers in a specific specialised technology related to the pull-request[2]. These two information sources were used for determining the developer's experience for code reviewing. The basic idea behind their proposed CRRS is that if the past pull requests have similar libraries or specialised technologies to the current pull requests, then the code reviewers who reviewed those pull requests can be considered as potential code reviewers for the current pull requests.

According to the authors' proposed idea, the developers having more experience in external libraries and the adopted specialised technologies in the change files in the token set of the current pull requests are considered as more appropriate choices to perform code review than the ones with less

[2] A pull requests are a mechanism in a version control system for a developer to notify team members that they have completed a feature and are requesting that their changes be merged into the master code base.

experience.

The authors conducted an exploratory study with commercial projects and 10 external libraries with specialized technologies present in them. The authors hypothesize that two pull requests with shared libraries and common technologies would be similar in the files that are changed. Based on this assumption, they calculated the cosine similarity using the library or technology names as a bag of tokens. The bag of tokens is divided into two sets of tokens, one for the current pull request and one for the past pull request. The cosine similarity value ranges from 0 to 1 with 0 being complete dissimilarity of libraries and technologies, and 1 being a complete similarity of libraries and technologies. Next the authors proceeded to calculate the similarity estimates (as a proxy to review expertise) to the corresponding code reviewers of the past pull requests.

3.3.4 TIE

Xia, Lo, Wang, *et al.* [13] proposed a hybrid and an incremental approach called TIE (**T**ext m**I**ning and a fil**E** location-based approach) which utilizes the advantages of the text mining and a file location-based approach for code reviewer recommendation. The idea behind this approach is to analyse textual content in a review request using an incremental text mining model and calculate the similarity between the reviewed file paths and changed file paths using a similarity model.

The overall architecture of TIE is divided into three phases: model construc-

tion, recommendation and model update as shown in Figure 3.2.

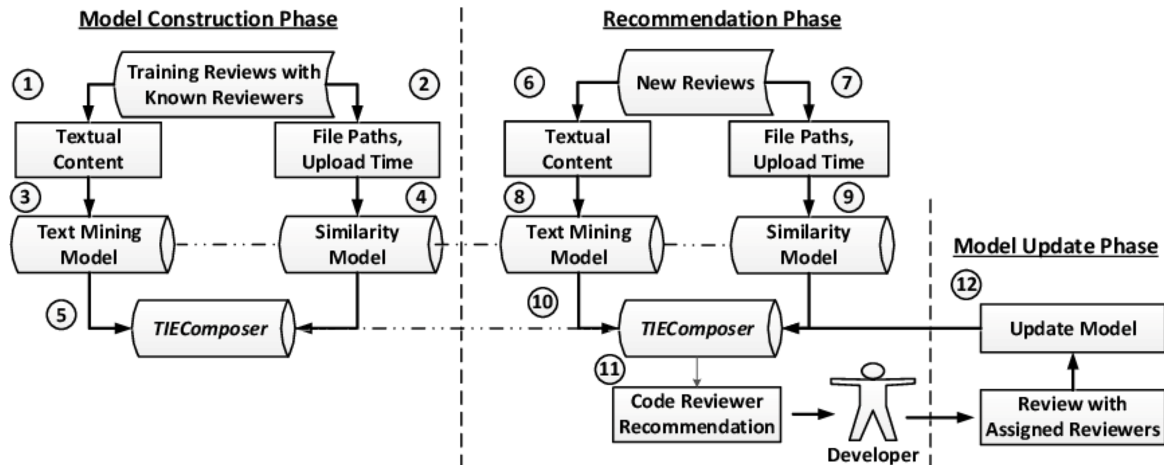

Figure 3.2: Architecture of TIE [13]

1. **Model Construction Phase**

   The model construction phase consists of a composite model called **T**IE**C**OMPOSER which is constructed using the historical reviews of known reviewers. In this phase, the TIE system first collects the training reviews of known reviewers from the textual content of past reviews and file paths, as well as the upload time. Next, TIE will build a text mining model based on the textual data processed using a text classification technique. The intuition behind the data mining mode is that the same reviewers are more likely to review the changes with similar terms or words.

   TIE also uses a time-aware file location based approach which aims at calculating the similarity between new and historic reviews. This similarity is calculated between the changed file paths (i.e. those paths of files that have been changed or modified in the new review request)

and the reviewed file paths (i.e. paths of files that have been reviewed in historical reviews). The intuition behind the file-location based approach is that the same reviewers tend to review the same files or files with similar paths.

These two models are blended together to build the **T**IE**C**OMPOSER model.

2. **Recommendation Phase**

   For this phase, TIE is used to recommend code reviewers for the new unassigned review request. TIE first extracts the change description, file paths and the upload time as it was done for the historical reviews in the 'Model Construction Phase'. For the next step, the textual data is extracted from the description and used as input in the data mining model constructed in the 'Model Construction Phase'. Similarly, the system also inputs the file paths and upload time in the similarity model constructed in the 'Model Construction Phase'.

   These two models then output a list of code reviewers and these two lists are then combined by leveraging the **T**IE**C**OMPOSER model constructed in the 'Model Construction Phase'.

3. **Model Update Phase**

   In the model update phase, the TIE system is updated using the newly assigned code reviewers. In practice, the developers normally check the

list of potential reviewers and then assign a new pull request to a group of reviewers.

In order to evaluate the performance of TIE, the authors used the data sets provided by Thongtanunam, Tantithamthavorn, Kula, *et al.*[12] containing 42,045 reviews and compared TIE's performance with RevFinder [12]. Each of the reviews in these data sets was labeled either 'merged' or 'abandoned' and contained at least one file path. It was seen that on an average across 4 open source projects TIE achieved top-1, top-3, top-5 and top-10 prediction accuracies of 0.52, 0.73, 0.79 and 0.85 and Mean Reciprocal Rank (MRR) value of 0.64 which beat the RevFinder results by 61%, 33%, 23%, 8% and 37% respectively.

3.3.5 CodeFlow

Bosu, Greiler, and Bird [11] did an empirical study at Microsoft on the characteristics of the useful code reviews by conducting interviews of developers, as well as analyzing the review comments of five Microsoft projects made using the CodeFlow CRRS. The study was conducted in three steps. First, they performed an exploratory study by conducting an interview of developers to understand their interpretation of 'useful' in the context of code reviews. Secondly, they build a classifier to segregate the 'useful' and 'not useful' comments using the data from the interviews. Lastly, they applied their classifier to five Microsoft projects to distinguish the 'useful' and 'not

useful' comments. Figure: 3.3 [11] shows the three-stage research method.

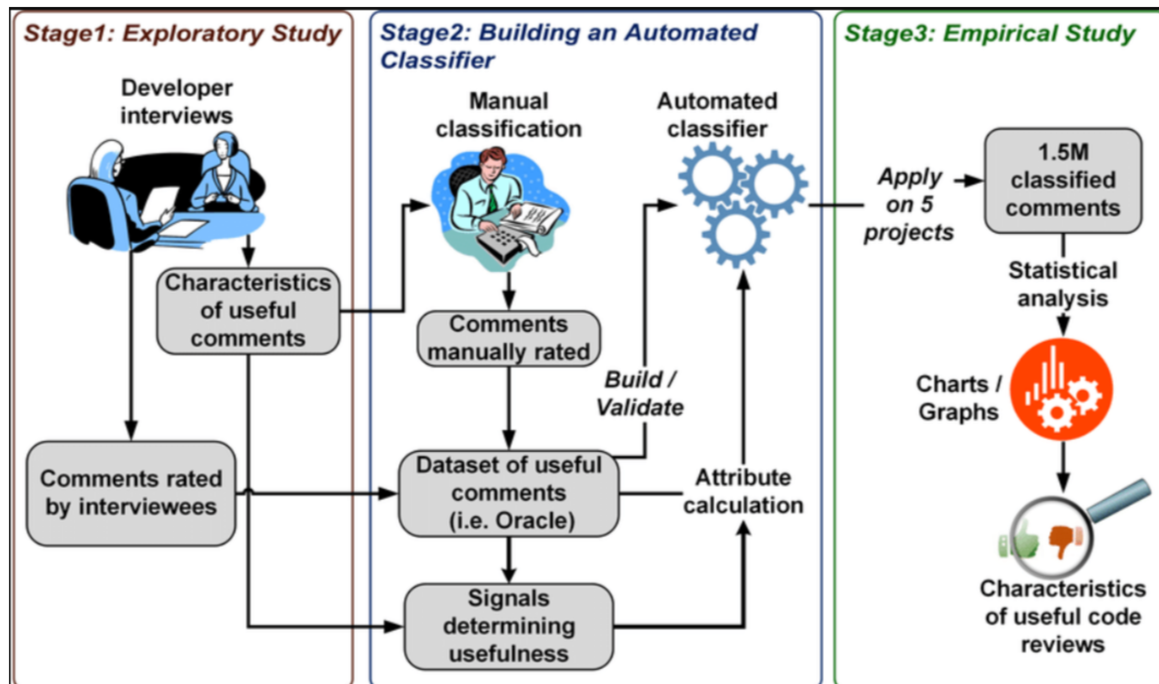

Figure 3.3: Three stage research method[11]

The workflow of CodeFlow is relatively straightforward. First, an author submits a review change and the reviewer is notified about the review request via email. Then the reviewer can review the change in the tool itself. When a reviewer wants to comment about a line or block of code, the reviewer highlights and adds comments for that part of the code. These comments appear as threads where the discussion starts, as well as interaction points for the people that are involved in the review. Each of these threads have a status that the participants can change during the course of review. This status is initially 'Active' and over time can be changed to 'Pending', 'Resolved', 'Won't Fix' and 'Closed'. In CodeFlow, each update is termed an 'iteration' and it constitutes another review cycle. Therefore, there could be numerous

iterations before the change in the code is finally merged into the source code repository.

As mentioned previously, the research study was done in three steps where the first step helped in distinguishing the useful and not useful code review comments based on interviewing developers. A *semi-structured individual interviews* of developers having different levels of experience in code reviewing and code development from four different Microsoft projects was conducted. The interviewees were asked to rate the comments from scale 1 - 3 (1- *Not Useful*, 2- *Somewhat Useful* and 3- *Useful*). The results of the interview showed that 69% of the review comments were either 'useful' or 'somewhat useful'. The review comments that indicated *functional defects* were considered as *useful* comments. On the other hand, the comments that belonged to the categories of: *documentation in the code*, *visual representation of the code* (e.g. blank line or indentation), *organization of the code* (e.g. how functionality is divided into methods) and *solution approach* were considered as *somewhat useful*. All of the comments that were either *false positives* (e.g. due to the lack of expertise when a reviewer incorrectly points out a problem in the code) or did not fall into any category as mentioned before were categorized as *Not Useful* comments.

In the second phase, the authors built an automated classifier using the findings obtained from the first phase. In order to build the classifier, the authors classified the review comments manually into two categories, *Useful*

and *Not Useful*. The comments that were classified as *Somewhat Useful* in the exploratory study were included into the *Useful* category for this second phase. Based on the interview and the manual analysis, the next 8 attributes of comments were identified. Based on these attributes and categories a 'Decision Tree Model to Classify Useful Comments' was built as shown below.

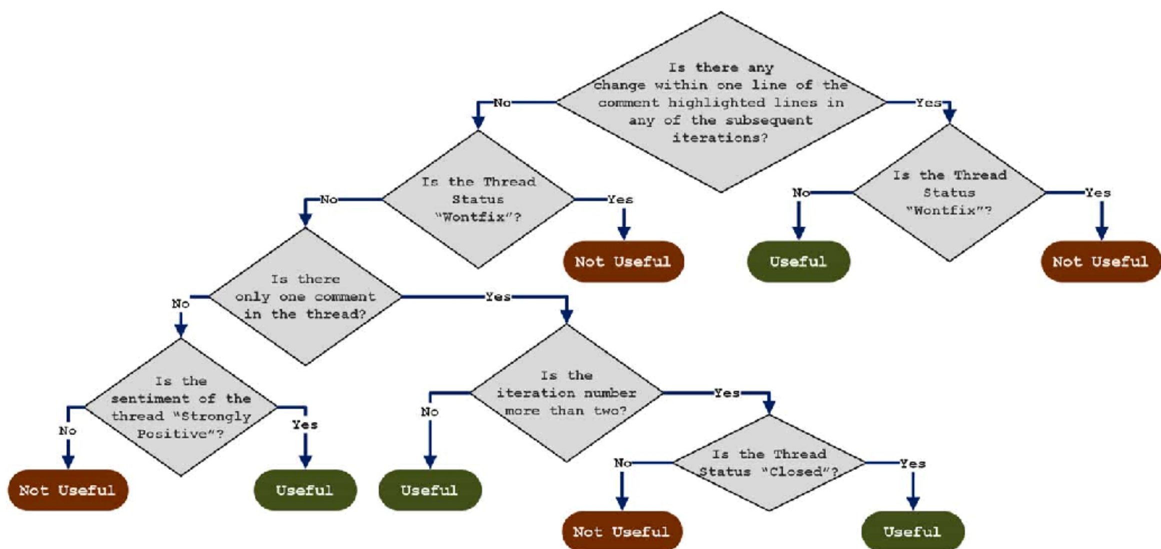

Figure 3.4: Decision tree model to classify useful comments[11]

Based on the decision nodes, the comments are classified as *useful* or *not useful*. In order to assess the proposed methodology, the authors used the comments from five major Microsoft projects which included Azure, Bing, Visual Studio, Exchange and Office. Based on the results, the authors concluded the following:

1. The developers who made changes or reviewed a piece of code or an artifact in the past give more useful comments.

2. There is a noticeable difference in the usefulness between the comments (i.e. those comments that have words like 'fixed', 'bug' or 'remove' were considered as 'useful' comments) that are made by the reviewers on the same team and the comments made by the author and reviewer from different teams.

3. The number of useful comments increased over time for four out of five projects and the reason behind this was considered to be the increased experience of reviewers with time.

Below are the implications of the results for the code review participants as well as for researchers:

1. The study showed that the number of the usefulness of code review comments increased with the experience of the developer of a code base.

2. The study also suggested that the effectiveness of the reviews decreased with the increase in the number of files. It was suggested that developers should submit smaller changes with more number of files for review.

3. The comment usefulness density can be used by a team of developers to identify areas where code reviews are less effective.

3.3.6 CRITIQUE

Sadowski, Söderberg, Church, *et al.*[22] did a case study where they conducted an exploratory study of Modern Code Review practices at Google.

Their exploratory study focused on 3 aspects of code review: 1) The motivations driving the code review, 2) the present practices and 3) interpretation of developers of code reviews.

In order to bring more structure into reviewing the code, several tools emerged in the Open Source Software (OSS) and industrial settings. For this, the authors studied some tool-based review approaches. These tools include *CodeFlow* [11] used by Microsoft, *Gerrit* [23] used by Google's Chromium, *ReviewBoard* [3] developed by VMware and *Phabricator* [24] used by Facebook. The following is a short overview of each of these code reviewer recommendation systems.

1. **CodeFlow:** CodeFlow tracked the status of each person (developer or reviewer) and where they stood in the process (i.e. waiting, reviewing, signed off). CodeFlow did not stop the author from submitting any changes without approval as well as provided support for chats in the comment threads.

2. **Gerrit:** Google's Chromium used the externally available code reviewer recommendation system called Gerrit where the changes are merged into the master branch only after approval from the reviewers and an automated verification that the change does not break the build.

3. **ReviewBoard:** ReviewBoard was developed by VMware and aims at integrating static analysis into the review process. This integration relies

on changes for which the authors are manually requesting analysis, resulting in quite an improvement in the code review quality.

4. **Phabricator:** Phabricator, which is used by Facebook, allows a reviewer to "take over" a change and commit it themselves. Also, the system provides a fix for automatic static analysis or continuous integration errors.

In order to understand the code review process at Google, the authors focused on two main aspects: the review process that the developers experience during specific reviews and whether the developers are satisfied with the reviews given despite the challenges. For the code review at Google, they used *CRITIQUE*, an internally developed, centralized web-based code review tool. In this tool, a reviewer can see the highlighted diff of the proposed change as well as start a threaded discussion over lines of code with developers or other reviewers. *CRITIQUE* also offers a view of all of the logging functionalities of a developer, as well as its interaction with the tool which include opening the tool, making changes, viewing the diff and approving the changes. In order to understand the motivation of the developers for code review at Google using *CRITIQUE* and the perceptions of the developers regarding the same, the authors used interviews as a tool to collect the data.

Based on the data collected from the interviews conducted, the following findings that were obtained.

Finding 1: The code reviews done at Google are not only aimed at correcting

the errors or problems, but also to ensure the code readability and maintainability which was considered as an educational aspect.

Finding 2: Expectations about a specific code review depends upon the relationship shared by a developer/author and a code reviewer (see Figure 3.5.).

When it comes to the developer and the project lead, as well as new team members, they share the education (teaching or learning from a code review) aspect of code review. For developer and other teams, they share the gatekeeping (establishment and maintenance of boundaries around source code) aspect of code review. Similarly, for the developer and readability reviewers, they share the maintaining norms (maintaining organizational rules such as formatting or API usage patterns) aspect of code review. Finally, for the developer and other team members, they share the education accidental prevention (teaching the bugs, defects or other quality related issues) aspect of code review.

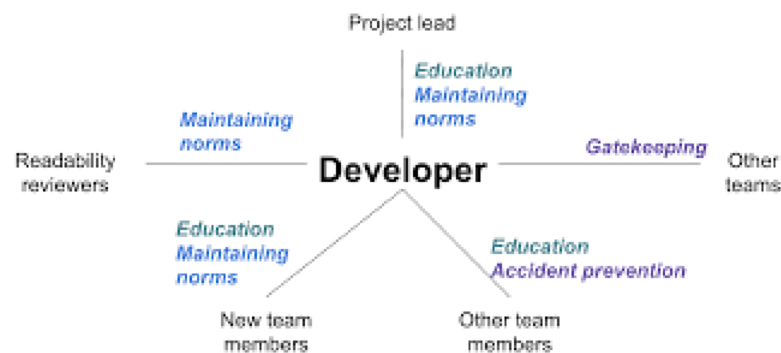

Figure 3.5: Relationship diagram that describes the themes of review expectations appearing primarily in a particular author/review context[22]

Finding 3: The code review process is aligned with the convergent practice of it being lightweight and flexible. The code review process is tightly combined with *CRITIQUE* which works as follows:

Creating The authors start creating, adding or editing a code.

Previewing With the help of *CRITIQUE*, the authors will then view the diff of the change and the results of the automatic code analyzers.

Commenting The authors/reviewers will see the diff in the UI of *CRITIQUE* and then start commenting as they go from one change to another.

Addressing Feedback Based on the comments from the previous steps, the authors will either start replying to comments or start making changes as per the requests made mentioned in the comments.

Approving Once all the comments have been addressed, the reviewers than approve the changes and mark it as 'LGTM' (Looks Good To Me).

Finding 4: Code reviews at Google have come to a point where the review process has become quicker with smaller changes when compared to the older projects. Also, one reviewer is considered to be enough, as compared to two reviewers required for older projects.

Finding 5: Despite the years of improvement, there have been a number of coding breakdowns at Google which are mostly linked to the complexity of the interactions that revolve around the code reviews.

It was seen that during the period of one week, almost 70% of the changes were committed in less than 24 hours after mailing out for the initial review. Based on the interviews, it was also seen that the developers were happy with the requirement to code review, the majority of the changes were small, reviews have one reviewer, and no comments other than the authorization to commit. These characteristics have made the process of code review faster and more light weight as compared to the other projects adopting similar process.

3.3.7 Profile-based CRRS

Fejzer, Przymus, and Stencel [20] proposed a profile- based code review recommendation system. In the proposed reviewer recommendation model, the reviewer's profile includes the review history and commits of a potential reviewer.

In their reviewer recommendation model, when a new commit request arrives in the repository, it is compared to the multi-set representation of commits (multiple sets of sequence of words present in a modified file path in a commit) as well as profiles of reviewers. The similarity between the multi-set representation of commits and profiles of reviewers is calculated and the top *n* reviewers are selected. Here, updating a reviewer's profile is one of the most important and frequently performed operations. Whenever a new comment is made by a reviewer, the commit gets added to his/her profile. Also, when it comes to a potential profile of a reviewer, time is one of the

important factors that needs to be taken into consideration. A candidate who has more recent reviews or commits in his/her profile is considered as a more probable candidate to review a commit request.

The authors did an empirical evaluation of their proposed method using Android, LibreOffice, OpenStack and Qt. The empirical results were as follows:

1. The number of reviews per a single reviewer: Most reviewers created less than 20 reviews for Android and LibreOffice and less than 60 reviews for OpenStack and Qt.

2. Duration of individual reviewers' activity: In the case of Android and LibreOffice, the reviewers took more time as compared to reviewers for Qt and OpenStack. The probable cause behind this result was considered to be the designated maintainers working for companies contributing to these projects.

3. Duration of individual reviews: Most of the reviews were completed within three days for LibreOffice and OpenStack projects, up to two days for Qt and up to six days for Android projects.

3.3.8 Categorization of systems

The categorization of the systems was done based on the data used for making a code reviewer recommendation. The data include similar file paths, code review histories, commits and technology experience.

Table 3.3: Data Sources for Code Review Recommendation Systems

System Name	Data Source
cHRev [17]	Code Review Histories
REVFINDER [12]	File Path Similarity
CodeFlow [11]	State of each reviewer or author
Gerrit [22]	Changes are merged after explicit approval from reviewers
ReviewBoard [22]	Integrates static analysis into the review process
Phabricator [22]	Automatic static analysis or continuous build/test integration
CORRECT [16]	Relevant cross-project and Technology Experience
TIE [13]	Text Mining and file location
Profile based CRRS [20]	Code reviewer profile

3.3.8.1 Project Used for Evaluation

Apart from the data source as one of the factors to categorize the research papers, the kind of project on which these code reviewer recommendation systems are experimented on can be considered as another factor on which the papers can be categorized. These projects include open source projects and commercial projects.

Table 3.4: Project used for evaluation

System Name	Type of Project
REVFINDER [12], TIE [13] and Profile based CRRS [20]	Open Source Projects
CodeFlow [11]	Commercial Projects
CORRECT [16] and cHRev [17],	Open Source and Commercial Projects
CRITIQUE [22]	No Project (Interview)

1. **Open Source Projects:** The follwing is a list of systems that were evaluated only on open source projects.

   For the evaluation purpose, RevFinder, TIE and Profile based CRRS used 42,045 reviews of open source projects which included Android Open Source Project (AOSP), OpenStack, Qt and LibreOffice. There were numerous reasons behind choosing these systems. First, these systems use the Gerrit system as the code review tool. Second, these systems are active, large, real-world software projects. Finally, each of these systems maintains a good code review system which helps to build a good oracle data set to evaluate the recommender system. Below mentioned are the results that we obtained through experimentation:

(a) RevFinder [12] achieved the top-10 accuracy (Top-*k* accuracy calculates the percentage of reviews that an approach can correctly recommend code reviewers over the total number of review) of 86%, 87%, 69% and 74% for Android, OpenStack, Qt and LibreOffice, respectively. On average, for 79% of the reviews, RevFinder ended up recommending CoRReCT code reviewers with a top-10 recommendation.

(b) It was seen that on average across 4 projects TIE [13] achieved top-1, top-3, top-5 and top-10 prediction accuracies and MRR values of 0.52, 0.73, 0.79, 0.85 and 0.64 which outperformed the RevFinder results by 61%, 33%, 23%, 8% and 37% respectively.

(c) Similar to TIE and RevFinder, the profile based CRRS [20] was experimented using 4 open source systems namely Android, OpenStack, Qt and LibreOffice. It was seen that for LibreOffice and OpenStack, that the majority of the reviews were completed within three days, for Android it took up to 6 days and for Qt it took up to 2 days.

2. **Commercial Projects:** CodeFlow [11] was experimented using five Microsoft projects which include Azure, Bing, Visual Studio, Exchange and Office. It was observed that there was an increase from 60% to 66% in the useful comments received from the reviewers in Azure, 62% to 67% in Bing, 60% to 70% in Visual Studio, 60% to 68% in Office and

60% to 65% in Exchange.

3. **Open Source and Commercial Projects:** There are some systems that were experimented using both the commercial projects and the open source projects which are mentioned below with the results obtained from the experiments performed.

   (a) CoRReCT [16] was experimented using 17,115 pull requests from ten commercial projects and six open source projects. The performance metrics that the authors used here include Top-K Accuracy, Mean Reciprocal Rank(MRR), Mean Precision(MP) and Mean Recall(MR).

      - When experimented on **open source projects** it was seen that CoRReCT has a Top-k accuracy of 85.20% whereas for **commercial projects** Top-K accuracy of 92.15% was achieved.
      - CoRReCT obtained a result of 85.93% precision for commercial projects and a precision of 84.76% for the open source projects.
      - For the commercial projects CoRReCT returned a recall of 81.39% whereas for the open source project the system achieved 78.73% recall.
      - CoRReCT obtained a MRR value of 0.62 for commercial projects whereas for the open source projects the authors did not mention the value of MRR but it was comparatively higher

(b) cHRev [17] was evaluated on 3 open source projects (Android, Mylyn and Eclipse) and one commercial project (MS Office). It was seen that the recall and precision gains obtained for MS Office were better than those achieved in Android, Mylyn and Eclipse for cHRev.

4. **No project (Interview)**

   CRITIQUE [22] which is used as a code review recommendation system at Google used the mode of interview to evaluate their system. It was seen that the developers spent on average of 2.6 hours a week reviewing changes which was low compared to the 6.4 hours/week of self-reported time for the Open source projects.

3.4 Summary

Based on the literature review study conducted, we found seven coder reviewer recommendation systems: cHREv, CoRReCT, profile-based CRRS, RevFinder, CodeFlow, TIE and CRITIQUE. These systems where divided based on two dimensions: the data source used to build the system and the type of project used to evaluate the system.

Chapter 4

Information Needs of Code Reviewers

4.1 Methodology

In order to conduct the survey of software engineers to determine the information needs for code reviewers we used the following steps to ensure that correct, non-biased and accurate results were obtained.

4.1.1 Screening Survey

Our survey for software engineers is divided into two parts where the first part is a screening survey. We used the screening survey in order to make sure we get responses from software product development members who have experience with code reviewer recommendation systems and can therefore provide accurate and unbiased information. Below are the questions that we included for our screening survey.

```
1. Please enter your email address.
```

2. How many year/s of software development experience do you have?
   (a) Less than 1 year
   (b) 1-2 year/s
   (c) 3-5 years
   (d) 6-10 years
   (e) 11+ years
3. Are you 20-years or older and able to provide informed consent?
   (a) Yes
   (b) No
4. How many years of experience you have in using code reviewer recommendation system/s?
   (a) Less than 1 year
   (b) 1-2 year/s
   (c) 3-5 years
   (d) 6-10 years
   (e) 11+ years
5. Which of the following code reviewer recommendation systems (CRRS) are you familiar with? (Multiple answer question)
   (a) Gerrit Code Review System (Chromium)
   (b) GitHub/GitLab
   (c) Code Flow Review Tool (Microsoft)
   (d) Review Board (VMware)

```
   (e) Phabricator
   (f) Bitbucket
   (g) Other
6. Which CRRS you have used?
7. If you have used a CRRS that is not listed, please
   provide their name or description of the system?
```

4.1.2 Demographic and Code Reviewer Recommendation Systems/Tools Usage Survey Questions

A demographic and CRRS experience survey was given to those that passed the screening step that is the participants having at least two years of work experienc and have an experience in using CRRSs. These questions helped us understand the information needs of code reviewers, what features they considered to be important in the code reviewer recommendation systems and what features they found missing in the existing systems.

```
1. What is your job role regardless of the position
   level in your organization?
   (a) Developer/Programmer/Software Engineer
   (b) Team Lead
   (c) DevOps Engineer/Infrastructure Developer
   (d) Architect
   (e) UI/UX developer
```

(f) Technical Support
(g) Data Analyst/Data Scientist/Data Engineer

2. What is your age group?

(a) 20-25
(b) 26-35
(c) 36-45
(d) 46-55
(e) 56-60
(f) Above 60

3. What is your geographic location?

(a) Europe
(b) Africa
(c) South America
(d) North America
(e) Asia
(f) Australia
(g) New Zealand
(h) Pacific Islands

4. What size is your project team?

(a) I work on my projects individually

(b) 2-7 people
(c) 8-12 people
(d) 13-20 people
(e) 21-40 people
(f) More than 40 people

5. Is your team distributed or co-located across the world?

   (a) Distributed
   (b) Co-located
   (c) Both

6. Which of the following code reviewer recommendation systems (CRRS) are you familiar with? (Multiple answer question)

   (a) Gerrit Code Review System (Chromium)
   (b) GitHub/GitLab
   (c) Code Flow Review Tool (Microsoft)
   (d) Review Board (VMware)
   (e) Phabricator
   (f) Bitbucket
   (g) Other

7. Which CRRS you have used?

8. If you have used a CRRS that is not listed, please provide their name or description of the system?

9. Which features present in the above mentioned CRRS proved to be useful? (Multiple answer question)

   (a) Pre commit code review
   (b) Code discussion with old and new versions being highlighted to show the change in code
   (c) Code improvement suggestion by the code reviewer (Other than just pointing out the code errors)
   (d) Prioritizing code changes based on its level of importance and its effect on the functionality of the software
   (e) Integration of project tracking software (such as Trello, JIRA etc.).
   (f) Integration of source-code editor (such as Visual Studio, Atom etc.).
   (g) Integration of business communication platform (such as Slack)

10. The following is a list of criteria that can be used for selecting a code reviewer. Please indicate how important you believe they are to selecting a code reviewer. (Likert scale: Extremely likely, somewhat likely, neither likely nor unlikely, somewhat unlikely, extremely unlikely)

(a) Number of years of work experience

(b) Code reviewer's expertise in programming language

(c) Code reviewer's expertise in a domain(such as software engineering, artificial intelligence etc.)

(d) Language of communication between the code reviewer and software developer

(e) Role of the code reviewer

(f) Count of projects worked on

(g) Count of code reviews done

11. What criteria were missing from the above list? What importance would you give them?

12. Which of the following User Interface (UI) features would make the User Experience (UX) more interactive, approachable, and convenient to use? (Multiple answer question)

    (a) Presence of a dashboard for everyone showing the statistical data of all actions performed (such

as number of commits, number of code reviews done, number of code errors/warnings in the current project etc.)

(b) An option to select a specific ‘branch/file’ in a project to maintain a systematic workflow and an organized code review procedure

(c) Presenting a pipeline showing which stage the project is in i.e. build, test, code review, deployment etc.

(d) Moved code detection using colour coded scheme with the developer/s name.

(e) Colour coded new and old code discussions when there is change in code.

13. Do you think that field of expertise of a code reviewer is important even if the reviewer has a little experience /knowledge in the field he is asked to review the code in?

14. What are some specific things you will look for in a code review (such as number of years of work

```
    experience, field of expertise etc.)?

15. At what point in your workflow would you prefer to
    have the code review recommendation?

    (a) Before the merge conflicts

    (b) After the merge conflicts

    (c) Does not matter at what point in the workflow the
        code review is done


16. What kind of code reviews would you prefer from the
    following?

    (a) More number of tiny code reviews

    (b) A long code review

    (c) Does not matter (depending on the kind of
        project)
```

4.2 Results

We obtained 27 queries about our survey but only 15 of those moved forward and responded to our screening survey. From these 15 responses we filtered out 4 responses which did not meet the minimum criteria using the

Screening Survey. Below are the results obtained for the *Demographic and Code Reviewer Recommendation Systems/Tools Usage Survey*.

1. **Job role of the participants**

   Based on the results obtained it was seen that most participants were either developers, or software engineers or programmers.

   Table 4.1: Job roles of the participants

Field	**Percentage**
Developer/Programmer/Software Engineer	72.73%
Team Lead	9.09%
DevOps Engineer/Infrastructure Engineer	9.09%
Product Owner	9.09%

   As seen in Table 4.1, 72.73% were developers/programmers/software engineers whereas there were 9.09% for each of teams leads, DevOps Engineer/Infrastructure Developer and Product Owner respectively.

2. **Geographic location of the participants**

   Most of our participants were from Asia and the remaining equivalent percentage of people were from North America and South America as shown in Table 4.2.

Table 4.2: Geographic location of the participants

Field	**Percentage**
South America	9.09%
North America	9.09%
Asia	81.82%

3. **Size of the project team**

   Of all of the participants who participated in our study, 45.45% of them have worked/are working in a team of 2-7 people and 8-12 people respectively whereas 9.09% of people have worked/are working in a group of more than 40 people as shown in Table 4.3.

Table 4.3: Size of the project team

Field	**Percentage**
2-7 people	45.45%
8-12 people	45.45%
More than 40 people	9.09%

4. **Distribution of the team**

   We also gathered information on how the teams are distributed, meaning if they are working in teams that are co-located or distributed. It was observed that 45.45% of the teams were co-located, 18.18% were distributed and 36.36% were a mix of both co-located and distributed.

Table 4.4: Distribution of the team

Field	**Percentage**
Co-located	45.45%
Distributed	18.18%
Both	36.36%

5. **Familiarity of the CRRS amongst the participants**

   The majority of participants that took part in our survey were familiar with *GitHub/GitLab*. On the other hand, no one was familiar with *Gerrit Code Review System (Chromium)* and *ReviewBoard*. It was observed that 47.62% of the participants were familiar with GitHub/GitLab, 23.81% of the participants were familiar with BitBucket, 14.29% of the participants were familiar with Code Flow Review Tool (Microsoft), 9.52% were familiar with other review tools (which include SVN) and 4.76% of the participants were familiar with Phabricator. Below is the line graph that shows the distribution of the familiarity of the the CRRS amongst the participants.

Table 4.5: Familiarity of the CRRS amongst the participants

CRRS	**Percentage of choices**
Gerrit Code Review System (Chromium)	0%
GitHub/GitLab	47.62%
Code Flow Review Tool(Microsoft)	14.29%
Review Board (VMware)	0%
Phabricator	4.76%
Bitbucket	23.81%
other	9.52%

6. **Usefulness of the features of CRRS**

   We listed a number of CRRS features to our participants and asked them to select all those features that proved to be useful for them. The following are the features that we posed to the participants with the percentage of participants that found it useful.

   (a) *Code discussion with old and new versions being highlighted to show the change in code:* 25.53% of the participants found it useful.

   (b) *Integration of project tracking software (such as Trello, Jira etc.):* 20.59% of the participants found it useful.

   (c) *Pre commit code review:* 14.71% of the participants found it useful.

   (d) *Code improvement suggestion by the code reviewer (Other than just pointing out the code errors):* 14.71% of the participants found

it useful.

(e) *Integration of business communication platform (such as Slack):* 11.76% of the participants found it useful.

(f) *Integration of source-code editor (such as Visual Studio, Atom etc.):* 11.76% of the participants found it useful.

(g) *Prioritizing code changes based on its level of importance and its effect on the functionality of the software:* 2.94% of the participants founds it useful.

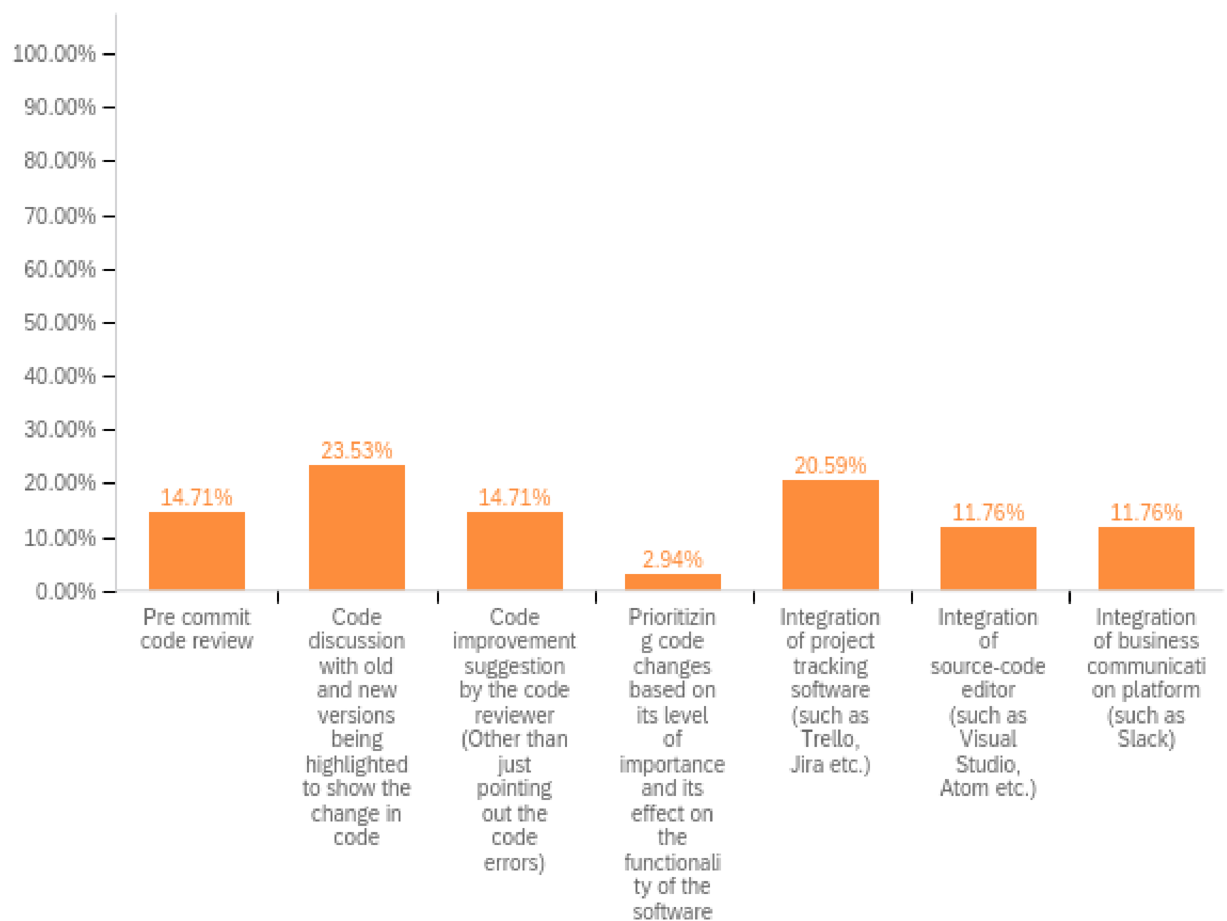

Figure 4.1: Reported usefulness of CRRS features

7. **Criteria for selecting a code reviewer**

   There are a number of factors that need to be taken into consideration when selecting a code reviewer. We listed several of them from which the participants selected the importance of each of them. The participants marked the importance of each of these features on a likert scale ranging from *Extremely Likely* to *Extremely Unlikely* as shown below.

#	Field	Extremely likely	Somewhat likely	Neither likely nor unlikely	Somewhat unlikely	Extremely unlikely	Total
1	Number of years of work experience	9.09%	72.73%	9.09%	9.09%	0.00%	11
2	Code reviewers expertise in programming language	54.55%	36.36%	9.09%	0.00%	0.00%	11
3	Code reviewer's expertise in a domain (eg> software engineering, artificial intelligence etc.)	45.45%	45.45%	9.09%	0.00%	0.00%	11
4	Language of communication between the code reviewer and software developer	54.55%	18.18%	27.27%	0.00%	0.00%	11
5	Role of the code reviewer	18.18%	45.45%	36.36%	0.00%	0.00%	11
6	Count of projects worked on	9.09%	63.64%	0.00%	27.27%	0.00%	11
7	Count of code reviews done	27.27%	45.45%	9.09%	18.18%	0.00%	11

Showing rows 1 - 7 of 7

Figure 4.2: Criteria for selecting a code reviewer

   (a) **Number of years of work experience**

   - Of all the participants **9.09%** of them considered this feature as *extremely likely*, **72.73%** of them considered it as *somewhat likely*, **9.09%** of them considered it as *neither likely nor unlikely*, **9.09%** considered them as *somewhat unlikely* whereas no one considered it as *extremely unlikely*.

   (b) **Code reviewers expertise in programming language**

- It was observed that **54.55%** of the participants considered this feature as *extremely likely*, **36.36%** of them considered it as *somewhat likely*, **9.09%** of them considered it as *neither likely nor unlikely* whereas no one considered it as *somewhat unlikely* or *extremely unlikely*.

(c) **Code reviewer's expertise in a domain (e.g: software engineering, artificial intelligence etc.)**

- For this criteria, **45.45%** of the participants considered this feature as *extremely likely*, **45.45%** of them considered it as *somewhat likely*, **9.09%** of them considered it as *neither likely nor unlikely* whereas no one considered it as *somewhat unlikely* or *extremely unlikely*.

(d) **Language of communication between the code reviewer and software developer**

- Of all the participants **54.55%** of them considered this feature as *extremely likely*, **18.18%** of them considered it as *somewhat likely*, **27.27%** of them considered it as *neither likely nor unlikely* whereas no one considered it as *somewhat unlikely* or *extremely unlikely*.

(e) **Role of the code reviewer**

- For this criteria, **18.18%** of the participants considered this

feature as *extremely likely*, **45.45%** of them considered it as *somewhat likely*, **36.36%** of them considered it as *neither likely nor unlikely* whereas no one considered it as *somewhat unlikely* or *extremely unlikely*.

(f) **Number of projects worked on**

- It was also observed that **9.09%** of the participants considered this feature as *extremely likely*, **63.64%** of them considered it as *somewhat likely*, **27.27%** considered them as *somewhat unlikely* whereas no one considered it as *extremely unlikely* and *neither likely nor unlikely*.

(g) **Number of code reviews done**

- For this criteria, **27.27%** of the participants considered this feature as *extremely likely*, **45.45%** of them considered it as *somewhat likely*, **9.09%** considered them as *somewhat unlikely*, **18.18%** of them considered it as *neither likely nor unlikely* whereas no one considered it as *extremely unlikely*.

We also asked the participants to provide some features other than those presented that they considered as important features when selecting a code reviewer. The reported features include:

- Making sure the reviewer is keeping himself/herself updated with technology and programming language (the participant

considered this feature as *extremely likely)*

- Number of pull requests assigned to reviewers

8. **User Interface (UI) of the CRRS:**

   Apart from the features of CRRS, we also asked a question regarding numerous different UI features that can be found useful for a CRRS by making the User Experience (UX) more interactive, approachable and convenient to use. The percentage of people who found the following features useful are as follows with a pie chart distribution shown below:

   (a) *Presenting a pipeline showing which stage the project is in i.e. build, test, code review, deployment etc.* : 31.25% of the participants found this UI feature useful and more convenient to keep the track of the project.

   (b) *Presence of a dashboard for each individual showing the statistical data of all actions performed (such as number of commits, number of code reviews done, number of code errors/warnings in the current project etc.)*: 28.13% of the participants found this feature useful to look up details and actions performed by each individual.

   (c) *Colour coded new and old code discussions when there is change in the code*: 15.63% of the participants found this feature useful making the code review process easier for the reviewer as well as developer.

(d) *Moved code detection using colour coded scheme with the developer/s name*: 12.50% of the participants found this feature useful.

(e) *An option to select a specific 'branch/file' in a project to maintain a systematic workflow and an organized code review procedure*: 12.50% of the participants found this feature useful by making the code review process more organized.

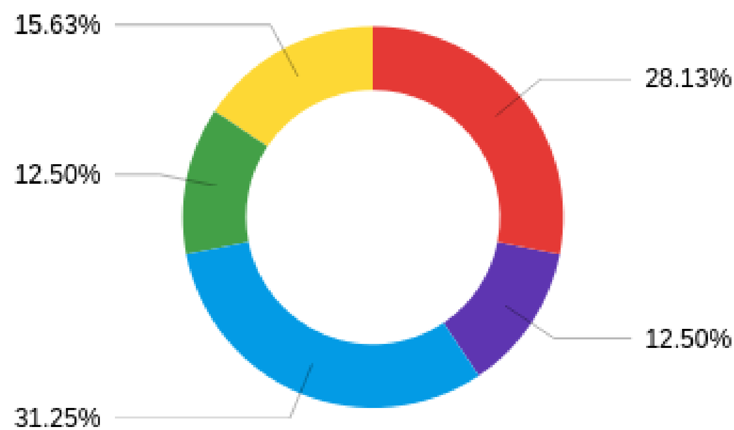

Figure 4.3: UI features of CRRS

9. **Preference of when to have the code review recommendation:**

   A code review recommendation can be made at various stages of the software development process which include before the merge conflicts, after the merge conflicts or at any point in the workflow. It was seen that 60% of the participants believed that the code recommendation should be done before the merge conflicts, 20.00% of them thought it should be done after the merge conflicts whereas to 20.00% of them it did not matter at what point in the workflow the code review is done.

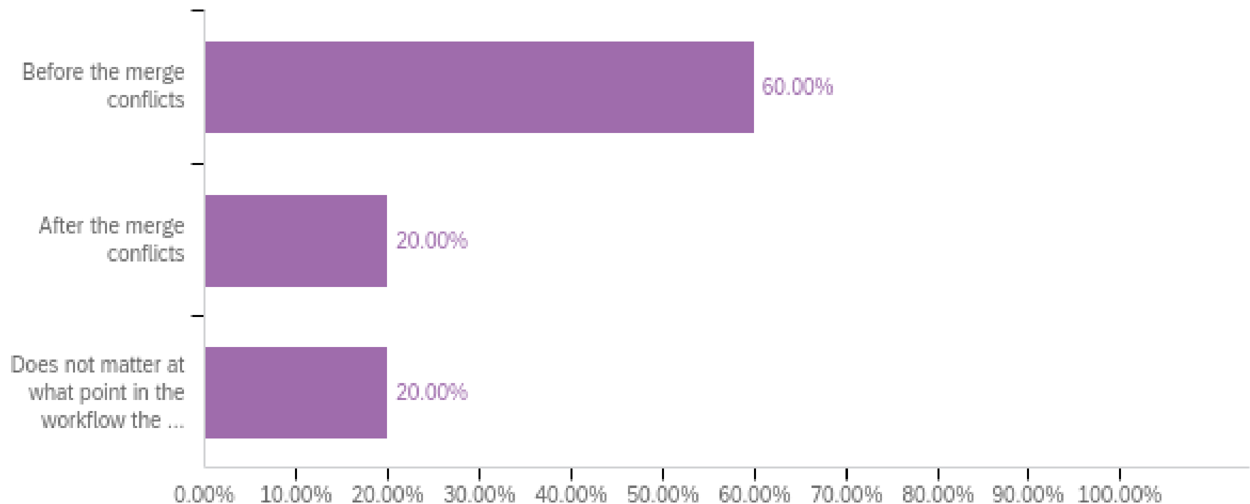

Figure 4.4: Preference of when to have the code review recommendation

10. **Type of code review**

    The participants were asked as to what type of code review would they prefer which includes either many smaller code reviews or a long code review or depends on the kind of project. It was seen that 60.00% of the participants preferred to do many smaller code reviews whereas the rest (40.00%) of the participants felt that it did not matter. None of the

participants preferred to do long code reviews.

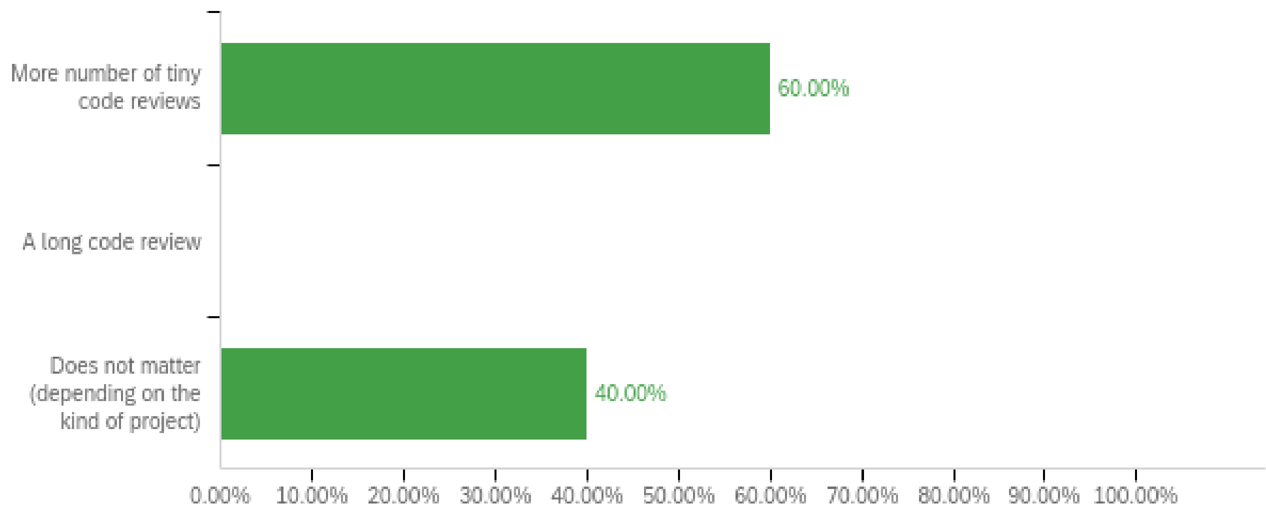

Figure 4.5: Kind of code reviews

4.3 Some observed trends and patterns

Based on the results obtained we found some patterns and trends between the results of two or more survey questions. Some of the trends that we observed are as follows:

(a) **The type of CRRS used and the job role**

Based on our observations, it was seen that the Developer was aware of almost all the CRRS systems that we had mentioned in the question whereas the DevOps Engineer was just aware of one CRRS, the Team lead was aware of 2 of the CRRSs and the product owner knew about 4 of the CRRSs.

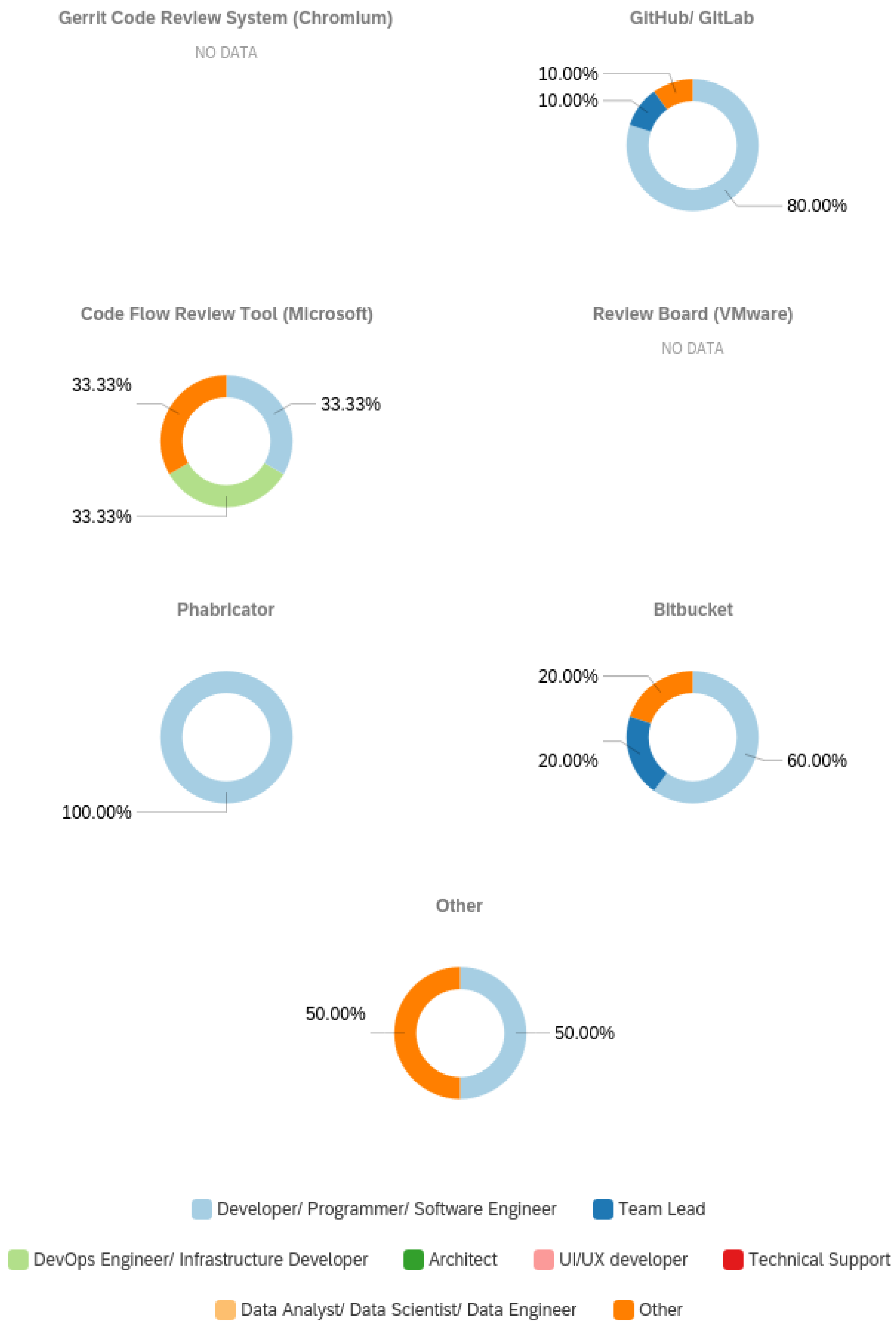

Figure 4.6: Type of CRRS and job role

We obtained the following results for each of the following CRRS along with the pie chart distribution in Figure 4.6:

i. **Github/GitLab:** Of participants who were familiar with this CRRS, 10% of the participants were team lead, another 10% were product owners and the rest (80%) were the developers/programmers/software engineers.

ii. **Code Flow Review Tool(Microsoft:** Of the participants that were familiar with this CRRS, 33.33% were developers/programmers, software engineers, DevOps Engineer/Infrastructure Developer and others.

iii. **Phabricator:** Developers/programmers/Software engineers were the only people who were familiar with Phabricator.

iv. **BitBucket:** Of the participants who were familiar with this CRRS, 60.00% of them were developers/programmers/software engineers, 20.00% of them were team leads and others.

v. **Others:** There are a number of other CRRSs in the market and the participants were asked to select that option if they knew of any other CRRSs not mentioned in the survey. For, we got 50.00% responses from developers as well as others.

Of the participants that took part in our survey, none of them were familiar with two of the Code Review Recommendation Systems/-Tools: ReviewBoard (VMware) and Gerrit Code Review System

(Chromium).

(b) **CRRSs features and job role**

We posed a number of CRRS features to the participants which they considered to be important to make use of the CRRSs more convenient and easier to use. Figure 4.7 shows what job role found which features useful.

#	Field	Pre commit code review	Code discussion with old and new versions being highlighted to show the change in code	Code improvement suggestion by the code reviewer (Other than just pointing out the code errors)	Prioritizing code changes based on its level of importance and its effect on the functionality of the software	Integration of project tracking software (such as Trello, Jira etc.)	Integration of source-code editor (such as Visual Studio, Atom etc.)	Integration of business communication platform (such as Slack)	Total
1	Developer/ Programmer/ Software Engineer	12.50%	25.00%	8.33%	0.00%	20.83%	16.67%	16.67%	24
2	Team Lead	0.00%	33.33%	33.33%	0.00%	33.33%	0.00%	0.00%	3
3	DevOps Engineer/ Infrastructure Developer	33.33%	33.33%	33.33%	0.00%	0.00%	0.00%	0.00%	3
4	Architect	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0
5	UI/UX developer	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0
6	Technical Support	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0
7	Data Analyst/ Data Scientist/ Data Engineer	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0
8	Other	25.00%	0.00%	25.00%	25.00%	25.00%	0.00%	0.00%	4

Showing rows 1 - 8 of 8

Figure 4.7: CRRS features and job role

i. **Pre commit code review:**

It was seen that of all the participants who agreed to consider this feature important 60% of them were developers, 20% were

DevOps engineer/Infrastructure developer and Others (i.e. product owner).

ii. **Code discussion with old and new versions being highlighted to show the change in code:**
Of all the participants, 75% of them were developers, 12.50% of them were team leads and DevOps engineer/infrastructure developer who considered this CRRS feature as important.

iii. **Code improvement suggestion by the code reviewer (other than just pointing out the code errors):** Of all of the participants who found this feature useful, 40.00% of them were developers, 20.00% of them were team leads, DevOps engineer/Infrastructure developer and others (product owner).

iv. **Prioritizing code changes based on its level of importance and its effect on the functionality of the software:** Of all the participants, only the product owner found this feature to be useful.

v. **Integration of project tracking software (such as Trello, Jira, etc.):** For this feature, 71.43% of the developers, 14.29% of the team leads and project manager found it useful.

vi. **Integration of source-code editor (such as Visual Studio, Atom etc.):** Only developers were the ones from all the participants who considered this feature as an important one.

vii. **Integration of business communication platform (such as Slack):** This feature was found useful only by the developers of all the participants who participated in our survey.

4.4 Summary

We presented the results obtained by surveying a broad range of software project members here where we found which CRRS features were found to be most useful, which features were missing in the existing system and what factors were important when choosing a relevant code reviewer. We also obtained a developer/reviewers preference to what kind of code reviews should be done (long or short review) and at what stage in the workflow it should be done. Apart from this, we also found some trends and patterns between the usage of the CRRS system and its relation with the demographic information of the reviewer/developer.

In the following chapter we answer and discuss our research questions

Chapter 5

Discussion

This section answers and discusses our research questions. By conducting the systematic literature review, we found answers to the first three research questions and using the survey we found answers to the last two research questions

By conducting a Systematic Literature Review (SLR) we identified some existing solutions for recommendation systems for code reviewers, factors that need to be taken into consideration when creating a CRRS and categorization of the existing CRRSs. On the other hand, by conducting the survey we found out the features that are important for a recommendation system for code reviewers and what improvements can be made in the existing CRRSs.

[RQ1.] What are the existing solutions for recommendation systems for code reviewers?

Answer: We reviewed a number of papers and found a number of existing code review recommendation systems. These systems/tools include *cHRev, REVFINDER, CoRReCT, TIE, CodeFlow, ReviewBoard,*

Phabricator, Gerrit, rDevX and Profile-based CRRS which make recommendations based on a number of factors/data types. These data types include code review history, file path similarity, relevant cross-project and technology experience, text-mining and file-location and tracking state of each reviewer or author.

[RQ2.] What are the factors that need to be taken into consideration when creating a recommendation system for code reviewers?

Answer: The primary factor to take into consideration when creating a recommendation system for code reviewers is the data source evaluation metrics or the type of project on which the system was experimented (i.e. open source or commercial or both). When it comes to recommending a code reviewer based on the reviewer's profile, it necessary to keep the reviewer's profile updated which include the past review and commit history. Similarly, when it comes to past review history, it is important to update the repository/data set of past reviews to recommend relevant code reviewers in the future based on the past reviews.

[RQ3.] How can existing recommendation systems for code reviewers be categorized?

Answer: The existing recommendation systems have been categorized based on the datatype which include code review history, file path similarity, relevant cross-project and technology experience, text-mining

and file-location, tracking state of each reviewer or author. *cHRev* was a code reviewer recommendation system that recommended code reviewers based on the code review histories. *REVFINDER* was another CRRS that recommended code reviewers based on the file path similarity. Similarly, we found a CRRS called *CoRReCT* which aimed at recommending code reviewers based on the relevant cross-project and technology experience. *TIE* (**T**ext m**I**ning and a fil**E**) as the name says, recommends code reviewers with the help of text mining and file location.

[RQ4.] What are the important features for a recommendation system for code reviewers?

Answer: Based on the software project member survey that was conducted, we found a number of features that were considered to be important for a recommendation system for code reviewers which include:

1. Code discussion with old and new versions being highlighted to show the change in code.
2. Integration with an issue tracking software such as Trello, JIRA etc.
3. Pre-commit code review.
4. Code improvement suggestions by the code reviewer, beyond point-

ing out the code errors.

5. Integration with a source-code editor, such as Visual Studio or Atom.
6. Integration with a business communication platform, such as Slack or MS Teams.
7. Prioritizing code changes based on its level of importance and its effect on the functionality of the software.
8. Presenting a pipeline showing which development stage the project is in including build, test, code review and deployment.
9. Presence of a dashboard for all project members showing the statistical data of all actions performed, such as number of commits, number of code reviews done, and number of code errors/warnings in the current project.
10. Colour-coded new and old code discussions when there is change in code.
11. An option to select a specific branch or file in a project to maintain a systematic workflow and an organized code review procedure.
12. Moved code detection using a colour-coded scheme with the developer/s name.

[RQ5.] How can existing recommendation systems for code reviewers be improved? In other words, what features are missing from

existing implementations for recommendation systems of code reviewers?

Answer: Based on the survey results the following are some of the features that can be improved or are missing in a code reviewer recommendation systems or when looking for a relevant code reviewer:

1. When it comes to selecting a code reviewer, the participants believed that the reviewers expertise in the project's programming language, field of expertise, years of work experience, code quality expertise and understanding of the project architecture are important factors.
2. Some of the participants believed that, number of years of work experience, as well as field of expertise, are both important. The reasoning was that these factors can prove useful when it comes to finding an optimized approach to a problem and giving suggestions for LLD (low-level design). Also, these factors are helpful in writing a standard code of practice which comes from experience and expertise.

5.1 Proposal for an Improved Code Review Recommender System

Based on the findings from our research, we propose an improved code reviewer recommendation system that would have all the necessary features that are not present in all systems or features found to be missing in the exist-

ing systems. The proposed recommender system would have the following features:

1. More transparent in the terms that all of the details about a code reviewer would be visible on a dashboard. These details include the number of projects they have worked on (i.e. their work experience), the specific application field in which they hold expertise, the number of code reviews done by them, and their workload (i.e. the number of reviews the reviewer is currently reviewing) to make sure the reviewer is not overburdened with new code reviews. It is believed that providing these details will thereby speed up the code review process.

2. A broader combination of data for training the recommender. This data set will include past review comments and commit messages, and relevant cross-project and technology experience. This data will help in knowing if the code that needs to be reviewed is a close match to the project experience a reviewer holds. Similarly, the past history of review comments and commits will help in choosing a relevant reviewer based on the number of commits and reviews previously done by them and how useful those past code reviews turned out to be for the developers. This will help to ensure that their future review comments will be useful for the code reviews.

3. Code reviews will happen before code merges and potential code con-

flicts. Therefore, the proposed recommendation system will recommend the code reviewers before merge conflicts may happen. However, the system will also allow the choice of the code review to be done after merge conflicts to avoid delays if needed, and yet still ensure the creation of a good quality software product.

Chapter 6

Conclusion

In this research we identified a number of code reviewer recommendation systems (CRRS) found in the literature, the different ways in which these systems can be categorized into, what features are important for a recommendation system for code reviewers and how the existing systems can be improved or which features are missing in existing CRRSs. A systematic literature review study was conducted to identify the existing code reviewer recommendation systems and understand the details regarding these systems, which includes the features and factors that are important for a CRRS. Then we conducted a survey to understand the needs of software project members regarding code reviewer recommendation systems as to which features they found to be important in a CRRS and what can be improved in the existing CRRSs.

6.1 Contributions:

This research makes the following contributions:

C1: A ranking of the features present in the existing code reviewer recommendation systems as to which of these were found to be most useful.

C2: A categorization of existing CRRSs from the literature along with different dimensions.

C3: Improvements that can be made in the existing code reviewer recommendation systems.

C4: Features that are important while selecting a code reviewer.

6.2 Future Work

Possible future directions based on this work include:

A broader systematic literature review: An expanded systematic literature review study could be conducted which looks at not just code reviewer recommendation systems but also code review practices and procedures. This could help to give a better picture about the needs of software project members regarding code reviews in junction with the use of with the CRRSs.

Building a code review recommendation system: For future work, we aim at building a system that would have all the details of the reviewer visible on the system dashboard (i.e. work experience, technology experience, number of code reviews done etc). Also, the system would have more

data to train the recommender system which include past review comments and commit messages, relevant cross-project and technology experience. Based on the feedback obtained from survey, the reviews would be done before the merge conflicts happen.

Chapter 7

Appendix

7.1 Ethical Review Certificate

Office of Research Ethics
4401 University Drive
Lethbridge, Alberta, Canada
T1K 3M4
Phone: (403) 329-2747
Email: research.services@uleth.ca
FWA 00018802 IORG 0006429

Monday, 19 October 2020

Student Investigator:	Palak Halvadia, Graduate Student
Faculty Supervisor:	John Anvik, Mathematics & Computer Science
Study Title:	Assessing current attitudes and needs of code reviewer recommendation systems
Action:	Approved
HPRC Protocol Number:	2020-094
Approval Date:	October 19, 2020
Term Date:	November 30, 2020

Dear Palak,

Your human research ethics application titled "Assessing current attitudes and needs of code reviewer recommendation systems" has been reviewed and approved on behalf of the University of Lethbridge Human Participant Research Committee (HPRC) for the approval period **October 19, 2020 to November 30, 2020**, and assigned Protocol #2020-094. The HPRC conducts its reviews in accord with University policy and the Tri-Council Policy Statement: Ethical Conduct for Research Involving Humans (2018).

Please be advised that any changes to the protocol or the informed consent must be submitted for review and approval by the HPRC before they are implemented. A final report will be required and is due to the Office of Research Ethics on or before November 30, 2020.

We wish you the best with your graduate research.

Sincerely,

Susan Entz, M.Sc., Ethics Officer
Office of Research Ethics
University of Lethbridge
4401 University Drive
Lethbridge, Alberta, Canada
T1K 3M4

Bibliography

[1] D. Gusfield, *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. 1997.

[2] D. Cubranic, G. C. Murphy, J. Singer, and K. S. Booth, "Hipikat: A project memory for software development," *IEEE Trans. Software Eng.*, vol. 31, no. 6, pp. 446–465, 2005. DOI: 10.1109/TSE.2005.71. [Online]. Available: https://doi.org/10.1109/TSE.2005.71.

[3] VMware, *Reviewboard*, 2006. [Online]. Available: https://www.reviewboard.org/.

[4] B. Kitchenham and S. Charters, "Guidelines for performing systematic literature reviews in software engineering," *Journal of Software Engineering and Applications*, 2007.

[5] H. H. Kagdi, M. Hammad, and J. I. Maletic, "Who can help me with this source code change?," pp. 157–166, 2008. DOI: 10.1109/ICSM.2008.4658064. [Online]. Available: https://doi.org/10.1109/ICSM.2008.4658064.

[6] E. W. T. Ngai, L. Xiu, and D. C. K. Chau, "Application of data mining techniques in customer relationship management: A literature review and classification," *Expert Syst. Appl.*, vol. 36, no. 2, pp. 2592–2602, 2009. DOI: 10.1016/j.eswa.2008.02.021. [Online]. Available: https://doi.org/10.1016/j.eswa.2008.02.021.

[7] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell, “A systematic literature review on fault prediction performance in software engineering,” *IEEE Trans. Software Eng.*, vol. 38, no. 6, pp. 1276–1304, 2012. DOI: 10.1109/TSE.2011.103. [Online]. Available: https://doi.org/10.1109/TSE.2011.103.

[8] D. H. Park, H. K. Kim, I. Y. Choi, and J. K. Kim, “A literature review and classification of recommender systems research,” *Expert Syst. Appl.*, vol. 39, no. 11, pp. 10 059–10 072, 2012. DOI: 10.1016/j.eswa.2012.02.038. [Online]. Available: https://doi.org/10.1016/j.eswa.2012.02.038.

[9] A. Bacchelli and C. Bird, “Expectations, outcomes, and challenges of modern code review,” D. Notkin, B. H. C. Cheng, and K. Pohl, Eds., pp. 712–721, 2013. DOI: 10.1109/ICSE.2013.6606617. [Online]. Available: https://doi.org/10.1109/ICSE.2013.6606617.

[10] V. Balachandran, “Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation,” D. Notkin, B. H. C. Cheng, and K. Pohl, Eds., pp. 931–940, 2013. DOI: 10.1109/ICSE.2013.6606642. [Online]. Available: https://doi.org/10.1109/ICSE.2013.6606642.

[11] A. Bosu, M. Greiler, and C. Bird, “Characteristics of useful code reviews: An empirical study at microsoft,” M. D. Penta, M. Pinzger, and R. Robbes, Eds., pp. 146–156, 2015. DOI: 10.1109/MSR.2015.21. [Online]. Available: https://doi.org/10.1109/MSR.2015.21.

[12] P. Thongtanunam, C. Tantithamthavorn, R. G. Kula, N. Yoshida, H. Iida, and K. Matsumoto, “Who should review my code? A file location-based code-reviewer recommendation approach for modern code review,” Y. Guéhéneuc, B. Adams, and A. Serebrenik, Eds., pp. 141–150, 2015. DOI: 10.1109/SANER.2015.7081824. [Online]. Available: https://doi.org/10.1109/SANER.2015.7081824.

[13] 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," R. Koschke, J. Krinke, and M. P. Robillard, Eds., pp. 261–270, 2015. DOI: 10.1109/ICSM.2015.7332472. [Online]. Available: https://doi.org/10.1109/ICSM.2015.7332472.

[14] M. Gasparic and A. Janes, "What recommendation systems for software engineering recommend: A systematic literature review," *J. Syst. Softw.*, vol. 113, pp. 101–113, 2016. DOI: 10.1016/j.jss.2015.11.036. [Online]. Available: https://doi.org/10.1016/j.jss.2015.11.036.

[15] S. Lee and S. Kang, "What situational information would help developers when using a graphical code recommender?" *J. Syst. Softw.*, vol. 117, pp. 199–217, 2016. DOI: 10.1016/j.jss.2016.02.050. [Online]. Available: https://doi.org/10.1016/j.jss.2016.02.050.

[16] M. M. Rahman, C. K. Roy, and J. A. Collins, "Correct: Code reviewer recommendation in github based on cross-project and technology experience," L. K. Dillon, W. Visser, and L. Williams, Eds., pp. 222–231, 2016. DOI: 10.1145/2889160.2889244. [Online]. Available: https://doi.org/10.1145/2889160.2889244.

[17] M. B. Zanjani, H. H. Kagdi, and C. Bird, "Automatically recommending peer reviewers in modern code review," *IEEE Trans. Software Eng.*, vol. 42, no. 6, pp. 530–543, 2016. DOI: 10.1109/TSE.2015.2500238. [Online]. Available: https://doi.org/10.1109/TSE.2015.2500238.

[18] K. Haruna, M. A. Ismail, S. Suhendroyono, D. Damiasih, A. C. Pierewan, H. Chiroma, and T. Herawan, "Context-aware recommender system: A review of recent developmental process and future research direction," 2017. [Online]. Available: www.mdpi.com/journal/applsci.

[19] E. Schön, J. Thomaschewski, and M. J. Escalona, "Agile requirements engineering: A systematic literature review," *Comput. Stand. Interfaces*, vol. 49, pp. 79–91, 2017.

DOI: 10.1016/j.csi.2016.08.011. [Online]. Available: https://doi.org/10.1016/j.csi.2016.08.011.

[20] M. Fejzer, P. Przymus, and K. Stencel, "Profile based recommendation of code reviewers," *J. Intell. Inf. Syst.*, vol. 50, no. 3, pp. 597–619, 2018. DOI: 10.1007/s10844-017-0484-1. [Online]. Available: https://doi.org/10.1007/s10844-017-0484-1.

[21] V. Kovalenko, N. Tintarev, E. Pasynkov, C. Bird, and A. Bacchelli, "Does reviewer recommendation help developers?," 2018.

[22] C. Sadowski, E. Söderberg, L. Church, M. Sipko, and A. Bacchelli, "Modern code review: A case study at google," F. Paulisch and J. Bosch, Eds., pp. 181–190, 2018. DOI: 10.1145/3183519.3183525. [Online]. Available: https://doi.org/10.1145/3183519.3183525.

[23] Gerrit, *Gerrit*, 2021. [Online]. Available: https://www.gerritcodereview.com/.

[24] phacility, *Phabricator*, 2021. [Online]. Available: https://www.phacility.com/phabricator/.