CORMS: A GitHub and Gerrit based Hybrid Code Reviewer Recommendation Approach for Modern Code Review

by

Prahar Pandya
202011001

A Thesis Submitted in Partial Fulfilment of the Requirements for the Degree of

MASTER OF TECHNOLOGY
in
INFORMATION AND COMMUNICATION TECHNOLOGY
to

DHIRUBHAI AMBANI INSTITUTE OF INFORMATION AND COMMUNICATION TECHNOLOGY

June, 2022

Declaration

I hereby declare that

i) the thesis comprises of my original work towards the degree of Master of Technology in Information and Communication Technology at Dhirubhai Ambani Institute of Information and Communication Technology and has not been submitted elsewhere for a degree,

ii) due acknowledgment has been made in the text to all the reference material used.

Prahar Pandya

Certificate

This is to certify that the thesis work entitled "**CORMS: A GitHub and Gerrit-based Hybrid Code Reviewer Recommendation Approach for Modern Code Review**" has been carried out by **Prahar Pandya (202011001)** for the degree of Master of Technology in Information and Communication Technology at *Dhirubhai Ambani Institute of Information and Communication Technology* under my supervision.

Dr. Saurabh Tiwari
Thesis Supervisor

Acknowledgments

First and foremost, I would like to thank my supervisor, Dr. Saurabh Tiwari for providing all the help, support and motivation needed for my research work. Without his continuous guidance and trust, this thesis would not have been concluded. He was always there to lead me towards the solution whenever I got stuck.

Last but not least, a special a special thanks to my parents, my family and my friends for their continuous support.

Contents

Abstract vi

List of Principal Symbols and Acronyms vii

List of Tables viii

List of Figures ix

1 Introduction 1

1.1 Objective and Problem Description . . . 1
1.2 Motivation . . . 6
1.2.1 Observations and Implications . . . 8
1.3 Thesis Contribution . . . 9
1.3.1 Literature Review . . . 9
1.3.2 Mining Code Review Repositories . . . 9
1.3.3 Automating Reviewer Recommendation Process . . . 9
1.3.4 Performance Evaluation . . . 10
1.3.5 Tool Support . . . 10
1.4 Organisation of the Thesis . . . 10

2 Code Review 12

2.1 What is Code Review? . . . 12
2.2 Modern Code Review Process . . . 13
2.3 Benefits and Challenges . . . 14
2.4 Factors Influencing the Modern Code Review . . . 15
2.4.1 Non Technical Factors . . . 15
2.4.2 Technical Factors . . . 15

3 Literature Review 18

3.1 Traditional Approaches . . . 18
3.1.1 ReviewBot . . . 18

3.1.2 RevFinder 19
3.2 Approaches based on Reviewer Expertise 20
3.2.1 CORRECT 20
3.2.2 RevRec 21
3.3 Approaches based on Social Relations 22
3.3.1 Comment-Network 22
3.4 Hybrid Approaches 23
3.4.1 CoreDevRec 23
3.4.2 TIE 24
3.4.3 WhoReview 25
3.5 Summary 27

4 Code Reviewer Recommendation: Proposed Approach 28
4.1 Data Mining 29
4.1.1 Mining reviews from Gerrit 29
4.1.2 Mining reviews from GitHub 30
4.2 Data Pre-processing 31
4.3 Natural Language Processing and Vector Transformation 32
4.3.1 Natural Language Processing 32
4.3.2 TFIDF Vectorization 33
4.4 Data Splitting 33
4.5 Similarity Model 33
4.6 Ensemble Modeling 35
4.7 CORMS Controller 36

5 Experimentation and Results 38
5.1 Experimental Setup 38
5.1.1 Project Selection 38
5.1.2 Statistics of collected data 39
5.2 Evaluation Metrics 39
5.2.1 Top-K 40
5.2.2 Mean Reciprocal Rank (*MRR*) 40
5.3 Research Questions and Analysis 40
5.3.1 Research Questions 40
5.3.2 Analysis Results 43
5.4 Threats to Validity 44

6 CORMS: A Tool **46**
6.1 What is CORMS-TOOL? . . . 46
6.2 CORMS-TOOL Architecture . . . 47
6.3 Django MVC Framework of CORMS-TOOL . . . 48
6.4 Features of CORMS-TOOL . . . 49
6.5 Various Interfaces of CORMS-TOOL . . . 50
6.5.1 Code-Review Interface . . . 50
6.5.2 Results and Feedback Interface . . . 51
6.5.3 Create or View Project Interface . . . 52

7 Conclusions and Future Work **53**

References **55**

Abstract

Modern Code review (*MCR*) techniques are widely adopted in both open-source software platforms and organizations to ensure the quality of their software products. However, the selection of reviewers for code review is cumbersome with the increasing size of development teams. The recommendation of inappropriate reviewers for code review can take more time and effort to complete the task effectively. We carried out a detailed literature review over existing recommendation approaches and extended the baseline of reviewers' recommendation framework – *RevFinder*[1] to handle issues with newly created files, retired reviewers, the external validity of results, and the accuracies of the state-of-the-art *RevFinder*. Our proposed hybrid approach, *CORMS*, works on similarity analysis to compute similarities among file-paths, projects/sub-projects, author information, and prediction models to recommend reviewers based on the subject of the change. We conducted a detailed analysis on the widely used 20 projects of both Gerrit and GitHub to compare our results with *RevFinder*. Our results reveal that on average, *CORMS*, can achieve top-10, top-5, top-3, and top-1 accuracies, and Mean Reciprocal Rank (*MRR*) of 79.9%, 74.6%, 67.5%, 45.1% and 0.58 for the 20 projects, consequently improves the *RevFinder* approach by 12.3%, 20.8%, 34.4%, 44.9% and 18.4%, respectively. Finally, we built a complete tool - *CORMS-TOOL* based on our proposed approach, *CORMS*, to support reviewer recommendation process in modern code review.

[1]https://github.com/patanamon/revfinder

List of Principal Symbols and Acronyms

LCPrefix Longest Common Prefix

LCSubseq Longest Common Subsequence

LCSubstr Longest Common Substring

LCSuffix Longest Common Suffix

MCR Modern Code Review

MRR Mean Reciprocal Rank

OSS Open Source Software

SVM Support Vector Machine

List of Tables

2.1 Factors Influencing the Modern Code Review 17

3.1 Technique, Platform, Source Code, Data-sets, Metrics Used 26
3.2 Features used in most common Recommendation Algorithms . . . 26

5.1 Statistics of the data collected from OSS 39
5.2 Performance evaluation of *CORMS* and *RevFinder* 41
5.3 Performance of *CORMS* with Normalization and Borda Count score propagation techniques 42
5.4 Measurement of Accuracy Gain for each Individual Models 43

List of Figures

1.1 Code review interface of GitHub 2
1.2 Reviewer Assignment Problem at GitHub 3
1.3 Code review interface of Gerrit 4
1.4 Code-Review 836629 - OpenStack project of Gerrit 7
1.5 Code-Review 798228 - OpenStack project of Gerrit 8
1.6 Code-Review 836814 - OpenStack project of Gerrit 8

2.1 Working of Modern Code Review (MCR) 14

3.1 Working of Revfinder 19
3.2 Working of CORRECT 20
3.3 Working of RevRec 21
3.4 Working of CodeDevRec 23
3.5 Working of TIE 25
3.6 Working of WhoReview 25

4.1 Proposed Hybrid Approach: *CORMS* 28
4.2 Our Mining Workflow for Gerrit 30
4.3 Our Mining Workflow for GitHub 31

6.1 CORMS-TOOL architecture 46
6.2 Django MVC Framework of CORMS-TOOL 49
6.3 Step-1 Interface: OSS Platform Selection 50
6.4 Step-2 Interface: Project Selection 50
6.5 Step-3 Interface: Upload code-review JSON file 50
6.6 Results Interface 51
6.7 Submit Feedback Interface 51
6.8 Create New Project Request Interface 52
6.9 View Supported Projects Interface 52

CHAPTER 1

Introduction

This chapter gives the overview of the objectives and problem description, motivation, observation and implications and finally the thesis contribution.

1.1 Objective and Problem Description

Software code reviews help the development process in reducing overall costs and helps in knowledge transfer. The reviews conducted on the software code identify logical errors, coding rule violations, and also assisted with the automated tools. This review process is known as *MCR* (Modern Code Review) [3]. The MCR are nowadays used by both the organizations such as Microsoft [24] or Google [34], and by the open-source software (*OSS*) platforms such as GitHub[1], Gerrit[2] or review-board[3].

The existing works highlighted that selection and assignment of inappropriate reviewers downgrade the review process and quality of the software product [40]. o Patanamon Thongtanunam et al. [40] investigated reviews in Gerrit open-source system and found that 4%–30% of the reviews have reviewer assignment issues. As a result, on average, 12 more days may require to complete it. Jing Jiang et al. [19] also analyzed the pull requests in GitHub and found that 40.6% of manual assignments. Gerrit and GitHub are the two widely used OSS platforms for performing code reviews.

GitHub is a widely used OSS Platform. GitHub provides support for a pull-based development. In GitHub, developers first fork a repository and make changes to fix some bugs or implement new features. When ready with the changes, they submit a pull request to merge code changes into the main repository. This pull request needs to be evaluated by a reviewer. This reviewer is a trusted experi-

[1]https://github.com/
[2]https://www.gerritcodereview.com/
[3]https://www.reviewboard.org/

enced member of a community. The reviewer then checks the code changes and code legibility and decides whether to merge code changes into the main branch or not. If the reviewers have any confusion, they may ask the developer for clarification or make updates and submit new commits for reevaluation. Ideally, reviewers are assigned to pull requests without any delay after the creation of the change requests. However, in reality, some popular projects receive many pull requests, and reviewers find difficulties in prioritizing pull requests [18]. Since reviewers need to be assigned to pull requests soon after their creation, finding an appropriate reviewer can be a time-consuming task.

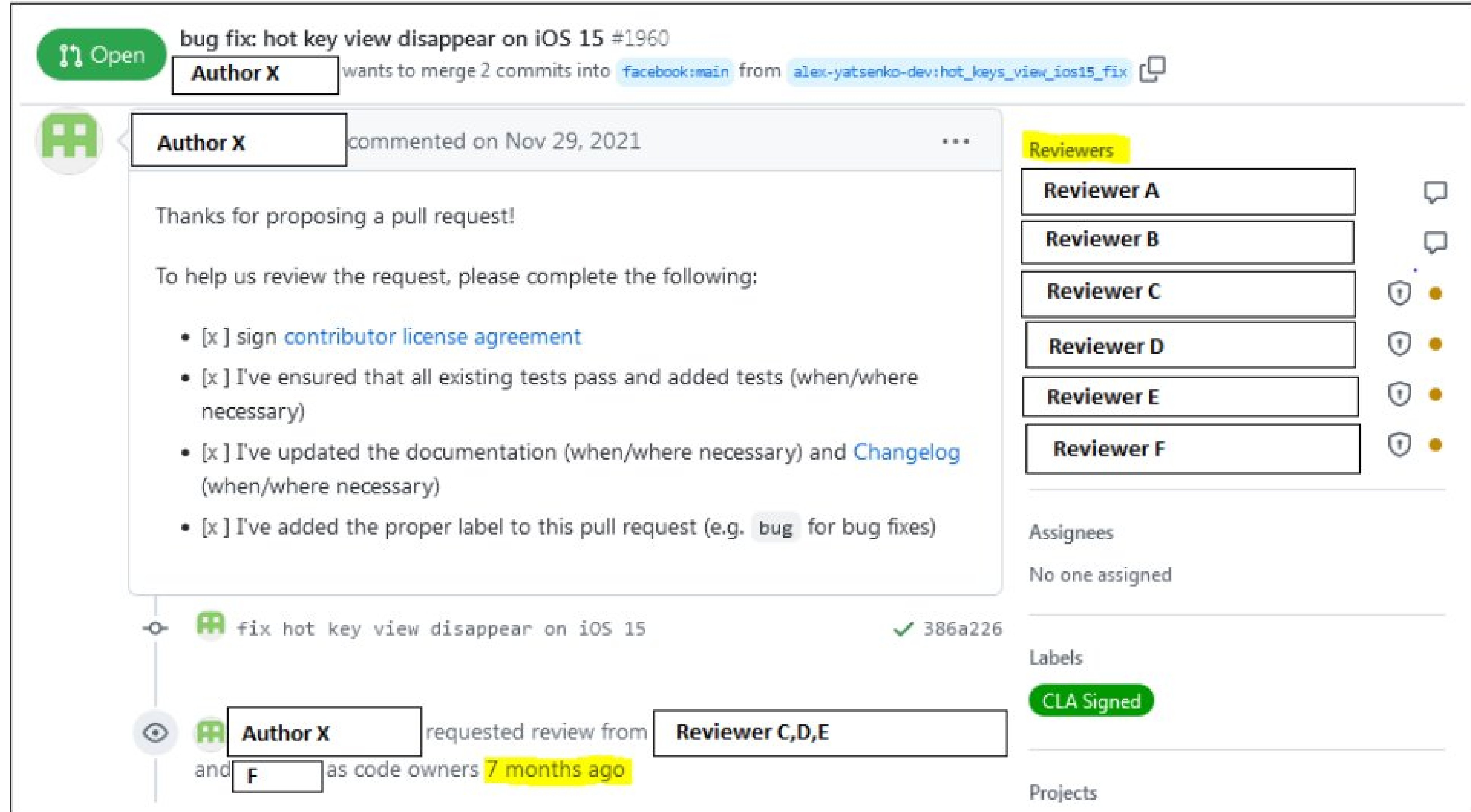


Figure 1.1: Code review interface of GitHub

Figure 1.1 shows the code review interface of GitHub for a Facebook project[4]. Here we can see that the *Author X* wants to merge 2 commits into the main branch of Facebook, but for that, he needs approval from reviewers. The status of this request is *open* and we can see that it is 7 months older. The right sidebar of the UI shows the number of reviewers and their status. In GitHub, these reviewers are further divided into core members and commenters. Core members are those who take the final decision for a merge, while commenters are those who help the code review process by providing their suggestions in comments. The lock symbol denotes that this reviewer is a code-owner. In the given example of code review interface of GitHub, the *Reviewer C,D,E and F* are the code-owners. GitHub introduced *code-owners*[5] to define individuals or teams that are responsible for

[4]https://github.com/facebook/facebook-ios-sdk/pull/1960
[5]https://docs.github.com/en/repositories/managing-your-repositorys-settings-and-

code in a branch. These code owners are automatically requested for review when someone opens a pull request that modifies code that they own. The yellow dot near reviewer denotes that requested review from this reviewer is still awaited. When any one of the reviewers approves the request, a green tick will appear next to that reviewer instead of a yellow dot and the author can able to merge the pull request. After a successful merge, the pull request status will be marked as *closed*.

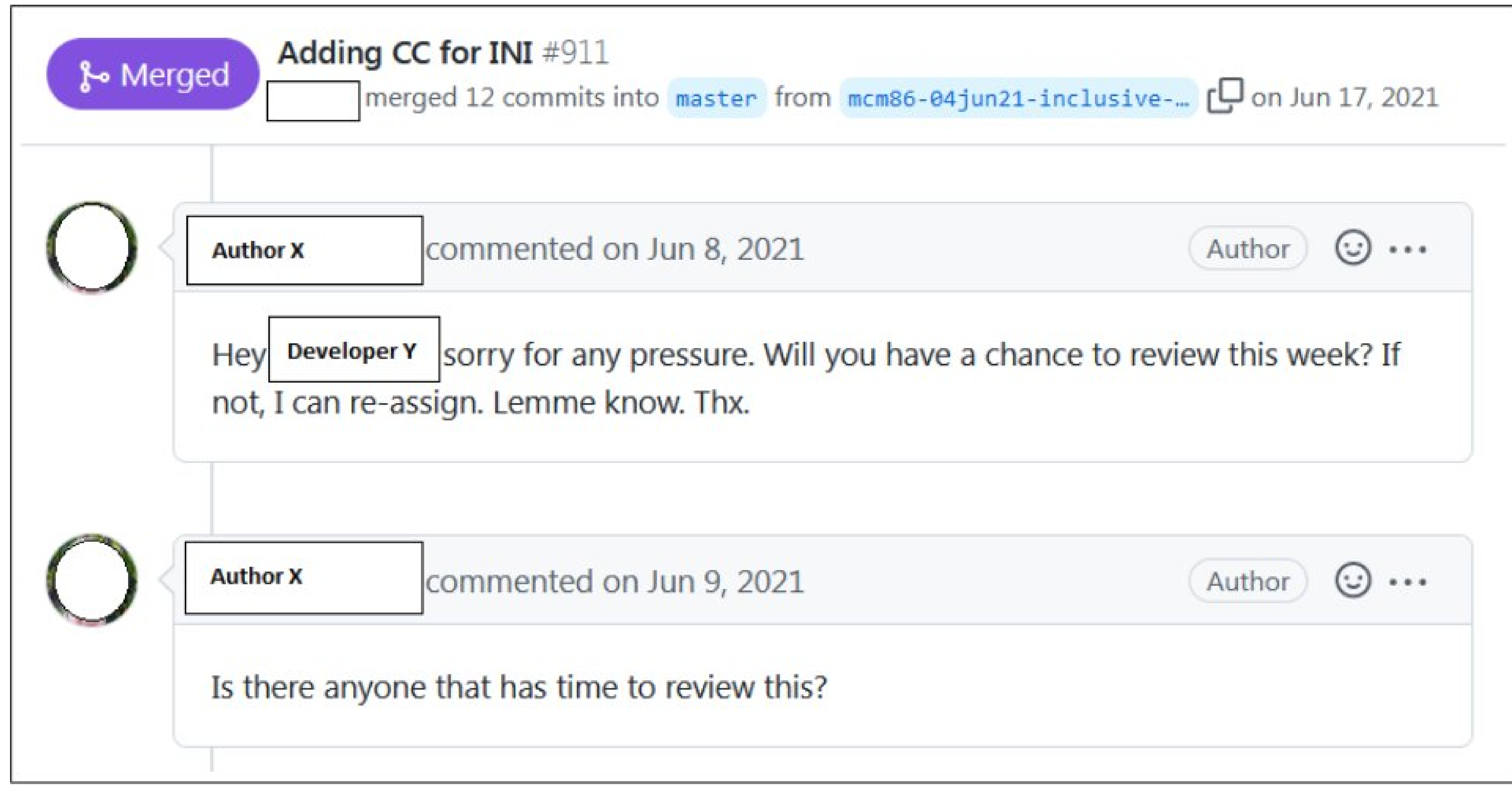


Figure 1.2: Reviewer Assignment Problem at GitHub

Figure 1.2 shows the reviewer assignment problem present at the *'change id-911'* of *betterscientificsoftware* project[6] of GitHub reviewed on Jun 8, 2021. Here, the *Author X* is requesting the reviewer *Developer Y* to review. However, he didn't get any reply. Hence, the *Author X* put the comment asking *'Is there anyone that has time to review my code?'*. The reviewers can be assigned to the pull requests immediately after their creation in an ideal scenario. However, it is not the common case at the moment. Recently, GitHub followed a round-robin and load balancer algorithm for recommending reviewers[7]. However, still this method doesn't count other features such as file path or expertise of reviewer or other social features.

Gerrit is also being used by many large open source projects, everyone with their unique setup in terms of what they require a code review to pass before being ready to submit into master. Gerrit reviews use *reviewUI*[8] which provides tool

features/customizing-your-repository/about-code-owners

[6]https://github.com/betterscientificsoftware/bssw.io/pull/911

[7]https://docs.github.com/en/organizations/organizing-members-into-teams/managing-code-review-settings-for-your-team

[8]https://gerrit-review.googlesource.com/Documentation/

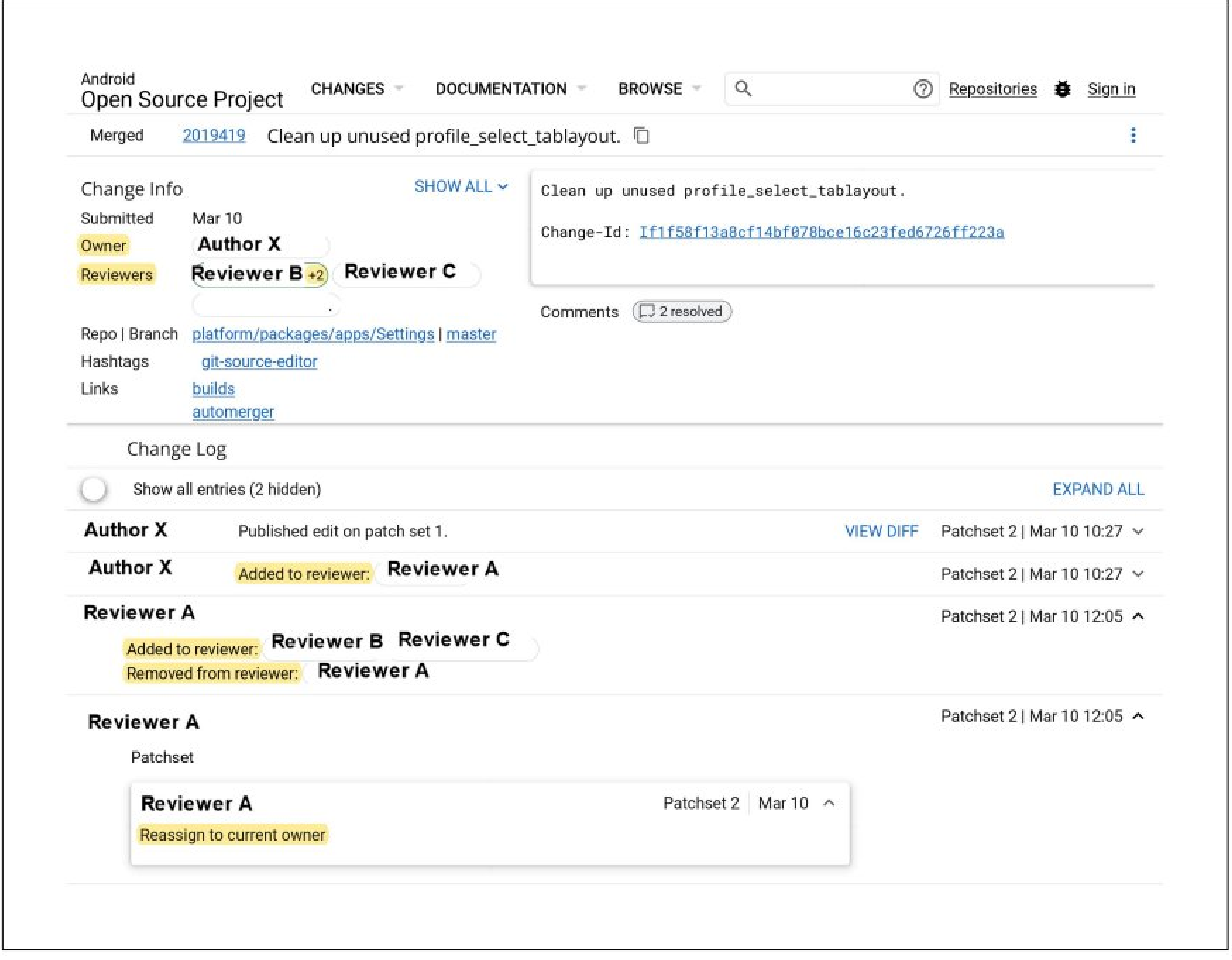


Figure 1.3: Code review interface of Gerrit

support to provide many functionalities to make the review process comfortable and efficient. In Gerrit, developers first clone a repository and make changes to fix some bugs or implement new features. When they are ready with the changes, they submit a pull request to merge code changes into the repository. Here, they have to provide details regarding the reference of a branch for which this change request is created. By default this branch setting is *master-branch*. After that, Gerrit *reviewUI* provides support to the author, where he can also perform tasks such as 1) comparing the changed lines with the previous version, 2) writing comments inline to ask reviewers for advice on a certain part of the change and 3) adding a list of reviewers. The author can manually add the reviewer or can use the plugins to automatically add reviewers to the change. The *Reviewers-plugin*[9] helps to configure default reviewers so that they can be automatically added to every change. The *Reviewers-by-blame-plugin*[10] helps to add reviewer by considering the reviewer code familiarity and experience. These added reviewers will get notified over email. In Gerrit, before the change is accepted, the following two checks must be completed: 1) Code-Review: It requires the reviewer to ensure that the code change meets project guidelines, 2) Verified: This check denotes that the

[9]https://gerrit-review.googlesource.com/admin/repos/plugins/reviewers,general
[10]https://gerrit-review.googlesource.com/admin/repos/plugins/reviewers-by-blame,general

code compiles successfully, passes tests, and performs without any issues. The *automated build server* in Gerrit checks this verified status through a *Gerrit Trigger*[11]. Finally, when reviewers make decisions, they have choices to vote from -2 to +2. +2 vote denotes that the change request looks good to the reviewer and is approved. +1 vote denotes that the change request looks good to the reviewer, but someone else must approve it. 0 vote denotes no score. -1 score denotes that the reviewer would prefer not to submit this change. -2 vote denotes that the reviewer made a decision to abandon this change request. A change must have at least one +2 vote and no -2 votes before it can be submitted.

Figure 1.3 shows the code review interface of a Gerrit android project[12]. This change request *'change id-2019419'* was reviewed on 10th March 2022. Here, we can notice that the *Author X* added the reviewer *Reviewer A*, however, he is not the actual reviewer, so he added two more reviewers *Reviewer B* and *Reviewer C* and removed himself as a reviewer as he is no longer an owner of this project. Finally, the *Reviewer B* approved the change by providing a +2 vote. The correct reviewer could be assigned to change requests without any delay after their creation in an ideal scenario. However, it is not a common case at the moment. The pull-based model has the potential to be more effective and enhancements in these areas.

It depends on the organization and their certain requirements to follow *GitHub fork-and-pull-model* or *Gerrit-code-reviews*. GitHub is the go-to place to host open-source projects. Gerrit can be indispensable for larger teams where there is intense code reviewing. Some similarities, as well as differences in GitHub and Gerrit's code review workflow, are:

1. While GitHub supports multiple commits in one pull request, Gerrit does not. GitHub's pull requests do not force the author to think about atomic or related changes as one commit. While Gerrit's reviews, by their restrictions (i.e. one commit per review), do force authors to think about this much more consciously, which is good for a code review.

2. Both Gerrit and GitHub provide the facility to push draft changes. By default draft changes are only visible to the author. This gives the author the possibility to have some staging before making the changes visible to the reviewers.

3. In GitHub, new members or authors have to fork the repositories, clone them locally, make the changes, push them to their fork and then create the

[11]https://wiki.jenkins-ci.org/display/JENKINS/Gerrit+Trigger
[12]https://android-review.googlesource.com/c/platform/packages/apps/Settings/+/2019419

pull request. With Gerrit, the author could clone the main repository, do his change, and then push it directly to the same remote as he cloned it from.

4. Gerrit provides voting facilities for a review, in which reviewers have choices to vote from -2 to +2. In GitHub reviewer can either approve the change or can abandon the change.

5. Compared to GitHub, the feedback loop is easier to follow and it's very easy to get up to speed with what's happened since the last time the author visited Gerrit. For example, if the author gets a -2 (i.e. rejected) on the Code-Review part, he can view the reviewer's comments in the *diff view* section in a certain code location. Now when the author is ready with the code change which fixes the reviewer's comments, the author can add the changes to his existing commit instead of creating a new commit locally, resulting in the author's feature still being only one commit. Then, the authors are only needed to simply push to Gerrit by *git push gerrit HEAD: refs/for/master* and the code review gets updated.

One may use both the Gerrit and GitHub together. The reason for using GitHub and Gerrit together are: 1) GitHub is widely recognized and accessible by lots of worldwide sites. 2) Using a public GitHub repository allows to *off-load* a lot of gits pull traffic. 3) Pull-request allows novice users to start getting involved. 4) Gerrit code-review defines the quality gates for avoiding *noise* of unstructured contributions. A GitHub plugin allows existing GitHub repositories to be integrated as Gerrit projects, while a Gerrit plugin can help control the GitHub replica and import the pull requests as Gerrit change requests.

1.2 Motivation

A typical review contains several fields such as the subject of the change, file paths, authors, reviewers, last reviewed date, numbers of inserted and deleted lines. The subject field contains textual information explaining in the abstract about the change. The reviewer field denotes developers assigned to the review. The author field specifies the owner of the change who submits the review. The project field specifies details about the project and sub-project under which the author has submitted the review. The file path field specifies the file-locations modified in the change. The last reviewed date specifies the date when the review got completed.

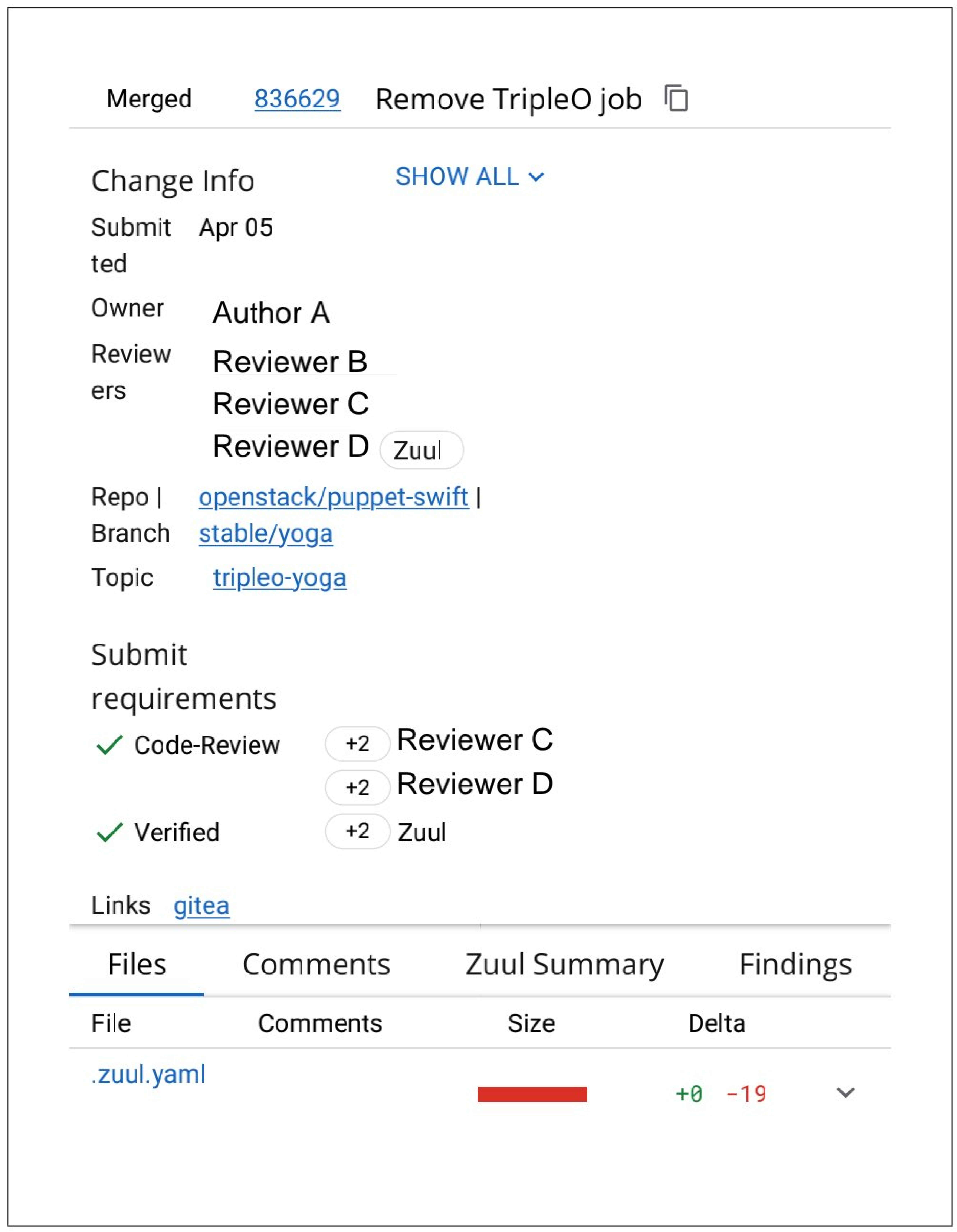


Figure 1.4: Code-Review 836629 - OpenStack project of Gerrit

Subject	Stein only: Remove tripleo job
Project/Subproject	openstack/puppet-swift
Author	Author A
Reviewers	Reviewer B, Reviewer D, Zuul
Files	.zuul.yaml

Figure 1.5: Code-Review 798228 - OpenStack project of Gerrit

Subject	Ussuri-only: Remove TripleO job
Project/Subproject	openstack/puppet-swift
Author	Author A
Reviewers	Reviewer B, Zuul
Files	.zuul.yaml

Figure 1.6: Code-Review 836814 - OpenStack project of Gerrit

Figure 1.4 shows a code-review 836629 from the Open-Stack project of Gerrit[13]. The *Author A* modified the code and submitted the code for review. The change removes the *"TripleO job"*. The reviewers *Reviewer* C and *Reviewer D* reviewed the change and approved it. Here, the *Zuul* is a open-source contiguous integration tool which checks the test cases and only merge changes if they pass the tests.

Figures 1.5 and 1.6 represent reviews 798228[14] and 836814[15] from the Open-Stack project of Gerrit. The change reviewed in review-798228 is to remove the *"TripleO job"* of *Stein* as stable/stein branch for all tripleo repositories was moved to end of life. The change in review 752343 is to remove the *"TripleO job"* of *Ussuri* as stable/ussuri branch for all tripleo repositories was moved to end of life. We notice that all of these changes were from the same subproject *"puppet-swift"* and these changes were proposed by the same author.

1.2.1 Observations and Implications

We make the following observations from the above three reviews:

1. The project/sub-project information can help recommending appropriate

[13]https://review.opendev.org/c/openstack/puppet-swift/+/836629
[14]https://review.opendev.org/c/openstack/puppet-swift/+/798228
[15]https://review.opendev.org/c/openstack/puppet-swift/+/836814

reviewers. Reviewer are more likely to review changes which are from the same projects/sub-projects. For example, change requests shown in Figures 1.4, 1.5 and 1.6 are from the same subproject: *"openstack/puppet-swift"*. These three reviews were assigned to the same reviewers.

2. The textual contents in the subject field give the abstract information about the change and are good measure to recommend suitable reviewers. For example, the textual contents of the subject in Figures 1.4, 1.5, and 1.6 specify that the changes are made to remove the *"TripleO job"*.

3. The review request from the same authors can be reviewed by similar reviewers as they know each other because of past collaborations. From Figures 1.4,1.5 and 1.6, we can see that the code review requests submitted by the *"Author A"* were assigned to same reviewers.

4. The files stored in same paths should be reviewed by same reviewers. From Figures 1.4,1.5 and 1.6, we can see that all the code review requests changes the file *".zuul.yaml"* and thus they were assigned to same reviewers.

1.3 Thesis Contribution

1.3.1 Literature Review

We have searched the literature and conducted the comparative review on the basis of the features used in commonly used reviewer recommendation approaches along with their technique, supported platform, source code and data-sets availability, metrics used. Detailed literature review over 8 such recommendation approaches is demonstrated in a tabular format.

1.3.2 Mining Code Review Repositories

We built two mining algorithms to mine and analyze the code-reviews from the most commonly used *OSS* platforms: Gerrit and GitHub. We mined a total of 30,648 code-reviews from our selected 20 projects.

1.3.3 Automating Reviewer Recommendation Process

We propose a hybrid approach, *CORMS* (Code Reviewer Recommendation in GitHub and Gerrit based on Machine Learning and Similarity Analysis), for rec-

ommending active reviewers in code review. The *CORMS* effectively uses following features of a code-review: 1) subject of the change, 2) file location, 3) author field, and 4) project/sub-project. The *CORMS* first builds a similarity model by computing similarities of the three features: file-paths, author information, project/sub-project of the current pull request separately with the same three features of all the pull requests historically studied by various developers. Then it builds a Support Vector Machine (SVM) model by analyzing text in the subject of the code-reviews. The *CORMS* then integrates both the SVM and similarity model to recommend the reviewers with improved performance.

1.3.4 Performance Evaluation

We evaluate the performance of our approach on 20 data-sets collected from both Gerrit and GitHub platforms. We mined and analyzed 30,648 code reviews from 2020 to 2021. The evaluation was done by computing the values of top-10, top-5, top-3, and top-1 accuracies, and Mean Reciprocal Rank (*MRR*) [5]. Next, we compare our experimental results with the *RevFinder* [40] and found that *CORMS* achieves average top-10, top-5, top-3, and top-1 accuracies, and Mean Reciprocal Rank (*MRR*) of 79.9%, 74.6%, 67.5%, 45.1% and 0.58 for the 20 projects, which improves the performance of *RevFinder* by 12.3%, 20.8%, 34.4%, 44.9% and 18.4%, respectively. The individual models of *CORMS* when combined with the *RevFinder*, also positively improves the *RevFinder* w.r.t the evaluation criteria and generated data-set.

1.3.5 Tool Support

A tool is developed based on the implementation of the proposed approach *CORMS* with taking all the required features as an input in form of JSON and predicting the active code-reviewers as output. It also enables the support to handle request for new projects and collects feedback from users on every predictions and uses it to demonstrate the effectiveness of *CORMS* when used in real scenario.

1.4 Organisation of the Thesis

The rest of the thesis is structured as follows. Section 2 provide a brief description of modern code review (MCR). Section 3 presents work related to our study. Section 4 discusses our approach that extends *RevFinder*. Section 5 reports and

discusses the results of our experiments. The section also discusses threats to validity and issues in mining data from GitHub/Gerrit. Section 6 demonstrate the working of our tool. Finally, Section 7 presents conclusions and highlight future directions.

CHAPTER 2
Code Review

This chapter describes the software code reviews, modern code review process, benefits and challenges of modern code review and factors influencing this process.

2.1 What is Code Review?

Code review refers to a manual assessment of a code that detects potential defects such as errors in both low level and high-level design and quality problems such as coding rule violations. These are the various code review techniques:

1. **Code Inspections**: Code inspections are one of the first processes for software code review. It includes planning, preparation, overview, code-inspection meetings, rework and follow-up [16]. The goal of software code inspections is to find errors during a synchronized inspection meeting, with reviewers and authors sitting together to check the code changes. From 2005, research on code inspection techniques are declined due to the increasing adaptability of asynchronous code review process. [34]

2. **Asynchronous Review**: In this, reviewers evaluate the code changes and communicate through the the channels such as email. When a change seems to high enough quality, reviewers allow commit. In this area, Kononenko et al. [21] find that review acceptance and response time are related to some social factors such as author experience and reviewer workload which were not available in software code inspection.

3. **Tool-based Review**: Several tool has been proposed to support the code review process. The workflow of most of the tools includes following steps:

   (a) Author submits change to the code-review tool

(b) Reviewers can view the difference of the proposed code change request with its previous version.

(c) Reviewers can start discussion on certain lines with the author and other reviewers.

(d) Author can submit the required changes to address the comments given by reviewers.

This cycle continues until the reviewers are satisfied or the review is abandoned.

Software code reviews help the development process in reducing overall costs and helps in knowledge transfer. The reviews conducted on the software code identify logical errors, coding rule violations, and also assisted with the automated tools. This review process is known as MCR (Modern Code Review) [3]. The MCR are nowadays used by both the organizations such as Microsoft [24] or Google [34], and by the open-source software (OSS) platforms such as GitHub[1], Gerrit[2] or review-board[3]. Rigby and Bird [32] examine several code review case studies from projects spanning various OSS platforms and organizations and found five common best practices of modern code review. These practices showed that modern code review 1) is a flexible and lightweight process; 2) happens early (before a change is committed), quickly, and frequently; 3) code change sizes are small; 4) generally involves two reviewers and 5) has changed from defect finding activity to a group problem-solving activity.

2.2 Modern Code Review Process

Figure 2.1 describes the working of the MCR process. MCR process has four different roles: author, developer, maintainer, commenter. Review planning consists of 3 steps. In the preparation step, the author prepares change metadata with a description providing extra details of the change. Then, reviewers are picked in the reviewer selection step. Finally, reviewers receive alerts to review the reviewer notification step. Code review consists of three main steps. In Code checking step, the reviewer individually performs code checking. They may also interact with the author and among themselves in the Reviewer Interaction step. In reviewer decision, reviewers decide on the change request which can be reworked, accept,

[1] https://github.com/
[2] https://www.gerritcodereview.com/
[3] https://www.reviewboard.org/

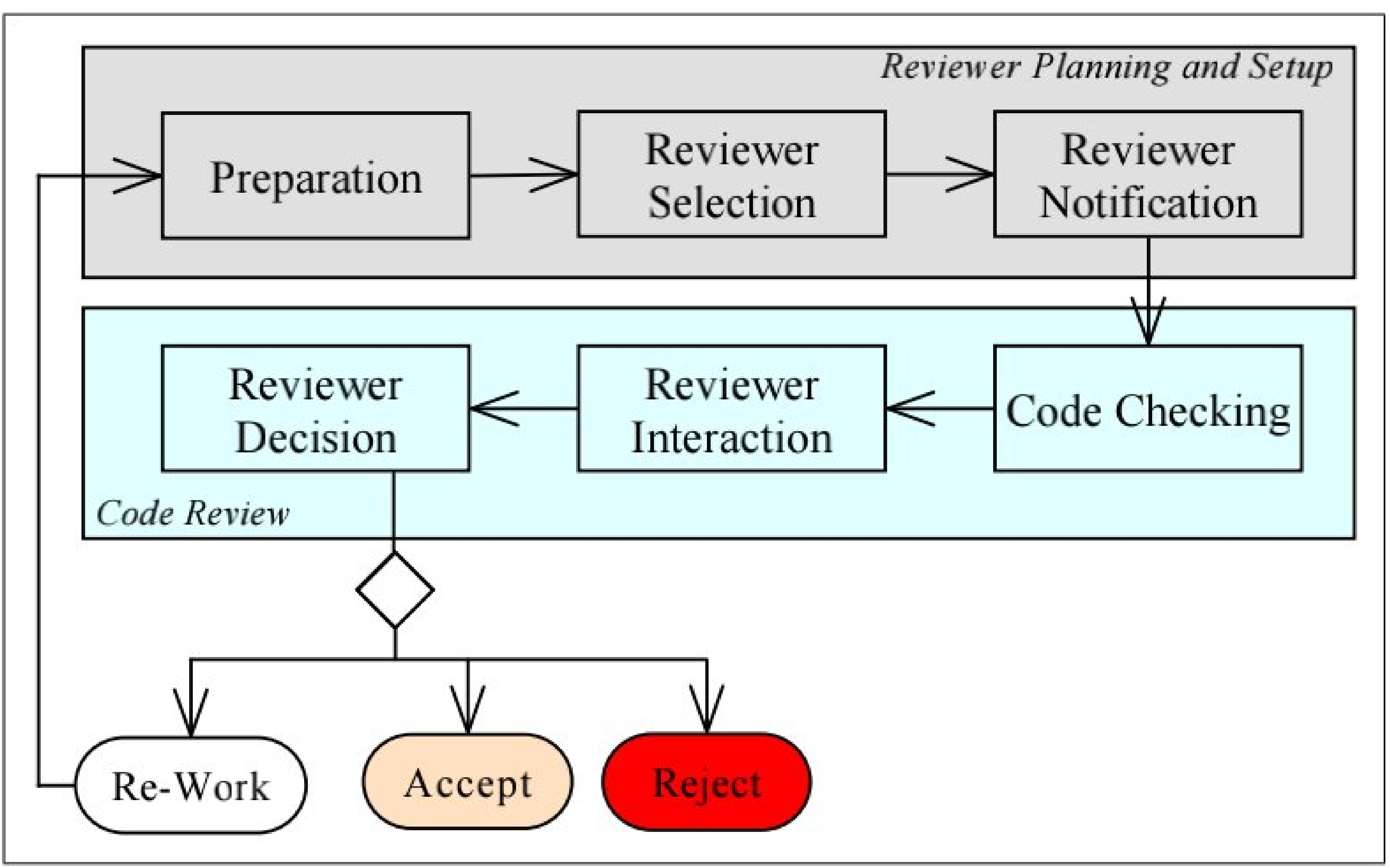


Figure 2.1: Working of Modern Code Review (MCR)

or reject. In recent years, MCR gained increasing popularity. Therefore, to understand the comprehensive perspective of MCR and the work that has been done till now, Nicole Davila et al. [12] presented the results of a systematic literature review on MCR, which includes 139 studies for drawing inferences.

Multiple approaches to support MCR have been proposed. Most of the approaches to support authors, focus mainly on the reviewer recommendation techniques which uses a review history of the reviewers and their interests to build recommendations. Approaches to support reviewers, focus on the code checking, review prioritization, review decision and feedback support. Most of them focused on the code checking to provide visualizations of code changes by highlighting changes to help understanding and the finding defects, which is a challenge for new reviewers. [12]

2.3 Benefits and Challenges

The benefits of code review are (1) finding defects in both low level and high-level design, (2) code improvements to handle coding conventions with no defect identification, (3) alternative solutions, (4) knowledge transfer which gives another developer insight about code, (5) team awareness and transparency with the code, and (6) sharing code ownership [3].

While challenges of code review include (1) understanding reason for the change, (2) familiarity with the code - it takes more time to review files for the reviewers who are not familiar with the code, (3) dealing with understanding needs - re-

viewers, in general, may try different paths such as sending emails or talk in person for asking clarifications, (4) lax review means directly approves the change request, (5) code size and poor quality, (6) delayed feedbacks, (7) difficulties in finding proper reviewers, (8) lack of developers' plan for knowledge sharing, (9) change request rejections, and (10) lack of available reviewing guidelines [3][17].

2.4 Factors Influencing the Modern Code Review

There are several factors influencing the modern code review which are important for code reviews and where tool support would be beneficial. Table 2.1 describes the factors influencing modern code review and their impact on code review.

2.4.1 Non Technical Factors

Non technical factors related to author include:

1. Author's familiarity with the code for a given project.
2. Author's overall development experience.

Non technical factors related to reviewer includes:

1. Reviewer's familiarity with the code for a given project.
2. Overall reviewing experience.
3. Reviewer's participation rate.

Other non technical factors include project's review workload which is the number of the code-review requests submitted in a certain period.

2.4.2 Technical Factors

Technical factors related to patch properties include:

1. Presence of poor programming practices that affect the code maintainability.
2. Code change location or the module it belongs to.
3. Number of files and directories in a code review request.
4. Change size, which is equal to the total number of inserted and deleted lines.

5. Type of change such as whether the change includes new code or it modifies the existing files.

Technical factors related to historical patch properties include:

1. Number of comments posted in the reviews of earlier code reviews that modifies the same files. These review comment involves the discussion topics related to code understanding, improvement, alternate solutions and social interactions.

2. Prior defects that impact the same files.

3. Overall feedback or review delay of the reviewers.

Other technical factors include:

1. Request description length which is the Number of words an author uses to describe a change in a code review request.

2. Reviewers notification that includes broadcast notification which is visible to all developers and unicast notifications which is visible to a specific group of developers.

3. Classification of changes based on priorities such as high-level or low-level bug fixing.

Table 2.1: Factors Influencing the Modern Code Review

Group	Factor	Description	Influence on code review
Non-Technical factors			
Author	**Code familiarity**	Author's familiarity with the code for a given project.	When author's familiarity with the code decreases, the number of comments and the number of active reviewers increases. In other words, new authors receive more attention from invited reviewers [25][39][23]. When the author's familiarity with the code increases, the review duration decreases [7][9][25].
	Development experience	Author's overall development experience.	When the author's experience with the code increases, the review duration decreases [7][9][25].
Reviewer	**Code familiarity**	Reviewer's familiarity with the code for a given project.	When the reviewer's familiarity with the code increases, reviewer may express confusion in comments [15][14]. The reviewer's code familiarity influences the quality of the feedback [29][10].
	Experience	Overall reviewing experience.	When the reviewer's experience increases, the review duration decreases [7]. When the reviewer's experience increases, the post-release defects decreases [22].
	Participation rate	Reviewer's participation rate.	Participation of reviewers in code review influences a product quality. Yang et. al [44] found that in OSS projects, an average of one or two reviewers per request responded to a review request. Reviewer who have more experience and have higher review participation rate are more likely to respond new invitations [33].
Other	**Project's review workload**	The number of the code review requests submitted in a certain period.	Authors usually prefer to invite reviewers who have more experience, leads to increase the burden upon them [26]. Yang et. al. [44] found that inviting inactive reviewers can speed up the code review process and it reduces the burden upon active reviewers.
Technical factors			
Patch properties	**Code maintainability**	Practices that affect the code maintainability.	Presence of poor programming practices such as indentation issues have a negative impact on the performance of the reviewers [1].
	Change location	Module it belongs to.	More change locations negatively affects the review team size [33].
	Scatteredness	Number of files in a code review request.	Changes that includes more modified files and lines of code increases the number of code review iterations [8].
	Change size	Equal to the total number of inserted and deleted lines.	A large change size negatively influences the review duration [39]. A small change size increases probability of attracting more reviewers [13]. The more the lines changed in the patch, the more likely the patch is discussed [39].
	Type of change	New code or modifications in the existing files.	Modified files are more vulnerable as compared to new files [9].
Historical	**Feedback**	Number of comments posted in the reviews	The review comment involves the discussions related to code improvement, understanding, social interaction and alternate solutions [35][4]. Rajshakhar Paul et al. [27] found that female express less sentiments in comments than males. Jing Jiang et al. [20] in their study found an average of seven comments for reopened requests and two comments for non-reopened review requests.
	Prior defects	Prior defects that impact the same files.	Files having prior defects, tend to undergo reviews that have more revisions and shorter discussions without any reviewer feedback as compared to normal files [38].
	Review delay	Overall feedback delay of the reviewers	A large change size and having more teams involved might have a negative influence on the review duration [39]. Reviewer assignment problems negatively impact the review delay [40].
Other	**Request description length**	Number of words author uses to describe a change in a request.	When the size of change description increases, the probability of attracting more reviewers increases [13].
	Reviewers notification	Reviewers notifications include unicast broadcast notifications.	Broadcast notifications are visible to all developers while unicast notifications are visible to a specific group of developers. Code reviews that uses broadcast method receive less iteration than the reviews that uses unicast method [2].
	Task classification	Classification of changes based on priorities such as high-level or low-level bug-fixing.	There is a need of review prioritization as reviewers might be invited to many reviews. Higher priority should be given to high quality tasks such as code changes that impact the critical project deliverables which cover a brad set of products [41].

CHAPTER 3

Literature Review

Recommending reviewers is the most common support in MCR. Mostly, recommendation techniques use a review history of the reviewers and their interests to build recommendations. This chapter provides an overview of algorithms developed for the recommendation of software code reviewers. Section 3.1 describes several traditional approaches. Section 3.2 describes the approaches based on reviewer expertise. Section 3.3 describes the approaches based on social relations. Section 3.4 describes the several hybrid approaches and Section 3.5 gives the comparative summary of these algorithms with respect to various parameters.

3.1 Traditional Approaches

Traditional recommendation approaches process historical reviews and use imperative algorithms to find the most appropriate code reviewers.

3.1.1 ReviewBot

In *ReviewBot* [6], the author analysed the code review process over Review Board. This algorithm is based on the line change history (LCH). As it will recommend same reviewers who has previously worked on the same lines. This algorithm has two phases: (1) Computing LCH: The *ReviewBot* algorithm iterates over all lines modified in the pull request and computes the LCH for all of them. The LHC over file difference Fd can be computed by

$$LCH_{fd_{rq}}(l) = (rq_1, rq_2, \ldots, rq_n) \tag{3.1}$$

where rq1,rq2 are the review request which contains Fd affecting line L. LCH of deleted and updated lines in Fd can be computed easily. For inserted lines, as there are no change requests in past which affected those lines, in this case authors chose the nearest line. It then assigns points to review requests related

to the lines. The more recent the review was, the more points it is assigned. (2) Reviewer Ranking: Previous review requests were assigned points in the first step which is then propagated to appropriate reviewers as points in the second step. The result of this step is a list of reviewer candidates sorted by their points.

The *ReviewBot* algorithm has several problems. One of the problems is with newly created files which don't have any line change history and thus cannot be correctly processed using this approach. Another problem is that most of the lines in average project are only changed once. The accuracy of results returned by the *ReviewBot* algorithm is therefore limited.

3.1.2 RevFinder

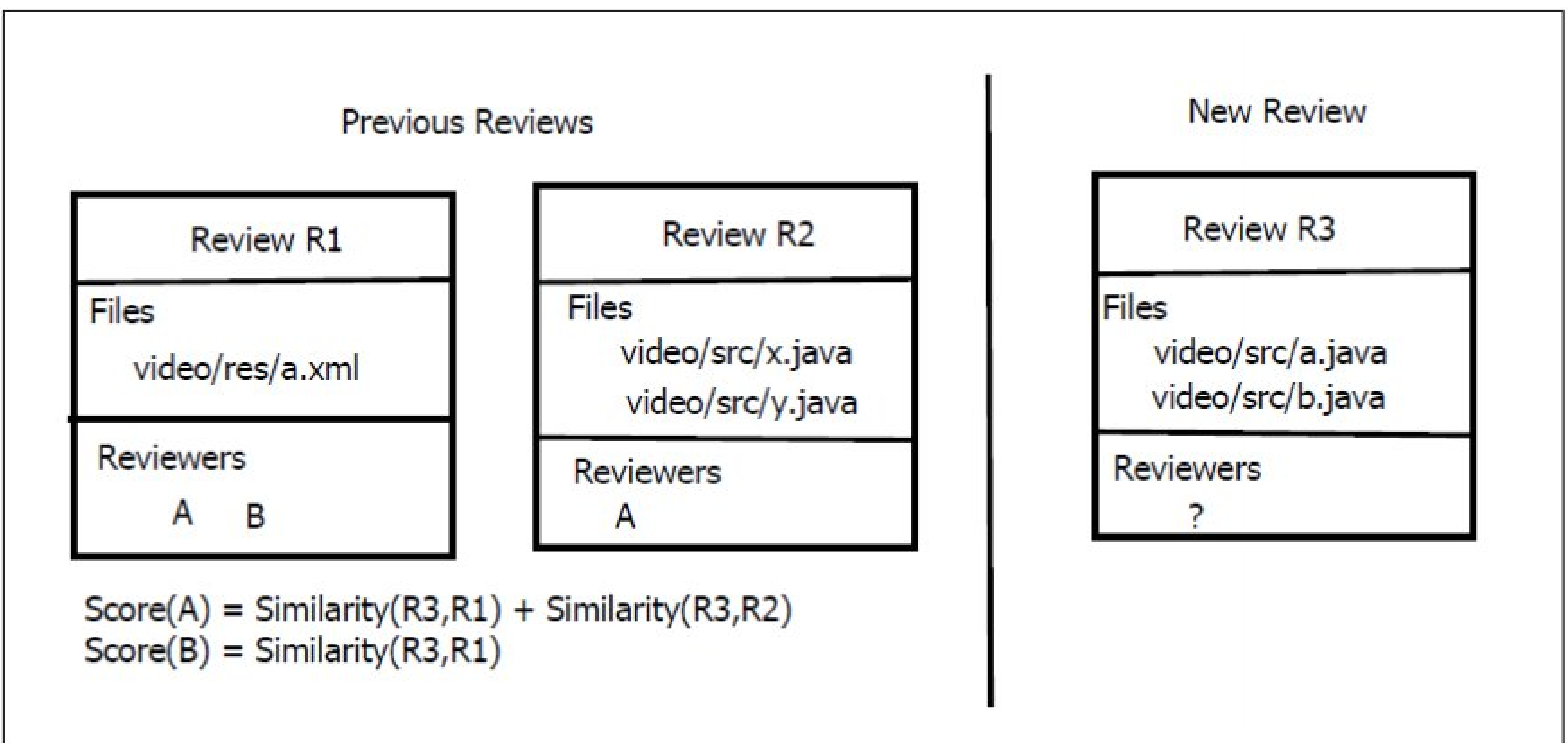


Figure 3.1: Working of Revfinder

In *RevFinder* [40], the author analysed the code review process of Gerrit. It is based on the file path information as files stored in same paths should be reviewed by same reviewers. Figure 3.1 describes the working of *RevFinder*. It has two phases. First phase compares previously reviewed file paths with file paths included in a new change request by using the 4 string comparison techniques (LCSuffix, LCPrefix, LCSubstr, LCSubseq) to examine the file paths' similarity between new and historical file paths. Filepathsimilarities of two file f1 and f2 can be given by:

$$filePathSimilarity\,(f1, f_2) = \frac{\text{StringComparison }\;(f_1, f_2)}{\max\left(\text{Length}\,(f_1)\,,\text{Length}\,(f_2)\right)} \tag{3.2}$$

(2) Borda count combination: It combines the results of these string comparison techniques and returns sorted list of reviewers with their scores. This approach was tested on more than 42000 reviews of four open source projects and

it was 4 times more accurate than the *ReviewBot* algorithm. *RevFinder* was able to correctly recommend 79% reviews considering a top-10 accuracy.

This algorithm has several problems though. It does not consider retired code reviewers. It is also not able to recommend code reviewers for new files where no file path similarity is found. Also there are threats to external validity as their results are only for the 4 data-sets: OpenStack, Android, LibreOffice and Qt.

3.2 Approaches based on Reviewer Expertise

Approaches based on reviewer expertise use historical data to measure expertise of reviewers and recommends suitable code reviewers.

3.2.1 CORRECT

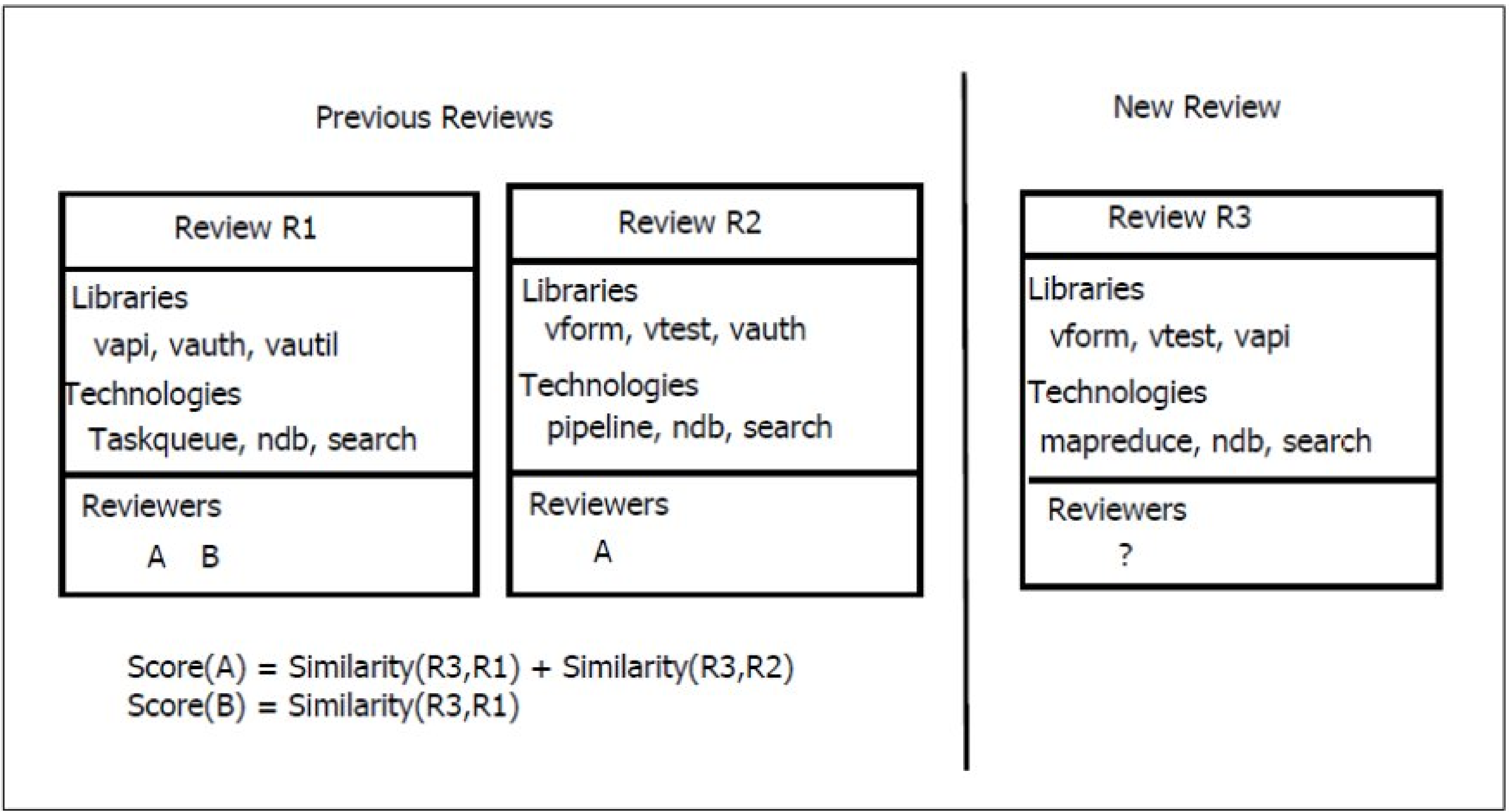


Figure 3.2: Working of CORRECT

In *CORRECT* (Code Reviewer Recommendation based on Cross-project and Technology experience) [28], author analysed the code review process of Github. It is based on the reviewers' expertise as it will recommend the same reviewers who worked on same external libraries or technologies.

$$R_i = L_i \mid L_i \epsilon L_{\text{ext}} \cup T_i \mid T_i \in T_{\text{special}} \tag{3.3}$$

As shown in equation, any change request Ri, can be viewed as a combination of specialized technologies (Tspecial) and tokens for external libraries (Lext) used

in the change request. Authors used Cosine Similarity to estimate similarity degree between current pull request and past pull request. Figure 3.2 describes the working of the *CORRECT*. This approach iterates over all files of a newly created change request and analyzes the imported libraries of these files. The developers who have the most experience with the attached libraries are considered to be the most relevant code reviewer candidates. The approach was tested on several private projects and was compared against the *RevFinder* algorithm. *CORRECT* outperformed the *RevFinder* algorithm on these projects having top-5 recommendation more than 11% better and *MRR* value by 0.02 better than *RevFinder*.

This approach was built only for python,java and ruby. Thus its not generalized to support all languages, ie. the syntax of import statement may differ language by language. Another problem of this approach is with projects which don't have many external dependencies and thus developers' expertise with external libraries would hardly be helpful.

3.2.2 RevRec

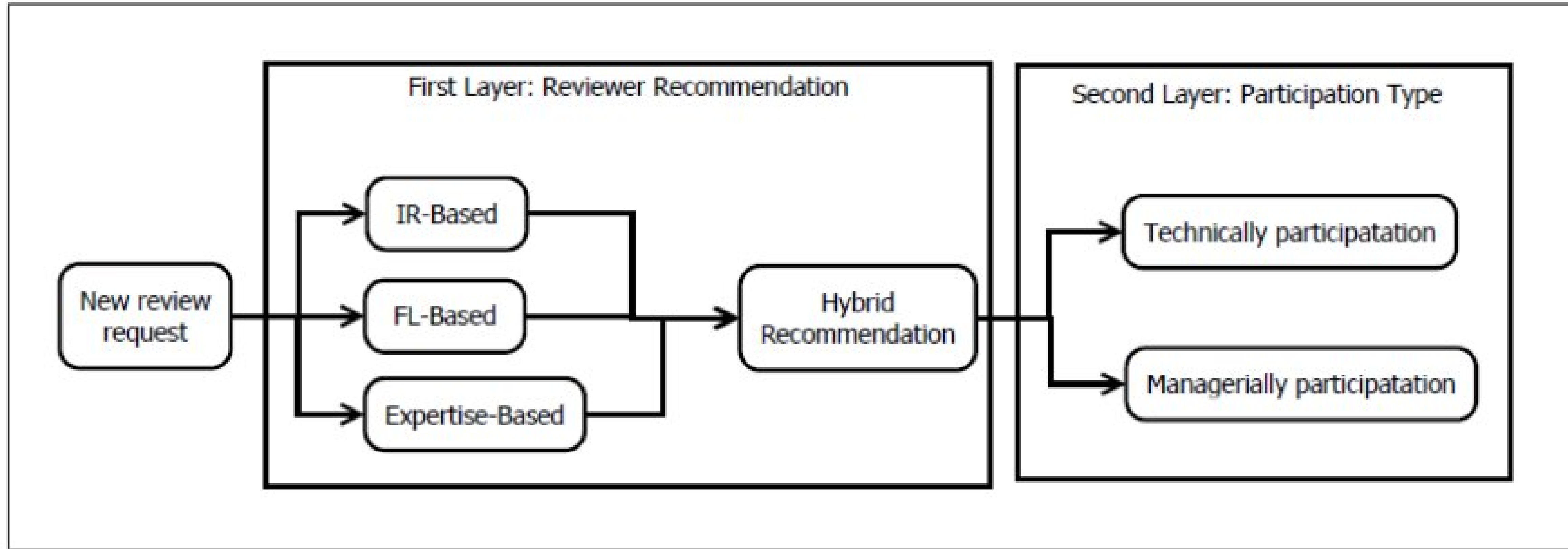


Figure 3.3: Working of RevRec

Yang cheng et al. [45] proposed two layer reviewer recommendation approach *RevRec*. In the first layer, it recommends appropriate reviewers to review the code-review requests based on a hybrid recommendation method combining the similarity, expertise, and IR methods. After getting the recommendation results from the first layer, in the second layer, it specifies the participation type from technical and managerial perspectives, whether the recommended reviewer will participate managerially or technically in the reviewing process. As some reviewers will identify the defects in the code change while other reviewers will just comment below the review description. The authors tested their results on the two projects of GitHub.

Figure 3.3 describes the working of *RevRec*. First layer includes hybrid recommendation consist of three methods: 1) IR-Based method which generates the technical terms of each code review requests based on its title and description, generates term vectors of each term and then calculates relationship between these code review requests using cosine similarities. 2) FL-Based method for recommending reviewers based on their similarities between file-paths computed using string comparison techniques. 3) Expertise-Based method focusing on the memory strength of reviewers by considering the lines of code change and correlation time. It is based on the fact that, with the time, the reviewer's expertise becomes lower towards that piece of code. First layer gives top-10 reviewers as a prediction result. For each of these reviewers, their participation type will be predicted in second layer based on their experience for technical or managerial review.

Their results showed that for the second layer, they achieved 72.8% accuracy for the ruby project and 58.6% accuracy for the angular project. Although it has some drawbacks. There may be the cases where reviewers just managerially participate in the code review request, but mention the code change in the comment. This should be categorised into technical participation instead of managerially participation. Also, participation types can be further classified into encouraging authors, asking questions or giving suggestions, testing code and changing the code.

3.3 Approaches based on Social Relations

Approaches based on social relations analyse the social collaboration between developers and reviewers.

3.3.1 Comment-Network

Yue yu et al. [46] proposed a *Comment-Network (CN)* to capture the common interests in social activities among developers and recommend appropriate reviewers based on it. The authors tested their results on 84 projects of GitHub. They defined comment network as a weighted directed graph G, where the vertices indicates set of developers and edges indicates the set of relations between developers. Their hypothesis is based on 1) the freshness of comments and 2) comments present in multiple reviews are more important than those in one review. Their approach works as follows: 1) They build the collaboration structure for a given project using developers and reviewers information 2) They introduce time-sensitive factor

to ensure the freshness of comments. 3) The introduce decay factor to differentiate between the comments submitted to multiple reviews against a single review. 4) They balances these two factors to measure the developer-reviewer relationship. Their results showed that the CN-based approach combined with traditional recommendation approaches can achieve significant improvements when compared with pure traditional algorithms. Though, there are threats to external validity as their results are based on GitHub only. Also it faces workload issue as a there were cases when a reviewer joined many discussion of multiple reviews which results in recommendation of same reviewer as top-1.

3.4 Hybrid Approaches

Hybrid Approaches considers multiple factors for a code reviewer recommendation.

3.4.1 CoreDevRec

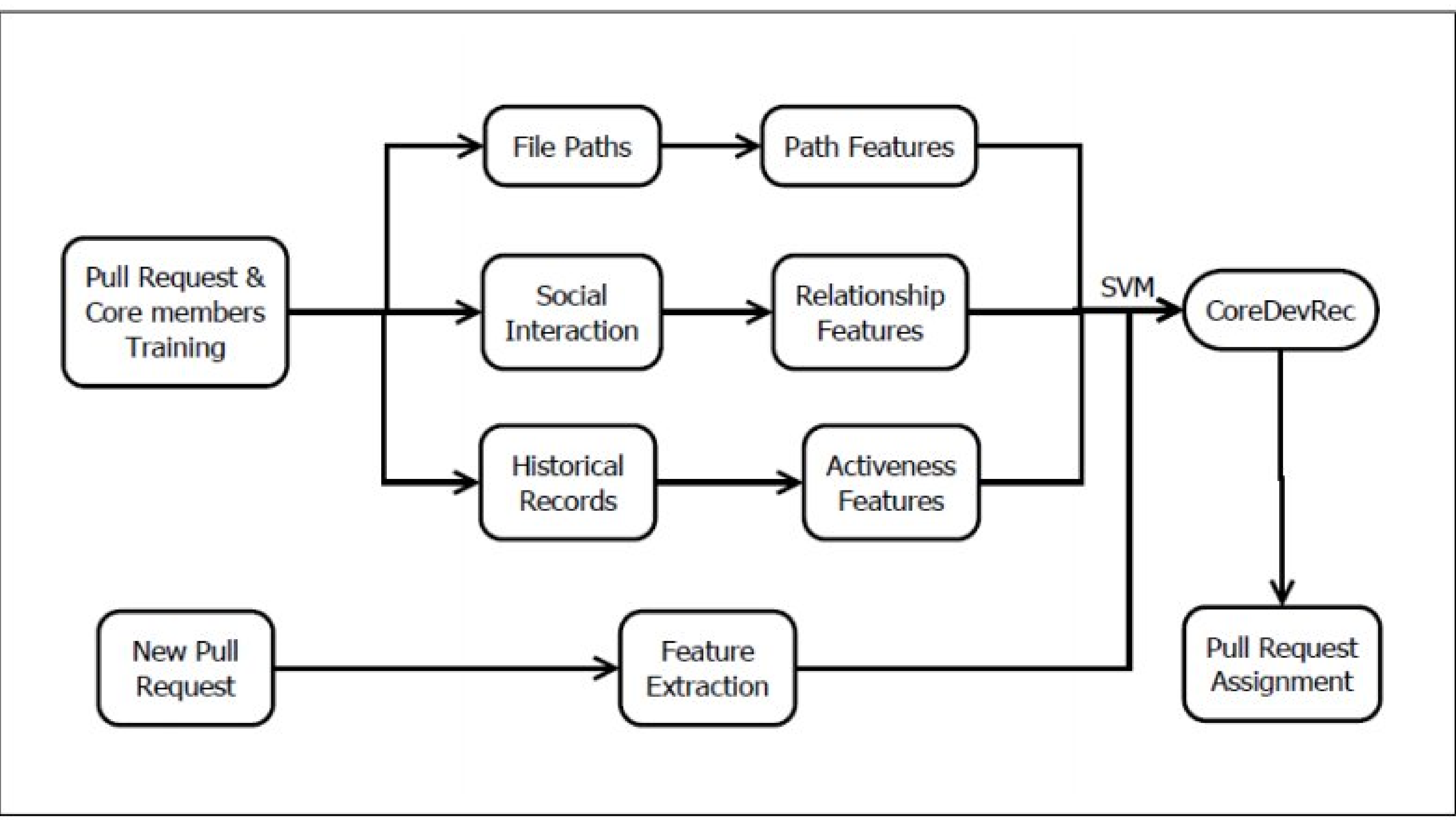


Figure 3.4: Working of CodeDevRec

In *CoreDevRec* (Automatic Core Member Recommendation for Contribution Evaluation) [19] the author analysed the code review process of Github. It is an approach based on Machine Learning and uses Support Vector Machine algorithm. Its overall process can be described by Figure 3.4. *CoreDevRec* algorithm has two phases. The first phase builds a prediction model from past change requests using three feature, while the second phase uses Machine Learning techniques to recommend code reviewers. Several Machine Learning techniques were

tried for the Prediction Phase. Support Vector Machine method had the best result and was hence chosen for *CoreDevRec*.

Features used for predictions:

1. Path Features It includes File path where Authors used Support Vector Machine algorithm with TFIDF unlike the four string comparison techniques used in the *RevFinder* algorithm.

2. Relationship Features: extracted from social relationship information between GitHub developer. It includes Follower Relation, Following Relation, Prior Evaluation, Recent Evaluation

3. Activeness Features: to recommend active and available reviewers. It includes Total Pull, Evaluate Pulls, Recent Pulls, Evaluate Time, Latest Time, First Time.

CoreDevRec algorithm was tested on five open source GitHub projects. It was compared against *RevFinder* and it outperformed *RevFinder* in all these projects with *MRR* value better by 0.21 on average.

The problems with this approach include (1) follower relation and following relation are not directly available on some projects (Apache/Mozilla). (2) In other OSS platform, social Data is not directly available and needed to be gathered using help from mining mails and other methods. (3) These social relations are not gathered at the time of every change request (4) It is not known if these results can be acceptable in another OSS platforms or not as these results are based on only GitHub (5) did not consider quality of results - whether recommended reviewers make correct decision of change request or not.

3.4.2 TIE

Xin Xia et al. [42] proposed a *TIE* that extends the state-of-the-art *RevFinder* by building the hybrid approach including the commit message as a new feature and using a Naive Bayes Classifier. The authors tested their results on the four projects of Gerrit. Figure 3.5 describes the working of *TIE*. Here, historical reviews are used to construct *TIECOMPOSER*, which is then used to recommend a set of reviewers for a new review request. They extended the *RevFinder* by including the commit messages and time features and also their file-path similarity checking technique differs from the one used in *RevFinder*. This approach was tested on same data-sets mentioned in *RevFinder*[40]. It improves the state-of-the-art *RevFinder* by 37%, 8%, 23% and 61% in terms of *MRR* and top-10,5,1 accuracies.

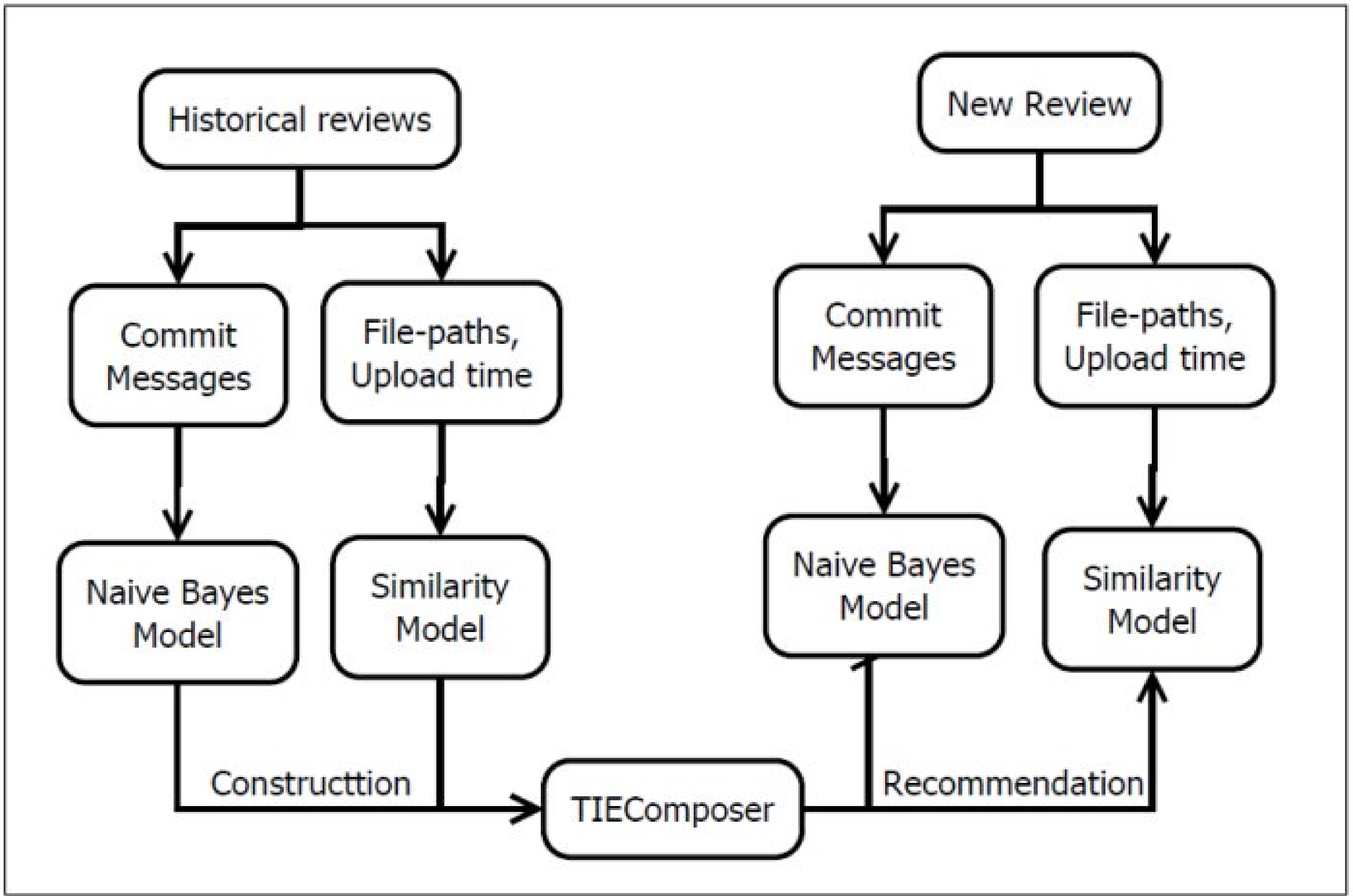


Figure 3.5: Working of TIE

Though it outperforms the *RevFinder*, there are threats to external validity as their results are only tested for the 4 data-sets: OpenStack, Android, LibreOffice and Qt.

3.4.3 WhoReview

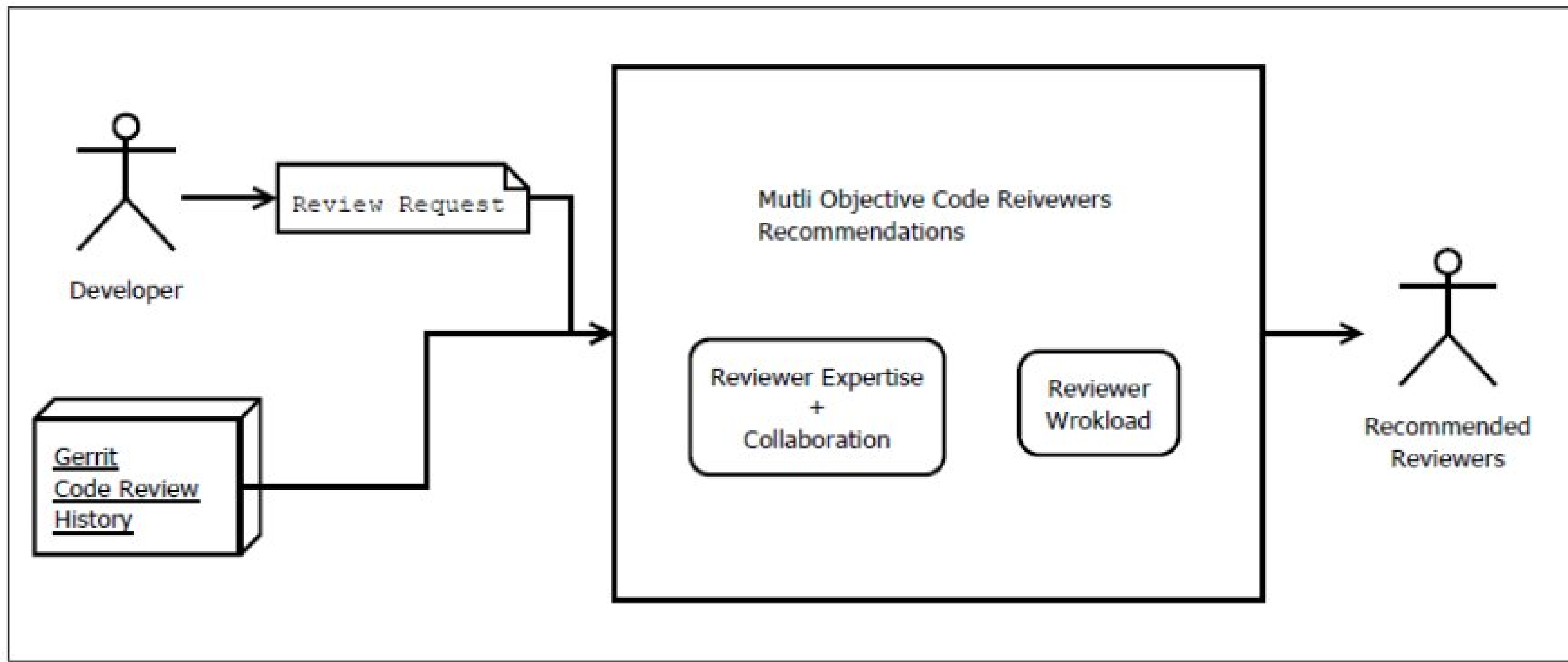


Figure 3.6: Working of WhoReview

WhoReview [11] is an approach based on Machine Learning. Here, the author analysed the code review process of Gerrit. They used the IBEA - Indicator-Based Evolutionary Algorithm to find the best set of reviewers who are more collaborative as well as experienced and has less workload. IBEA is a multi objective search based algorithm. Here, we have 2 indicators: 1) maximizing the reviewer collaboration and expertise 2)minimizing reviewer workload. The main reason

Table 3.1: Technique, Platform, Source Code, Data-sets, Metrics Used

Existing Approaches	Year	OSS Platform	Technique Used	Tested Projects	Availability of Data	Metrics			
						Top - k	MRR	Precision	Recall
ReviewBot	2013	ReviewBoard	Line Change History	1	Yes	Yes	No	No	No
RevFinder	2015	Gerrit	String Comparison	4	No	Yes	Yes	No	No
TIE	2015	Gerrit	Naive Bayes	4	No	Yes	Yes	No	No
Correct	2016	GitHub	Cosine Similarity	10	No	Yes	Yes	Yes	Yes
CoreDevRec	2015	GitHub	Machine Learning	5	No	Yes	Yes	No	No
Comment Network	2017	GitHub	Comment Network	84	No	Yes	Yes	Yes	Yes
RevRec	2018	GitHub	Cosine Similarity	2	No	Yes	Yes	Yes	Yes
WhoReview	2020	Gerrit	Evolutionary Search	4	No	Yes	Yes	Yes	Yes

Table 3.2: Features used in most common Recommendation Algorithms

	Existing Approaches								
Feature	**ReviewBot**	**RevFinder**	**TIE**	**CORRECT**	**CoreDevRec**	**Comm. Net**	**RevRec**	**WhoReview**	**Total**
File Path	No	**Yes**	**Yes**	No	**Yes**	**Yes**	**Yes**	No	5
Social Interaction	No	No	No	No	**Yes**	**Yes**	No	**Yes**	3
Line Change History	**Yes**	No	No	No	No	No	**Yes**	No	2
Reviewer Activeness	No	No	**Yes**	No	**Yes**	No	**Yes**	No	3
Subject	No	No	No	No	No	No	**Yes**	No	1
Libraries	No	No	No	**Yes**	No	No	No	No	1
Commit Messages	No	No	**Yes**	No	No	No	No	No	1
Author	No	No	No	No	No	No	No	No	0
Reviewer Workload	No	No	No	No	No	No	No	**Yes**	1
Sub-Project	No	No	No	No	No	No	No	No	0
Total	1	1	3	1	3	2	4	2	

to consider workload with the expertise is that usually the most experienced reviewer is recommended by most of the tools, which significantly increases her/his workload. Tools should consider one's workload too.

The working of *WhoReview* is given in figure 3.6. It has 3 models (1) reviewers expertise model - based on recent comment on same file. It is defined by review frequency(no of comments) and recency(freshness of comments), (2) reviewers collaboration model - based on number of interactions between developer and reviewer and (3) reviewers workload model - which is based on the size of change and average time spent on the review. Then maximization of the reviewer expertise and collaboration was done with minimizing the reviewer workload. Finally reviewer ranking was done as reviewer who appears more in near-optimal solutions has more points.

WhoReview was tested on 4 oss projects- OpenStack, Android, QT, LibreOffice and results showed that it outperformed *RevFinder* by an average recall of 77% and precision of 68%. Problems include 1) they didn't check over/under estimation in expertise and workload model 2) Used very costly, evolutionary search.

3.5 Summary

We summarized the well-known algorithms such as *ReviewBot* [6], *RevFinder* [40], *TIE* [42], *Correct* [28], *CoreDevRec* [19], *Comment-Network* [46], *RevRec* [45], *WhoReview* [11]. Table 3.1 represents and compares the technique, platform, source code, data-sets, metrics used. Table 3.2 represents features used in commonly used recommendation algorithms such as File-Path, Social Interaction, Line Change History, Reviewer Activeness, Subject of the change, External Libraries/Technologies, Commit Messages, Author, Reviewer Workload and Project/Sub-project information. Although there are multiple approaches available, Vladimir Kovalenko et al. [24], showed that there is a need for more user-centric approaches in reviewer recommendation.

CHAPTER 4

Code Reviewer Recommendation: Proposed Approach

This chapter describes Gerrit and GitHub based proposed hybrid code reviewer recommendation approach *CORMS* for a modern code review. Figure 4.1 shows

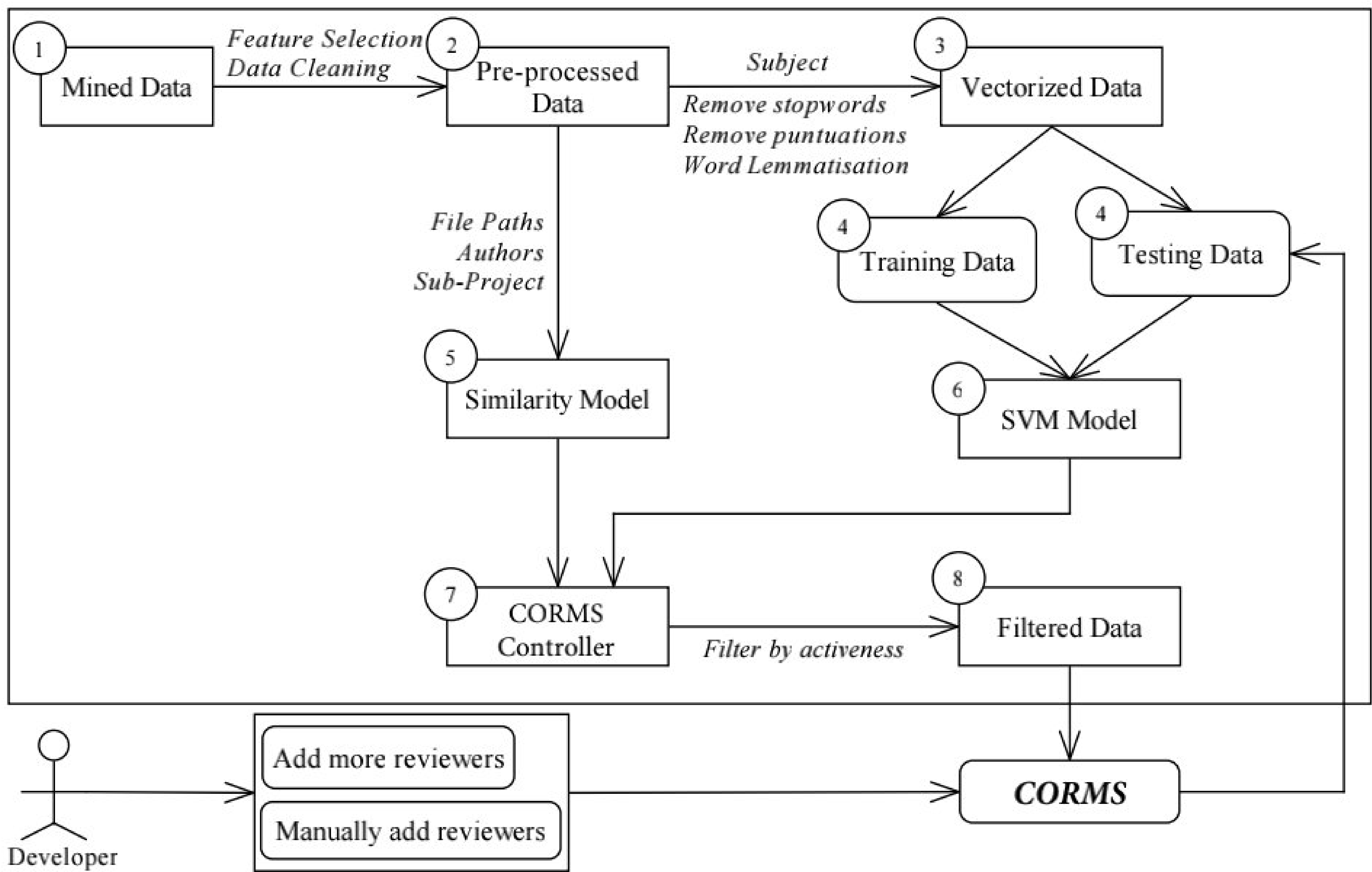


Figure 4.1: Proposed Hybrid Approach: *CORMS*

our hybrid proposed approach – *CORMS*. It includes the following steps:

1. **Data Mining**: To mine the required gerrit and github repositories of a given project.

2. **Data Pre-processing**: To process to remove the duplicate data and outliers.

3. **Natural Language Processing and Vector Transformation**: To process the

textual content present in subject field and to convert it into a numerical vectors.

4. **Data Splitting**: To split the data into training and testing. We used the terms closed reviews to describe the train data and new reviews to describe the test data.

5. **Similarity Model**: To compute similarities of the file-paths, author information, project/sub-project of the current pull request separately with the file-paths, author information, project/sub-project of all the pull requests historically studied by various developers.

6. **Support Vector Machine Model**: To predict the reviewer based on the tfidf vectorized form of subject field.

7. **CORMS Controller**: To combine and normalize the results of both similarity and Support Vector Machine model and give the top-10 reviewer recommendations by filtering out the retired reviewers.

Each of these steps are briefly explained in this chapter.

4.1 Data Mining

As shown in figure 4.1, the first step includes the data gathered from mining Gerrit or GitHub repositories. We created two different mining algorithms: one for mining Gerrit reviews, and another for mining GitHub reviews. We selected Python to implement these algorithms. We mined a total of 30,648 reviews from both Gerrit and GitHub in which 27,157 reviews are mined from 10 Gerrit projects while 3,491 reviews are mined from 10 GitHub projects.

4.1.1 Mining reviews from Gerrit

We mined recent code reviews of very popular projects from Gerrit using Python script. Our mining process for Gerrit is described in figure 4.2. Our script generated threads to mine parallel data faster compared to sequential mining. Each thread then built URL pattern required for mining and downloaded response in JSON. Though some essential extra parameters need to be specified explicitly - project, thread count, status, after and before date. In each thread change details URL[1] will be created and then thread will query this URL to download

[1]https://review.openstack.org/changes/?q=after:"2015-01-01"+before:"2022-01-01"&o=all_revisionss&o=all_files&o=messages&o=detailed_labels& S=1500& n=100

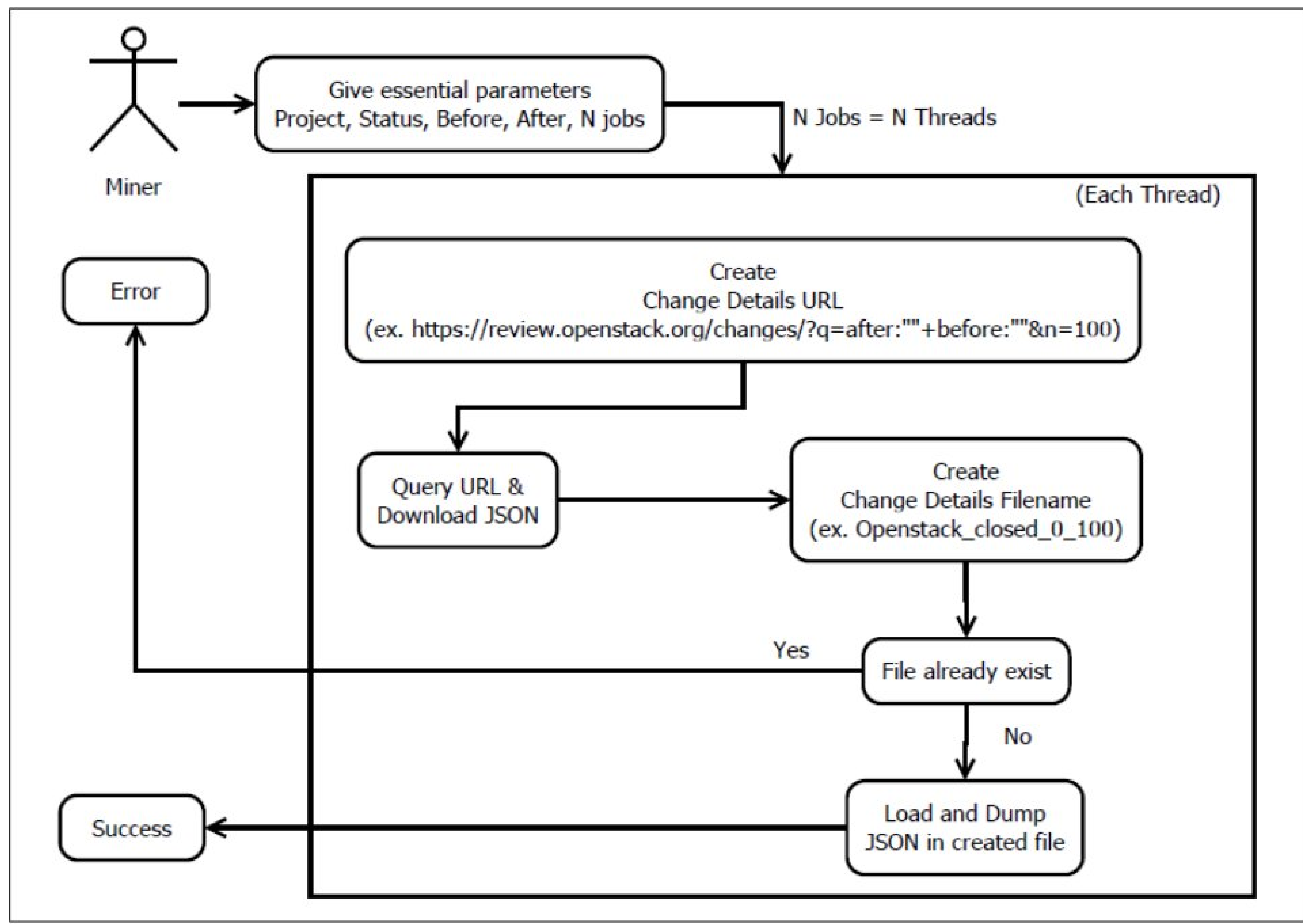


Figure 4.2: Our Mining Workflow for Gerrit

JSON data. For storing data, appropriate filename will be created (ex. openstack_closed_0_100). Further, filename will be checked whether similar file exist or not, if yes, it will give error otherwise it will load and dump JSON data into that file for storing it in appropriate format.

4.1.2 Mining reviews from GitHub

For mining the GitHub reviews we used GitHub Search API[2]. The daily API call limit is 60 for unauthenticated requests, which is not sufficient for mining large data-sets. Thus we generated Personal Access Tokens through which we can access access the GitHub API and can extend this limit to 5000. Our mining algorithm for GitHub is shown in figure 4.3. In a first step, we are making an API call to list all the repositories of a particular selected project. Then for each repositories we are making API call for mining closed and open issues related to that repositories. Then for each of these issues, we are making an API call for mining the pull requests related to that issue. Then from the data collected from these pull requests we further check that if any reviewers are assigned to pull requests or not. And if assigned we make a final API call to mine all commits related to

[2]https://docs.github.com/en/rest/reference/search

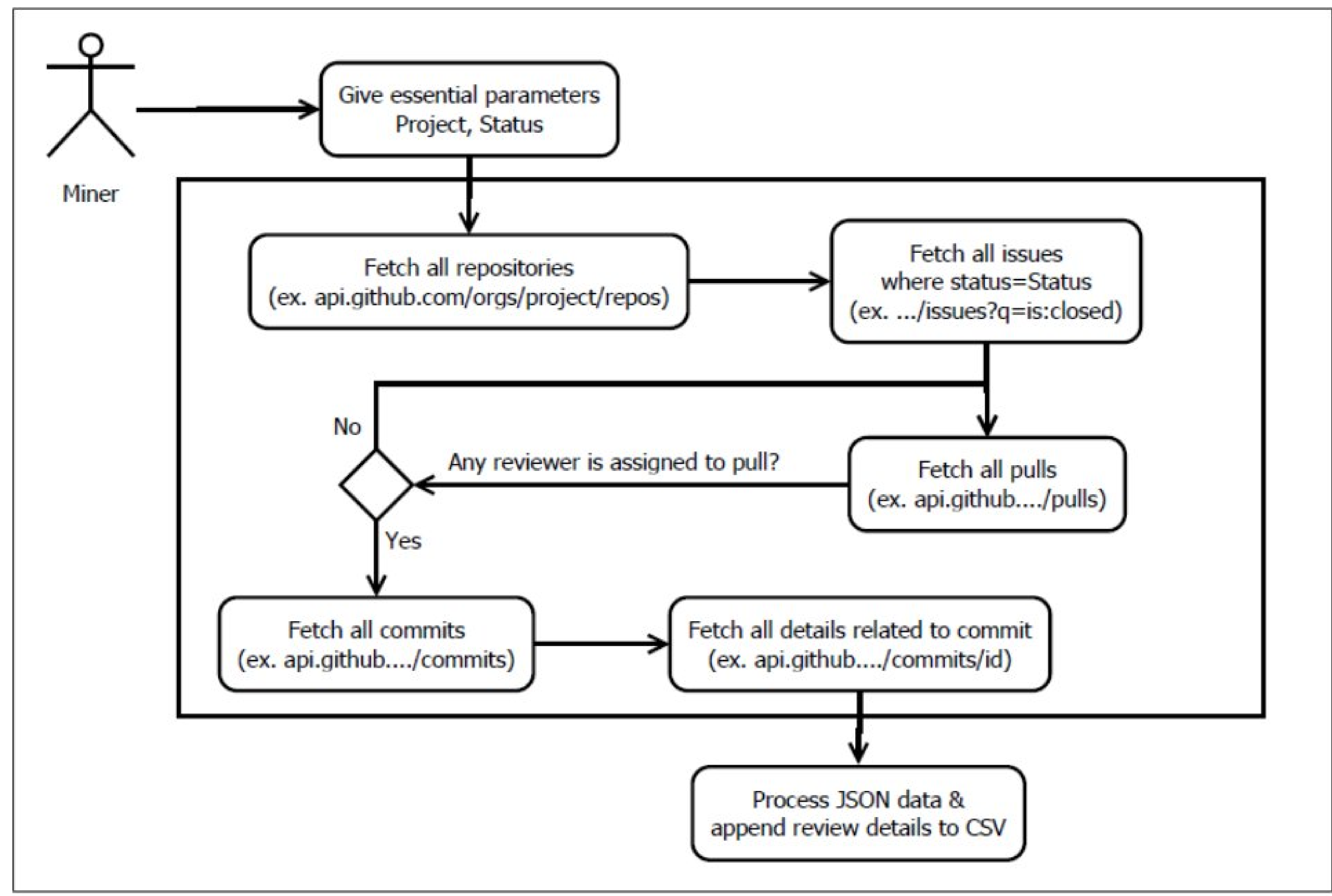


Figure 4.3: Our Mining Workflow for GitHub

this pull request. This will return the details about the list of commits related to this pull requests However, not about the files changed during this pull request. So to get this detail, we also need to make another API call in which particular commit id is given in a API call which we got from the results of earlier API call. So finally we will get details about the files that are changed during the commit which is the main feature of *RevFinder* and other details such as total updated/inserted/deleted lines. We then finally merge important details which we got from mining project, repository, issue, pull requests, commits and store or append it in a csv file.

4.2 Data Pre-processing

As shown in Figure 4.1, the second step includes the pre-processed data. From the mined data, we extracted the important features: 1) File-Path info 2) All Reviewer details 3) Subject of the change, 4) Project/Sub-project Information, 5) Last reviewed date, 6) Author details, 7) Change size which is equivalent to inserted lines + deleted lines, 8) Final reviewers details who voted +2 or -2 by making an important decision.

We then removed the reviews in which change-size was equal to zero. We

have also removed reviews that had status="open" or "new" and had duplicate values.

4.3 Natural Language Processing and Vector Transformation

As shown in Figure 4.1, the third step includes the natural language processing and vector transformation of the textual content.

4.3.1 Natural Language Processing

The subject of the change contains the textual contents and we can not pass this human language to our models. Thus in this step we process this human language.

1. **Lowered case**: Our first pre-processing step includes the lowering of the textual content.

2. **Removal of Punctuation**: We removed all punctuation from the string using the string library[3]. For example, for the string "(There is a tree, near the river!)", after removing all punctuation, we can get "There is a tree near the river".

3. **Removal of StopWords**: We removed all stop-words from the subject of the change using gensim library[4]. For example, for the same string, after removing stop-words we can get 'there', 'tree, 'near', 'river'.

4. **String Lemmatization**: We used string lemmatization to reduce inflectional and derivationally related forms of a word to a common base form. For instance *"am, are"* falls in the same class *"b"* while *"car, cars, car's, cars"'* falls in the same class *"car"*.

5. **Removal of Common Words**: Then we removed some most common words from the subject of the change. These words are: *"Insert"*, *"Update"*, *"Delete"*, *"Add"*, etc. So that in the end, we can have only meaningful words for which we can train and test various machine learning models.

[3]https://docs.python.org/3/library/string.html
[4]https://pypi.org/project/gensim

4.3.2 TFIDF Vectorization

We used the TFIDF vectorizer to transform a string into numeric vectors. It contains the following steps:

1. **Tokenization**: The first step to implement TF-IDF, is tokenization where a string is tokenized into a bag of words.

2. **Find TF-IDF Values**: The TF value refers to term frequency and can be calculated as follows:

$$TF = \frac{\text{No. of instances of word } w \text{ in sentence } s}{\text{Total no. of words in sentence } s} \tag{4.1}$$

   IDF refers to inverse document frequency and can be calculated as follows:

$$IDF = \frac{\text{Total no. of sentences}}{\text{No. of sentences that contain word w}} \tag{4.2}$$

   Here, the IDF value of a word depends upon the total number of documents thus it remains the same throughout all the documents. On the contrary, TF values for a word differ from document to document.

   For example, if we have 2 sentences: 1) I love playing cricket, 2) Prahar play cricket. Here, "Prahar" in the second sentence occurs only once in that sentence and the total number of words in the second sentence is 3, hence, TF value for the word "Prahar" for second sentence is $1/3$. Also, we have two sentences and the word "Prahar" occurs in the second sentence, therefore the IDF value of the word "Prahar" is $2/1 = 2$.

4.4 Data Splitting

As shown in Figure 4.1, the fourth step includes data splitting. Finally, we split the data into 2 parts: 80% data for training and 20% of data for testing. We define this data as *NewReviews* and *TrainReviews*. *NewReviews* are the reviews for which we want to predict the results and *TrainReviews* are the reviews on which we can train the model.

4.5 Similarity Model

As shown in Figure 4.1, in the fifth step, our similarity model predicts the reviewers using the same string comparison techniques used in the state-of-the-art

RevFinder. However, we extended it to compute similarities between project or sub-projects and author information. These review similarity scores are propagated to each code-reviewer who has involved in. In project or sub-project information, we only considered the sub-project, as project information is mostly the same in GitHub projects. Thus, we removed that constant values and only considered sub-project information as we wanted to build our approach for a generalized purpose.

Algorithm 1 Similarity Model

Input: $nR(NewReview), TrainReviews$
Output: *Reviewers* (Sorted list of reviewers assigned with score)

1: **procedure** SIMILARITYPREDICTION(nR) ▷ Computes similarity between NewReview nR and all TrainReviews
2: $subProjnR \leftarrow extractSubProj(nR)$
3: $authornR \leftarrow extractAuthor(nR)$
4: $filesnR \leftarrow extractFiles(nR)$
5: **while** $cR \in TrainReviews$ **do** ▷ Compare with all TrainReviews
6: $subProjcR \leftarrow extractSubProj(cR)$
7: $filescR \leftarrow extractFiles(cR)$
8: $authorcR \leftarrow extractAuthor(cR)$
9: $reviewercR \leftarrow extractReviewers(cR)$
10: $score \leftarrow dict()$
11: **while** $fN \in filesnR$ **do**
12: **while** $cN \in filescR$ **do**
13: $score \leftarrow score + similarity(fN, cN)$
14: **end while**
15: **end while**
16: $score \leftarrow score + similarity(subProjnR, subProjcR)$
17: $score \leftarrow score + similarity(authornR, authorcR)$
18: $NormalizationFactor \leftarrow \frac{1.0}{\sum v \in score}$
19: $score \leftarrow score * NormalizationFactor$
20: **while** $r \in reviewercR$ **do** ▷ Propagate similarity scores to reviewers involved in a closed review cR
21: $reviewers[r] \leftarrow reviewers[r] + score[r]$
22: **end while**
23: **end while**
24: $reviewers \leftarrow sortByScore(reviewers)$
25: **end procedure**

We implemented this similarity model in Python. Algorithm 1 shows the algorithm of our similarity model. It takes the *NewReview nr* and all *TrainReviews* as input and then extracts the File-Paths, Sub-Projects, Authors information from it. It calculates the similarity score as the combination of all similarity scores of File-Paths, Sub-Projects, Authors between each *NewReview* and all *TrainReviews*. For

computing the similarities, this algorithm uses the four string comparison techniques namely, longest common sub-sequence, longest common prefix, longest common suffix, and longest common sub-string.

Finally, it propagates this score to all the reviewers who are involved in each *TrainReviews* using normalization methods instead of using the borda count [30] combination method used in state-of-the-art *RevFinder*. We defined normalization factor of a dictionary *dict* as follows:

$$\text{Normalization Factor} = \frac{1.0}{\sum v \in dict} \tag{4.3}$$

where, v is the values of dictionary *dict*. For instance, if we have a dictionary with a *key-value* pair of *reviewer-score* then it sums all the values of the score and measures the normalization factor by dividing this value by 1. Now, for normalizing the scores of the reviewers, it multiplies all the reviewers' scores with the normalization factor in four dictionaries separately: 1) reviewer-score dictionary obtained from computing similarities using longest common sub-sequence, 2) reviewer-score dictionary obtained from computing similarities using longest common sub-string, 3) reviewer-Score dictionary obtained from computing similarities using longest common prefix, 4) reviewer-Score dictionary obtained from computing similarities using a longest common suffix. It then combines all these dictionaries into the final reviewer's dictionary and then sorts this set of reviewers based on their normalized scores in descending order. The final output of this model is a set of reviewers with a score assigned to them. Our results showed that for each newly added feature, our similarity model greatly improves the accuracies of state-of-the-art *RevFinder*.

4.6 Ensemble Modeling

As shown in Figure 4.1, our sixth step includes the ensemble technique to predict the set of reviewers based on the selected classification algorithm. We tested four classification algorithms of machine learning namely, Support Vector Machine (SVM), Random Forest, K Nearest Neighbour, and Multinomial Naive Bayes. Our results showed that SVM gave the best results when used with the same settings for all the 20 projects of Gerrit and GitHub.

Algorithm 2 shows the algorithm of the classification model used in our study. We implemented the SVM Classification Model [36] in Python Language. The vectorized representation of the subject of change and reviewers of all closed re-

Algorithm 2 SVM Classification Model

Input: $trainVector, trainRev, testVector$
Output: $reviewer$

1: **procedure** SVMPREDICTION($trainVector, trainRev, testVector$)
2: $model \leftarrow fit(trainVector, trainRev)$
3:
4: $reviewer \leftarrow model.predict(testVector)$
5: ▷ Prediction using Support Vector Machine
6: **end procedure**

views is taken as input in our prediction model. The model then predicts (or recommends) an appropriate reviewer for a given vectorized representation of the subject of the change of a new review.

4.7 CORMS Controller

As shown in Figure 4.1, in the seventh step of the proposed approach, our CORMS Controller process all the new reviews.

We implemented CORMS Controller in Python. Algorithm 3 shows the algorithm of our CORMS Controller. It takes all *NewReviews* and *TrainReviews* as input. Then it constructs the reviewer activeness by extracting the closed date from all *TrainReviews* and propagating or updating these to all the reviewers who are involved in it. This controller then sorts this reviewer's activeness based on their last reviewed date. Then it processes all the new reviews and passes each of these pre-processed reviews to both similarity and SVM Models to obtain the following:

1. Sorted list of reviewers predicted by similarity model.
2. Reviewer predicted by the Support Vector Machine model.

This CORMS Controller then combines these two results and normalizes them using the same normalization factor defined in the similarity model and builds the combined list of a set of reviewers with a final score assigned to them. It then filters the reviewers based on the *Reviewer's Activeness*. This filtering can be done by removing all reviewers whose last reviewed period is greater than 12 months. The final output of this model is the sorted list of reviewers.

Algorithm 3 CORMS Controller

Input: $NewReviews, TrainReviews$

Output: $Reviewers$ (Sorted list of reviewers assigned with score)

```
1: procedure HYBRIDPREDICTION(nR)
2:     while cR ∈ TrainReviews do
3:         cd ← extractClosedDate(cR)
4:         trainRev ← extractReviewers(cr)
5:         activeness ← update(activeness, trainRev, cR)
6:     end while
7:     activeness ← sortByLastReviewedDate(activeness)
8:     trainVector ← tfidf(extractSubject(TrainReviews))
9:     while nR ∈ newReviews do
10:        testVector ← tfidf(extractSubject(nr))
11:        revSim ← SimilarityPrediction(nr)
12:        revSvm ← SVMPrediction(trainVector, trainRev, testVector)
13:        reviewer ← revSim + revSvm
14:        revFact ← 1.0 / ∑v∈reviewer
15:        reviewer ← reviewer ∗ revFact
16:        while r ∈ finalScore do
17:            if activeness[r] > 12 then
18:                reviewers ← remove(reviewer[r])
19:            end if
20:            reviewers ← sortByScore(reviewers)
21:        end while
22:    end while
23: end procedure
```

CHAPTER 5

Experimentation and Results

In this chapter, we evaluate the performance of *CORMS*. Our experimental environment includes 14 GB of GPU memory and 12 GB RAM.

5.1 Experimental Setup

We mined 30,648 reviews from 20 projects of both Gerrit and GitHub from 2020 to 2021. For the processing of the textual content, we use the *Gensim*[12] library for removing stop-words and *WordNetLemmatizer* library of NLTK[1] for word lemmatization and *string*[11] library for removing the punctuation. The state-of-the-art *RevFinder* [40] and TIE [42] used the data-sets provided by Thongatanunam et al. [40] which contain a total of 42,045 reviews mined in time-period 2008-2014. These approaches have experimented with four projects of Gerrit: 1) OpenStack, 2) LibreOffice, 3) QT, 4) Android.

The proposed approach, *CORMS*, examines the author's information and the subject of the change information which is not available in the data-set provided by Thongatanunam et al. [40], hence, we created a new data-set by mining from both Gerrit and GitHub.

5.1.1 Project Selection

We have selected a total of 20 projects for the analysis of *CORMS*. In which we selected 10 projects from Gerrit and 10 projects from GitHub. Our selected Gerrit projects are 1) OpenStack, 2) LibreOffice, 3) QT, 4) Android, 5) Go, 6) Eclipse, 7) Unlegacy, 8) Cloudera, 9) Opencord, 10) Chromium. While our GitHub projects are as follows: 1) TWBS, 2) Joyent, 3) Jquery 4) NodeJS, 5) Shopify, 6) H5bp, 7) Nix-Community, 8) BSSW, 9) RenovateBot, 10) FullStorydev.

[1] https://www.nltk.org/_modules/nltk/stem/wordnet.html

Table 5.1: Statistics of the data collected from OSS

S.No.	Project	# Revi.	# Files	# Re.	# Avg. Re.
1	**Go**	1842	11470	78	3.64
2	**Eclipse**	1965	17458	135	2.46
3	**QT**	2144	10832	116	3.61
4	**Openstack**	2573	5657	269	3.99
5	**Android**	1903	1117	119	4.45
6	**Libreoffice**	1879	10573	54	2.26
7	**Unlegacy**	2624	2342	8	1.41
8	**Cloudera**	2169	15597	46	3.96
9	**OpenCord**	6163	53788	45	4.43
10	**Chromium**	3113	10501	278	2.63
11	**TWBS**	656	6661	12	1.09
12	**Joyent**	139	4455	24	1.13
13	**BSSW**	208	2588	19	1.4
14	**RenovateBot**	184	21228	6	1.09
15	**Shopify**	584	5360	171	1.44
16	**Nix-Community**	418	5072	28	1.19
17	**GetSentry**	781	22525	66	1.22
18	**FullStoryDev**	310	5424	27	1.66
19	**NodeJS**	113	2114	35	1.58
20	**H5bp**	98	648	8	1.02

5.1.2 Statistics of collected data

Table 5.1 shows the statistics of the collected data used for the analysis of *CORMS* from 2020 to 2021. The columns correspond to the serial number (S.No), the project name (Project), the total number of collected reviews including both the closed and open reviews (# Revi.), the total number of modified files (# Files), the total number of unique code-reviewers (# Re.), and the average number of code-reviewers per review (# Avg. Re.). The first ten projects correspond to the projects from Gerrit, and the last ten projects belong to GitHub.

5.2 Evaluation Metrics

To evaluate the performance of the *CORMS*, we use the Top-K prediction accuracy and the Mean Reciprocal Rank (*MRR*) that are common in the evaluation of code review and software engineering recommendation systems.

5.2.1 Top-K

If we have sorted list of reviewers, then Top-K accuracy denotes percentage of reviewers who are present at the Top-K positions in this sorted list. The Top-K accuracy is computed as follows:

$$\text{Top-K accuracy} = \frac{\sum_{r \in R} \text{ isPresent } (r, k)}{|Rev|} \tag{5.1}$$

where, *Rev* refers to a set of reviews and *isPresent (r,k)* returns 1 if at least one of the actual reviewers r is correctly recommended at the Top-k position, otherwise returns 0. In this paper, we set k = 1, 3, 5, and 10. The higher the k value, the better a reviewer recommendation technique performs.

5.2.2 Mean Reciprocal Rank (*MRR*)

MRR corresponds to the average of the reciprocal ranks of a set of recommendations. It describes a mean position of actual reviewer in the sorted list. The *MRR* is computed as follows:

$$\text{Mean Reciprocal Rank} = \frac{1}{|Rev|} \sum_{r \in R} \frac{1}{\text{rank}(r, \text{slist}(r))\rangle} \tag{5.2}$$

where, *Rev* refers to a set of reviews and *rank(r,slist(r))* returns the position of a reviewer r present in the sorted list - *slist*. Hence, the outcome of *1/rank* will be 0, if no reviewers are present in the list, otherwise if the reviewer appear in the top positions, then *1/rank* will be near to 1, and if reviewer appear far from the top, then *1/rank* will be near to 0.

5.3 Research Questions and Analysis

5.3.1 Research Questions

Our experimental study aims at addressing the three main research questions (RQs).

1. How efficient is *CORMS* in recommending reviewers? What is the performance of *CORMS* when compared with the state-of-the-art *RevFinder*?

Table 5.2: Performance evaluation of *CORMS* and *RevFinder*

No	Project	Top-1			Top-3			Top-5			Top-10			MRR		
		Ext	Rev	Imp%	Ext	Rev	Imp%	Ext	Rev	Imp%	Ext	Rev	Imp%	Ext	Rev	Imp%
1	**Go**	43.43	25.18	**72.48**	70.07	43.8	**59.98**	74.81	58.76	**27.31**	87.96	75.18	**17**	0.58	0.4	**31.03**
2	**Eclipse**	55.81	18.31	**204.81**	72.67	31.4	**131.43**	82.85	41.57	**99.3**	87.21	58.72	**48.52**	0.67	0.3	**55.22**
3	**QT**	49.45	45.86	**7.83**	64.92	55.52	**16.93**	70.72	59.39	**19.08**	80.39	63.81	**25.98**	0.59	0.53	**10.17**
4	**Openstack**	40.66	28.2	**44.18**	85.98	52.62	**63.4**	93.13	61.34	**51.83**	98.08	66.28	**47.98**	0.64	0.44	**31.25**
5	**Android**	60.94	50	**21.88**	65.63	57.81	**13.53**	73.44	60.94	**20.51**	78.13	65.63	**19.05**	0.66	0.55	**16.67**
6	**Libreoffice**	74.57	54.29	**37.35**	80.57	64.57	**24.78**	83.71	72	**16.26**	89.43	78	**14.65**	0.79	0.62	**21.52**
7	**Unlegacy**	64.79	47.89	**35.29**	95.77	85.92	**11.46**	97.89	97.89	**0**	100	99.3	**0.7**	0.77	0.67	**12.99**
8	**Cloudera**	46.15	41.02	**12.51**	60.51	58.46	**3.51**	71.28	70.26	**1.45**	84.1	82.31	**2.17**	0.58	0.55	**5.17**
9	**OpenCord**	28.67	28.67	**0**	71.67	71.33	**0.48**	88.33	88	**0.37**	96	95	**1.05**	0.53	0.52	**1.89**
10	**Chromium**	37.96	32.64	**16.3**	52.31	46.3	**12.98**	57.18	51.62	**10.77**	61.81	55.79	**10.79**	0.47	0.41	**12.77**
11	**TWBS**	52.38	51.59	**1.53**	86.51	80.95	**6.87**	89.68	88.1	**1.79**	96.83	96.83	**0**	0.7	0.68	**2.86**
12	**Joyent**	57.14	35.71	**60.01**	78.57	42.86	**83.32**	85.71	85.71	**0**	85.71	85.71	**0**	0.69	0.48	**30.43**
13	**BSSW**	45	35	**28.57**	77.5	77.5	**0**	87.5	82.5	**6.06**	92.5	92.5	**0**	0.64	0.57	**10.94**
14	**RenovateBot**	82.5	62.5	**32**	96.25	96.25	**0**	100	100	**0**	100	100	**0**	0.89	0.79	**11.24**
15	**Shopify**	8.92	4.46	**100**	16.96	8.04	**110.95**	23.21	9.82	**136.35**	32.14	17.86	**79.96**	0.16	0.09	**43.75**
16	**Nix-Community**	55.71	51.43	**8.32**	61.43	61.43	**0**	67.14	64.29	**4.43**	67.14	65.71	**2.18**	0.59	0.57	**3.39**
17	**GetSentry**	11.59	5.07	**128.6**	41.3	26.09	**58.3**	55.07	46.38	**18.74**	59.42	57.97	**2.5**	0.29	0.21	**27.59**
18	**FullStoryDev**	23.21	23.21	**0**	64.29	60.71	**5.9**	66.07	64.29	**2.77**	69.64	69.64	**0**	0.44	0.42	**4.55**
19	**NodeJS**	18.75	12.5	**50**	43.75	31.25	**40**	56.25	43.75	**28.57**	62.5	62.5	**0**	0.35	0.27	**22.86**
20	**H5bp**	50	31.82	**57.13**	81.82	45.45	**80.02**	86.36	86.36	**0**	86.36	86.36	**0**	0.66	0.48	**27.27**
-	**AVG**	45.1	34.3	**44.9**	67.5	54.9	**34.4**	74.6	66.7	**20.8**	79.9	73.8	**12.3**	0.58	0.48	**18.4**

2. How much improvement does *CORMS* gain with each newly added feature over state-of-the-art *RevFinder*?

3. Which problems of state-of-the-art *RevFinder* are solved in *CORMS*?

RQ1: How efficient is CORMS in recommending reviewers? What is the performance of CORMS when compared with the state-of-the-art RevFinder?

We compare *CORMS* with *RevFinder* [40], and evaluate them on 20 projects of Gerrit and GitHub, and measure the *MRR* and Top-K accuracies (k = 10, 5, 3, and 1). We define improvement(%) as:

$$\text{Improve(\%)} = \frac{(ExtendAccuracy - OrigAccuracy) * 100}{OrigAccuracy} \tag{5.3}$$

Table 5.2 shows the Top-K accuracies (k = 1, 3, 5, and 10) and *MRR* values of *CORMS* and state-of-the-art *RevFinder*. Here, column 'Rev' corresponds to the accuracies of *RevFinder* and 'Ext' corresponds to the extended accuracies of *CORMS*. We observe that *CORMS* outperforms state-of-the-art *RevFinder* by a significant margin. On average for the 20 projects, *CORMS* achieves top-1, top-3, top-5, and top-10 accuracies, and *MRR* values of 79.9%, 74.6%, 67.5%, 45.1% and 0.58, which improves the state-of-the-art approach *RevFinder* by 12.3%, 20.8%, 34.4%, 44.9% and 18.4%, respectively. Here, the improvement of accuracies of Top-K reduces as the k value of Top-K increases. The reason for that is greater the k value, the more accuracy will be there and thus improvement will be less.

Table 5.3: Performance of *CORMS* with Normalization and Borda Count score propagation techniques

	Borda Count	Normalization
Top-1	38.7 %	44.9 %
Top-3	30.54 %	34.4 %
Top-5	18.3 %	20.8 %
Top-10	12.3 %	12.3 %
MRR	16.9 %	18.4 %

For Gerrit, we observe *CORMS* achieves the most improvement over state-of-the-art *RevFinder* in the project Eclipse; it improves *RevFinder* by 49%, 99%, 131%, 205% and 55% in terms of top-10, top-5, top-3, and top-1 accuracies, and *MRR* respectively. For GitHub, we observe *CORMS* achieves the most improvement over state-of-the-art *RevFinder* in the project Shopify; it improves *RevFinder* by 80%, 136%, 111%, 100% and 44% in terms of top-10, top-5, top-3, and top-1 accuracies, and *MRR* respectively.

Table 5.3 shows the evaluation of the performance of *CORMS* with two score propagation techniques: 'Borda count' [30] used in the *RevFinder* and our proposed 'normalization' technique. We can notice that for the 20 projects, *CORMS* when used with the normalization technique achieves the Top-1, 3, 5, 10 and *MRR* values of 12.3%, 20.8%, 34.4%, 44.9% and 18.4% respectively, which performs better than the Borda count combination technique.

RQ2: How much improvement does CORMS gain with each newly added feature over state-of-the-art RevFinder?

We compare the results of the two prediction models of *CORMS*: SVM model and similarity model with state-of-the-art *RevFinder*. We evaluate these approaches on 20 projects of GitHub and Gerrit and record the Top-K accuracies, and *MRR* values. We define accuracy gain as the difference between the accuracies of our models built over top of *RevFinder*, and state-of-the-art *RevFinder*. Also, we define Top-K gain as the average accuracy gain of top-5, top-3, and top-1 accuracies.

Table 5.4 presents the average Top-K and *MRR* gain of:

1. Similarity Model computing similarities of file-paths and sub-projects, which alone overall improves the state-of-the-art *RevFinder* in Top-K and *MRR* accuracies by 8.6% and 7.29%.

2. Similarity Model computing similarities of file-paths and authors, which alone overall improves the state-of-the-art *RevFinder* in Top-K and *MRR* accuracies by 12.1% and 11.11%.

Table 5.4: Measurement of Accuracy Gain for each Individual Models

		SVM Model (Subject) + RevFinder	Similarity Model (File-Path) + (SubProject)	Similarity Model (File-Path) + (Author)
Overall	**Top-K Gain**	7.98 %	8.6 %	12.1 %
	MRR Gain	6.31 %	7.29 %	11.11 %
Gerrit	**Top-K Gain**	6.05 %	7.3 %	17.58 %
	MRR Gain	3.92 %	5.88 %	15.69 %
GitHub	**Top-K Gain**	9.92 %	9.91 %	6.62 %
	MRR Gain	8.7 %	8.7 %	6.52 %

3. *SVM* Model predicting reviewers based on the textual content(subject), when added with the *RevFinder*, overall improves the state-of-the-art *RevFinder* in Top-K and *MRR* accuracies by 7.98% and 6.31%.

Here, we observe that for each of the newly added feature, both the models positively improves the performance of the state-of-the-art *RevFinder*.

5.3.2 Analysis Results

How CORMS is different than the RevFinder in terms of methodology and working?

Our work aims to improve the approach, *RevFinder*, proposed by Patanamon Thongtanunam et al. [40]. The *RevFinder* computes the similarities between the file paths to recommend code reviewers. Our proposed approach, *CORMS*, also uses the same four-string comparison techniques defined in the RevFinder to compute similarities. However, *CORMS* uses the more effective 'normalization' technique for score propagation compared to the 'Borda count' technique used in state-of-the-art *RevFinder*. Also, the *CORMS* computes the similarities between project/subproject and authors' information too with the file-paths. The CORMS also includes the *SVM* model to examine the textual contents in the subject of the code reviews to recommend appropriate code-reviewers.

Which problems of the state-of-the-art RevFinder is being solved by CORMS?

There are several problems in *RevFinder* such as:

1. It does not consider retired code reviewers.

2. It is not able to recommend code reviewers for new files.

3. It has threats to external validity as their results are limited to 4 data-sets only.

4. Results are unknown for other repositories (e.g., GitHub).

To address all these problems, *CORMS* used a hybrid approach combining both the SVM and similarity model and filtered out the results based on the reviewer's activeness. We solved the following issues:

1. **Retired Reviewers Issue**: by filtering out the reviewers who reviewed 12 months ago.

2. **Issue with newly created files**: State-of-the-art *RevFinder* was based on only the File-Path information. So it can't able to predict the reviewers for the newly created file where no File-Path information is mentioned. We solved this issue by considering other important features such as Authors, Subject of the change, and Sub-project details.

3. **Accuracy enhancements**: On average for the 20 projects of Gerrit and GitHub, *CORMS* can achieve top-10, top-5, top-3, and top-1 accuracies, and Mean Reciprocal Rank (*MRR*) of 79.9%, 74.6%, 67.5%, 45.1% and 0.58, which improves the state-of-the-art approach *RevFinder* by 12.3%, 20.8%, 34.4%, 44.9% and 18.4%, respectively.

4. **External Validity**: In the state-of-the-art *RevFinder* paper, authors tested it on the 4 data-sets of Gerrit. We had extended this to 10 projects of Gerrit to check and compare the results of our approach with existing *RevFinder*. Apart from Gerrit, we also checked the performance of both approaches in 10 GitHub projects.

5.4 Threats to Validity

Threats to *construct validity* refer to the selection of evaluation criteria. We have used pre-defined Top-K prediction accuracies and *MRR* commonly used to evaluate the effectiveness of MCR recommendation techniques [6][40][42][28][19][46][45][11], and various automated software engineering techniques [31][47][37][43]. Hence, the possibility of evaluation bias is very less. Threats to *internal validity* here refer to the code selection and check for error. We have checked, tested, debug our code, and ensured that it would run. However, there is still a possibility of errors that we missed. Also, in the *CORMS* approach, the data needs to be updated on a daily/weekly, or monthly basis based on the organization's requirements and the frequencies of new reviews updated on the project. So, for every

new cycle of data, our classification model needs to be trained again. Though carefully done, one can argue for a potential bias here.

Another threat, *external validity*, may be related to the generalizability of results. Mining data from open-source repositories are challenging especially from GitHub. For instance, in GitHub, for fetching details about the one review, we need to make 4 API calls in case no reviewer is assigned. And, if the reviewer is assigned to the pull request then we need to make a total of 6 API calls. This process makes the mining very complicated for GitHub. However, in Gerrit, we can mine a bunch of reviews in just a single API call. We have analyzed 30,648 reviews from twenty open-source projects for 2020 to 2021. The currently used data-set[2] can be further improved by analyzing more reviews for other projects from Gerrit and GitHub.

[2]https://figshare.com/s/d4d2f350b2ddf2bab2fd

CHAPTER 6

CORMS: A Tool

This chapter gives the overview of our tool and describes its features and various interfaces.

6.1 What is CORMS-TOOL?

CORMS-TOOL is a reviewer recommendation tool to support modern code review. It is uses proposed algorithm *CORMS* and predicts the top reviewers for a new code review request. We built *CORMS-TOOL*[1] using Django MCV framework of python and used MongoDB for providing database connectivity. We hosted *CORMS-TOOL* on Heroku[2]. The current version of *CORMS-TOOL* provides support for 34 projects of Gerrit and GitHub.

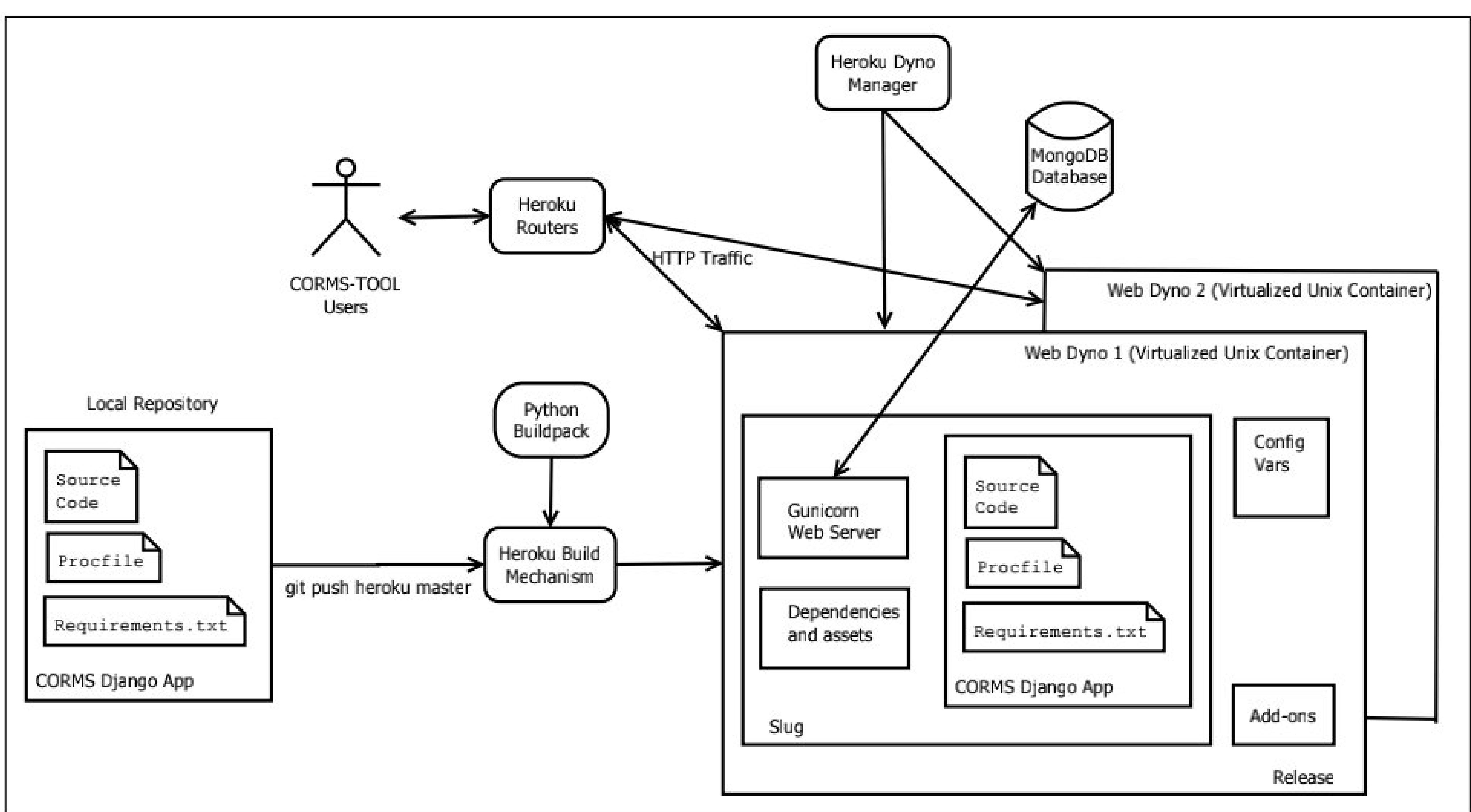


Figure 6.1: CORMS-TOOL architecture

[1] https://youtu.be/t6qnB3IIGMI
[2] http://cormstool.herokuapp.com/

6.2 CORMS-TOOL Architecture

Fig 6.1 describes the architecture of our *CORMS-TOOL*, a Heroku based Django application. It contains following:

1. **Local Repository**: The local repository of *CORMS-TOOL* contains the following items:

   (a) **Source Code**: *CORMS-TOOL* uses Django MVC framework. More details about this framework is given in section 6.3.

   (b) **Procfile**: Procfile is a simple text file without a file extension. A procfile declares its process types on individual lines, each with the following format: <process type> : <command>. <process type> is an alphanumeric name of the command, such as web or worker. <command> indicates the command that every dyno of the process type should execute on startup. Our procfile requires Gunicorn, the production web server that is used for the Django applications.

   (c) **Requirements.txt**: This file contains all the dependencies along with version information.

2. **Build Mechanism and Python Build-pack**: The local repository code is uploaded to Heroku using git. Heroku build mechanism analyzes this code and selects build-pack. As *CORMS-TOOL* uses Django framework, it selects python build-pack. This build-pack is composed of a set of scripts that will perform tasks such as retrieve dependencies mentioned in requirements.txt, create assets, compiles code and creates an app slug.

3. **CORMS-TOOL Users**: The users sends http requests to Heroku routers and receive the http response.

4. **Routers**: Heroku's HTTP routers distribute incoming requests for the *CORMS-TOOL* across the running web dynos. In case of too much of web requests, scaling the number of web dynos is required to scale the capacity of *CORMS-TOOL* to handle web traffic.

5. **Web Dynos**: Each time a new release of *CORMS-TOOL* is created, the Heroku Dyno Manager kills the running dynos and creates new dynos with a new release. These dynos are isolated and virtualized Linux containers used to execute the code. Currently, *CORMS-TOOL* is limited to 100 total dynos. Each dyno contains slug, config vars and other addons.

6. **Slug**: After git push to Heroku, code is received by the slug compiler which transforms the repository into a slug. This slug contains the compressed copy comprising of *CORMS-TOOL* git repository and gunicorn web server along with the *CORMS-TOOL* dependencies and assets.

7. **Gunicorn Web Server**: *CORMS-TOOL* uses Gunicorn web server. Gunicorn is a pure python HTTP server for Web Server Gateway Interface(WSGI) applications, which is is Django's primary deployment platform and the python standard for web servers and applications. It allows to run python application concurrently by running multiple python processes within a single dyno.

8. **MongoDB Database**: *CORMS-TOOL* uses MongoDB database for providing database connectivity to web sever. Our MongoDB database includes multiple collections such as feedback collection, project collection, code review collections for each project, and reviewer profiles collections for each project.

6.3 Django MVC Framework of CORMS-TOOL

Figure 6.2 shows the Django MVC Framework of CORMS-TOOL. It contains following modules:

1. **URL Dispatcher (URLs.py)**: It maps the requested URL to a view function and calls it. If caching is enabled, the view function can check to see if a cached version of the page exists and bypass all further steps, returning the cached version, instead.

2. **View Functions**: It perform the requested actions such as predict reviewers, view supported projects, create new project request, etc. After performing the requested tasks, view returns the HTTP response object.

3. **Model/Document**: It defines the data in python and are used to interact with the database. *CORMS-TOOL* uses MongoDB database, which is a NoSQL database. Thus the models in Django are replaced by the documents and fields when used to interact with MongoDB.

4. **Template**: Template returns the HTML pages.

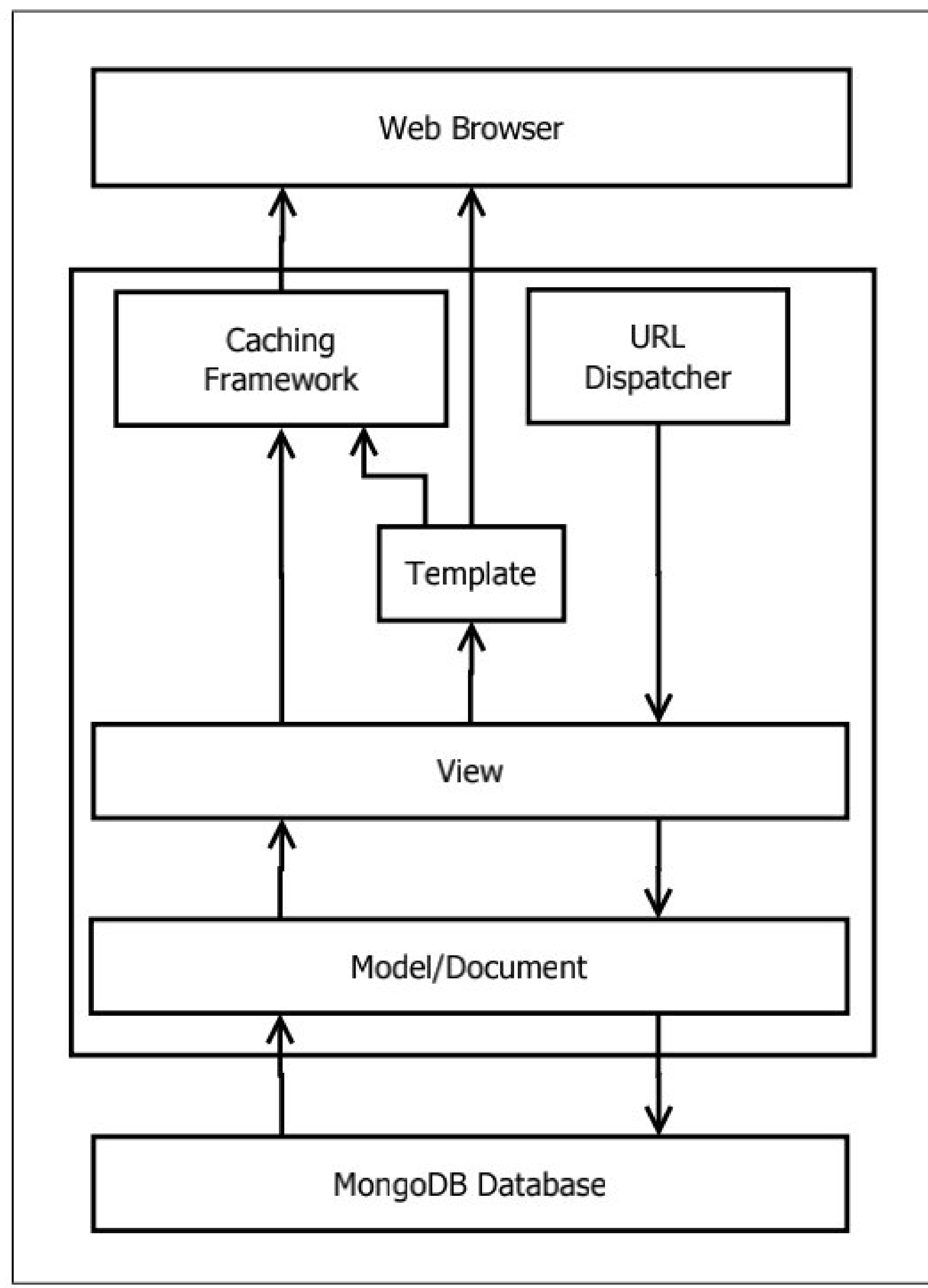


Figure 6.2: Django MVC Framework of CORMS-TOOL

6.4 Features of CORMS-TOOL

The reviewer recommendation phase of *CORMS-TOOL* includes following features:

1. It provides tool-support to predict reviewers
2. It provides the sorted list of reviewers by their score in descending order with full details of reviewers along with reviewer activeness and current workload of the reviewer.
3. It provides support to copy and print the results with filters.
4. It provides support to convert data into pdf, csv or excel sheet with filters.

The feeback phase of *CORMS-TOOL* includes following features:

1. It allows user to view overall feedback of *CORMS* given by other users.
2. It allows user to provide feedback for the predictions given by *CORMS*.

The create or view project phase of *CORMS-TOOL* includes following features:

1. It supports projects from both the GitHub and Gerrit OSS platform
2. It allows creation of new project request
3. It supports view/search/filter operations on all supported projects
4. It supports view/search/filter operations on all requested projects

6.5 Various Interfaces of CORMS-TOOL

6.5.1 Code-Review Interface

Figure 6.3: Step-1 Interface: OSS Platform Selection

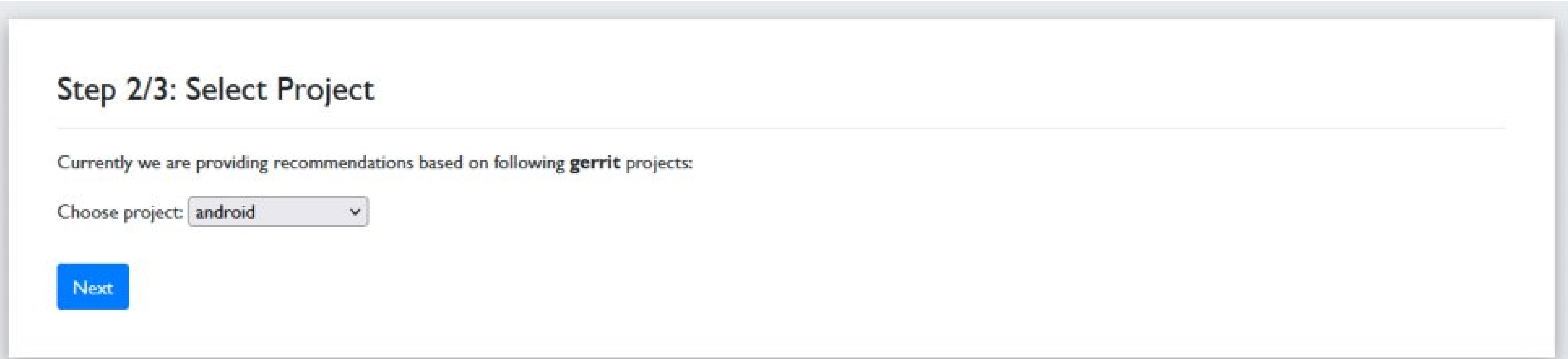


Figure 6.4: Step-2 Interface: Project Selection

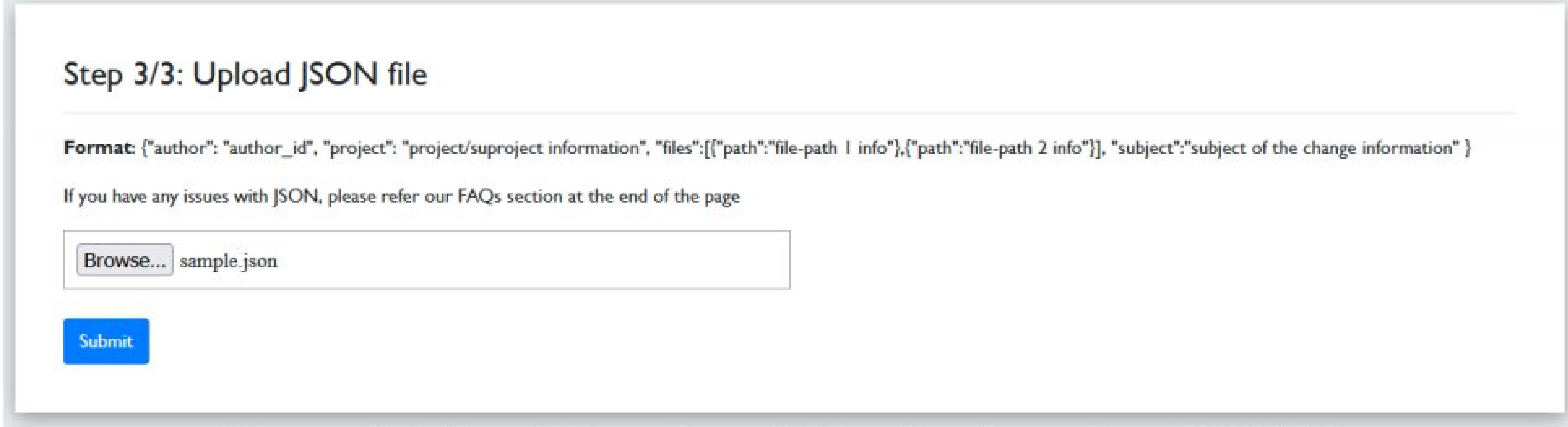
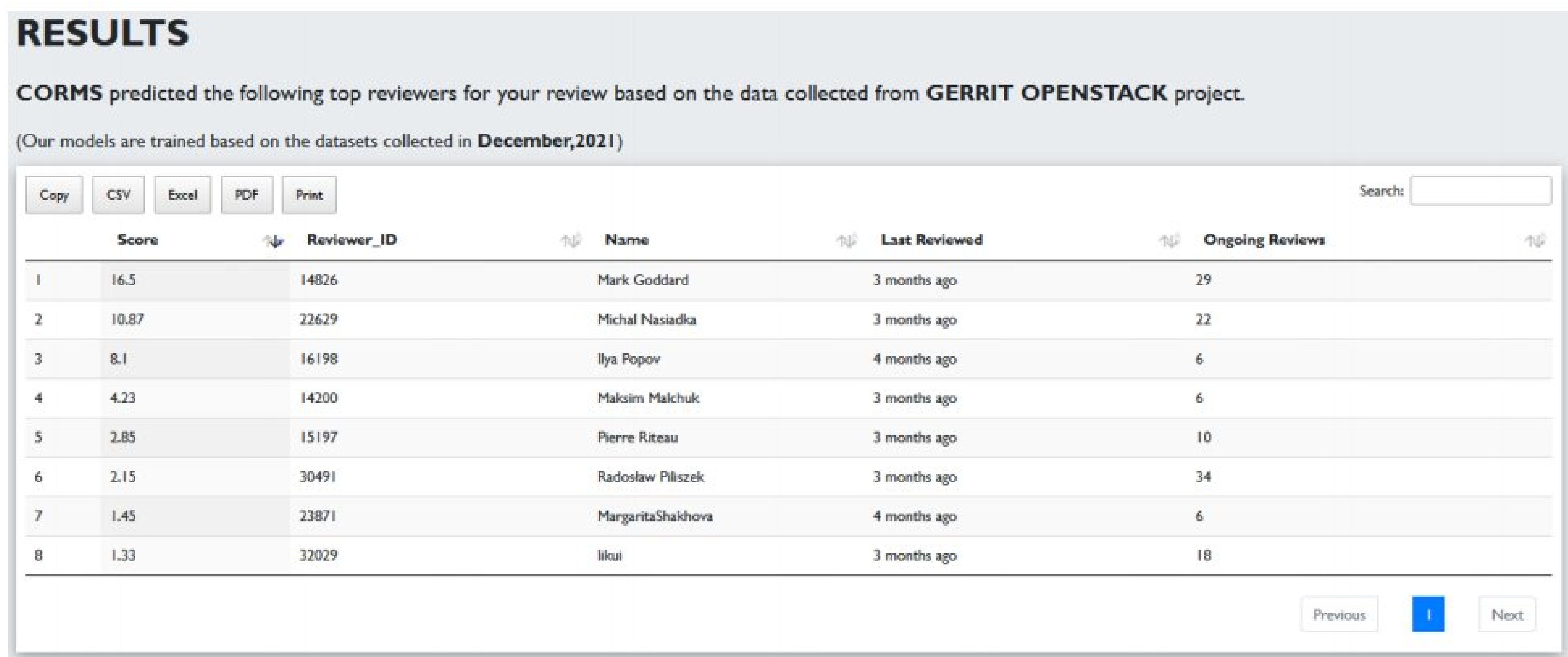


Figure 6.5: Step-3 Interface: Upload code-review JSON file

Figure 6.3, 6.4 and 6.5 shows the three simple steps to predict reviewers using *CORMS-TOOL*. In the first step, the user needs to select the OSS platform

from either Gerrit or GitHub. In the second step *CORMS-TOOL* will show the supported projects in the platform selected by the user. The user needs to select his/her project in this step. In the third step, the user needs to upload the JSON file containing the code-review details. The format of this JSON file contains the information about the authors, projects, file paths, and subject.

6.5.2 Results and Feedback Interface



	Score	Reviewer_ID	Name	Last Reviewed	Ongoing Reviews
1	16.5	14826	Mark Goddard	3 months ago	29
2	10.87	22629	Michal Nasiadka	3 months ago	22
3	8.1	16198	Ilya Popov	4 months ago	6
4	4.23	14200	Maksim Malchuk	3 months ago	6
5	2.85	15197	Pierre Riteau	3 months ago	10
6	2.15	30491	Radosław Piliszek	3 months ago	34
7	1.45	23871	MargaritaShakhova	4 months ago	6
8	1.33	32029	likui	3 months ago	18

Figure 6.6: Results Interface

Figure 6.6 shows the interface of predictions given by *CORMS-TOOL*. It includes the top reviewers sorted by their score in descending order. It also provides full details of reviewers such as reviewer id and reviewer name along with the reviewer's activeness, and current workload of the reviewer. Reviewer workload is calculated as the number of ongoing reviews, the reviewer is working on. Reviewer activeness is calculated as the difference between the current month and the last reviewed month when the reviewer completed a successful review.

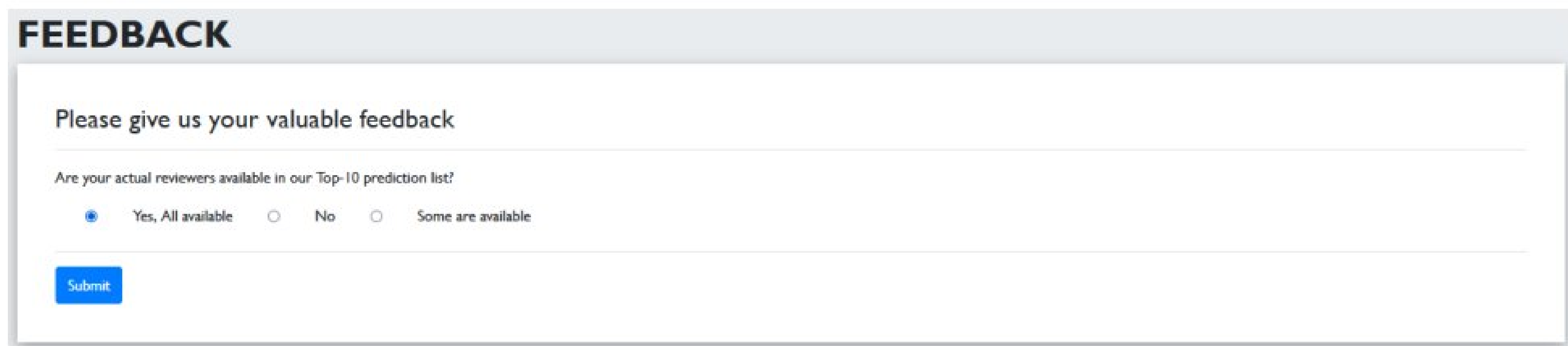


Figure 6.7: Submit Feedback Interface

Figure 6.7 shows the submit feedback interface where the user can provide feedback on the predictions given by *CORMS-TOOL*, whether it includes all the appropriate reviewers or some of the appropriate reviewers, or none of the appropriate reviewers. Users can also see the overall feedback given by other users,

6.5.3 Create or View Project Interface

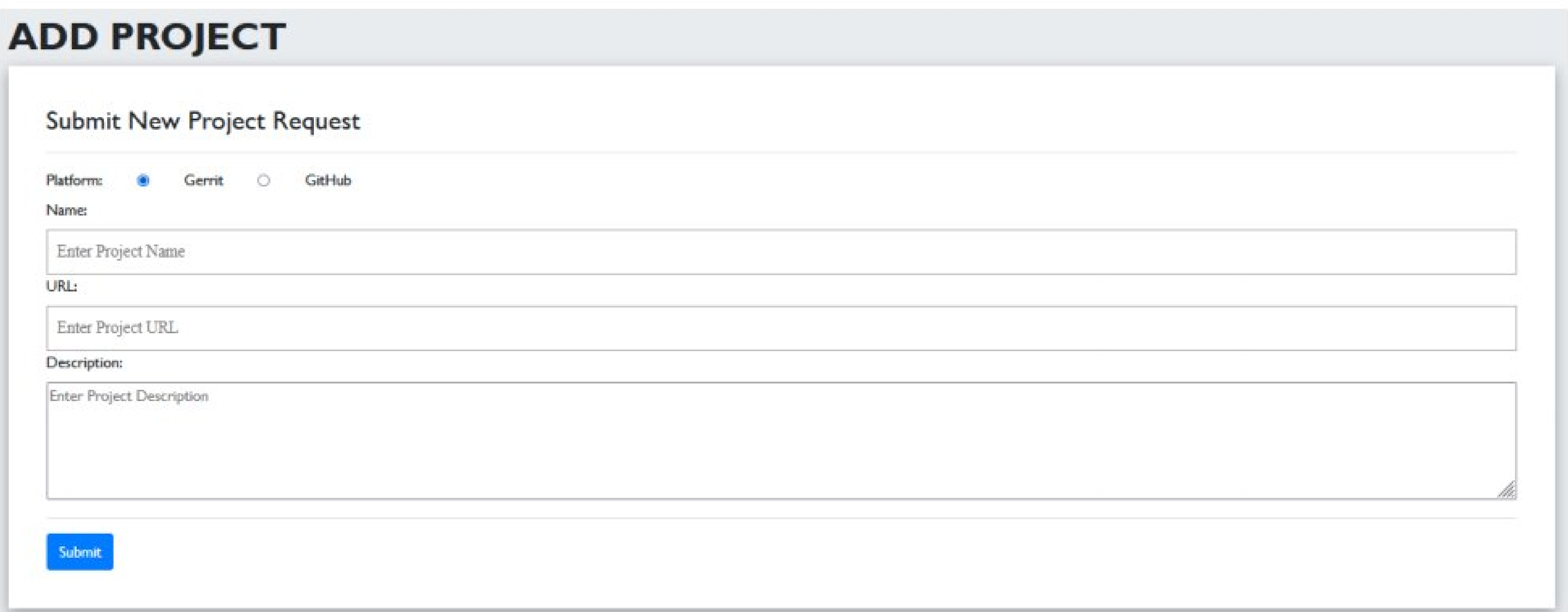
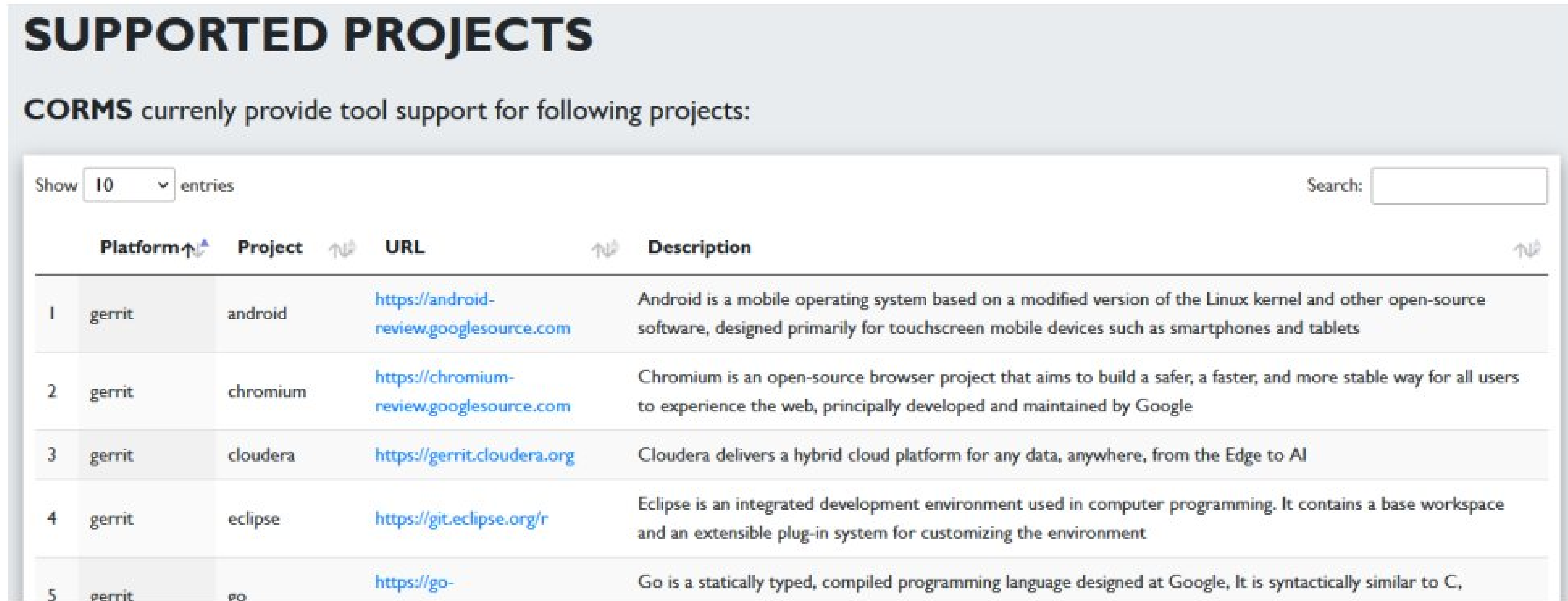


Figure 6.8: Create New Project Request Interface

If the project is not listed in the supported project list, the user can create a new project request as shown in figure 6.8. Creating a new project request requires the information about project platform, name, URL, and a small description of the project. Once the user has successfully submitted the request, he/she can view his project inside the requested project list and can track the status. Approval of a new project request can take about 1 week, as once we receive such requests, we need to mine, pre-process and test all the relevant code reviews and reviewer profiles of that project using our mining scripts written in python. Once it is completed we will upload its data-set to our MongoDB cloud.



SUPPORTED PROJECTS

CORMS currently provide tool support for following projects:

Show 10 entries Search:

	Platform	Project	URL	Description
1	gerrit	android	https://android-review.googlesource.com	Android is a mobile operating system based on a modified version of the Linux kernel and other open-source software, designed primarily for touchscreen mobile devices such as smartphones and tablets
2	gerrit	chromium	https://chromium-review.googlesource.com	Chromium is an open-source browser project that aims to build a safer, a faster, and more stable way for all users to experience the web, principally developed and maintained by Google
3	gerrit	cloudera	https://gerrit.cloudera.org	Cloudera delivers a hybrid cloud platform for any data, anywhere, from the Edge to AI
4	gerrit	eclipse	https://git.eclipse.org/r	Eclipse is an integrated development environment used in computer programming. It contains a base workspace and an extensible plug-in system for customizing the environment
5	gerrit	go	https://go-	Go is a statically typed, compiled programming language designed at Google, It is syntactically similar to C,

Figure 6.9: View Supported Projects Interface

Figure 6.9 shows the interface of the supported projects. It includes the platform, project name, project URL, and project description. Current version of *CORMS-TOOL* supports 34 projects. Once the request for the new project is approved, the project will be visible to the supported project list and users can predict reviewers for that project.

CHAPTER 7

Conclusions and Future Work

We proposed a hybrid approach, *CORMS*, which works on similarity analysis to compute similarities among file-paths, projects/sub-projects, author information, and prediction models to recommend reviewers based on the subject of the change. We conducted a detailed analysis on the widely used 20 projects of both Gerrit and GitHub to compare our results with the state-of-the-art *RevFinder*. Our results show that on average *CORMS* can achieve top-1, top-3, top-5, and top-10 accuracies, and Mean Reciprocal Rank (*MRR*) of 45.1%, 67.5%, 74.6%, 79.9% and 0.58 for the 20 projects, which improves the state-of-the-art approach *RevFinder* by 44.9%, 34.4%, 20.8%, 12.3% and 18.4%, respectively. Finally, We built and hosted a *CORMS-TOOL* to provide a tool-support for recommending reviewers using a hybrid approach, *CORMS* for Modern Code Review.

The proposed methodology doesn't consider the reviewer's workload. Reviewer workload defines the number of ongoing reviews per reviewer and there may be some cases where our approach predicts the same reviewers again irrespective of their current workload. Thus the addition of a workload feature is required considering the user-centric approaches. The proposed approach also doesn't consider the effect of social collaborations between the developers and reviewers. Previous literature showed that it is also an important feature for the recommendation of appropriate code reviewers. Thus it would be interesting to investigate the results with considering the social relations. Commit messages that were used in *TIE* contains more textual content than the subject field used in our proposed approach *CORMS*. Thus it would be interesting to investigate the performance of *CORMS* with the commit messages. The current version of *CORMS-TOOL* is not integrated with any *OSS* platform. With complete integration, the tool would be able to fetch all the details itself rather than asking users to upload a set of information through the JSON file. Thus, we consider it our future work.

Publications

Prahar Pandya and Saurabh Tiwari, *CORMS: A GitHub and Gerrit based Hybrid Code Reviewer Recommendation Approach for Modern Code Review*, In 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2022), Research Paper/Technical Track, Mon 14 - Fri 18 November 2022, Singapore.
[Accepted], Core A* Ranked Conference.

References

[1] Ö. Albayrak and D. Davenport. Impact of maintainability defects on code inspections. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 1–4, 2010.

[2] F. Armstrong, F. Khomh, and B. Adams. Broadcast vs. unicast review technology: Does it matter? In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 219–229. IEEE, 2017.

[3] A. Bacchelli and C. Bird. Expectations, outcomes, and challenges of modern code review. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 712–721. IEEE, 2013.

[4] A. Bacchelli and C. Bird. Expectations, outcomes, and challenges of modern code review. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 712–721. IEEE, 2013.

[5] R. Baeza-Yates, B. Ribeiro-Neto, et al. *Modern information retrieval*, volume 463. ACM press New York, 1999.

[6] V. Balachandran. Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 931–940. IEEE, 2013.

[7] O. Baysal, O. Kononenko, R. Holmes, and M. W. Godfrey. Investigating technical and non-technical factors influencing modern code review. *Empirical Software Engineering*, 21(3):932–959, 2016.

[8] M. Beller, A. Bacchelli, A. Zaidman, and E. Juergens. Modern code reviews in open-source projects: Which problems do they fix? In *Proceedings of the 11th working conference on mining software repositories*, pages 202–211, 2014.

[9] A. Bosu, J. C. Carver, M. Hafiz, P. Hilley, and D. Janni. Identifying the characteristics of vulnerable code changes: An empirical study. In *Proceedings*

of the 22nd ACM SIGSOFT international symposium on foundations of software engineering, pages 257–268, 2014.

[10] A. Bosu, M. Greiler, and C. Bird. Characteristics of useful code reviews: An empirical study at microsoft. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 146–156. IEEE, 2015.

[11] M. Chouchen, A. Ouni, M. W. Mkaouer, R. G. Kula, and K. Inoue. Whoreview: A multi-objective search-based approach for code reviewers recommendation in modern code review. *Applied Soft Computing*, 100:106908, 2021.

[12] N. Davila and I. Nunes. A systematic literature review and taxonomy of modern code review. *Journal of Systems and Software*, page 110951, 2021.

[13] E. W. dos Santos and I. Nunes. Investigating the effectiveness of peer code review in distributed software development. In *Proceedings of the 31st brazilian symposium on software engineering*, pages 84–93, 2017.

[14] F. Ebert, F. Castor, N. Novielli, and A. Serebrenik. Confusion detection in code reviews. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 549–553. IEEE, 2017.

[15] F. Ebert, F. Castor, N. Novielli, and A. Serebrenik. Confusion in code reviews: Reasons, impacts, and coping strategies. In *2019 IEEE 26th international conference on software analysis, evolution and reengineering (SANER)*, pages 49–60. IEEE, 2019.

[16] M. E. Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 38(2.3):258–287, 1999.

[17] N. Fatima, S. Chuprat, and S. Nazir. Challenges and benefits of modern code review-systematic literature review protocol. In *2018 International Conference on Smart Computing and Electronic Enterprise (ICSCEE)*, pages 1–5. IEEE, 2018.

[18] G. Gousios, A. Zaidman, M.-A. Storey, and A. Van Deursen. Work practices and challenges in pull-based development: The integrator's perspective. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 358–368. IEEE, 2015.

[19] J. Jiang, J.-H. He, and X.-Y. Chen. Coredevrec: Automatic core member recommendation for contribution evaluation. *Journal of Computer Science and Technology*, 30(5):998–1016, 2015.

[20] J. Jiang, A. Mohamed, and L. Zhang. What are the characteristics of reopened pull requests? a case study on open source projects in github. *IEEE Access*, 7:102751–102761, 2019.

[21] O. Kononenko, O. Baysal, and M. W. Godfrey. Code review quality: How developers see it. In *Proceedings of the 38th international conference on software engineering*, pages 1028–1038, 2016.

[22] O. Kononenko, O. Baysal, L. Guerrouj, Y. Cao, and M. W. Godfrey. Investigating code review quality: Do people and participation matter? In *2015 IEEE international conference on software maintenance and evolution (ICSME)*, pages 111–120. IEEE, 2015.

[23] V. Kovalenko and A. Bacchelli. Code review for newcomers: is it different? In *Proceedings of the 11th International Workshop on Cooperative and Human Aspects of Software Engineering*, pages 29–32, 2018.

[24] V. Kovalenko, N. Tintarev, E. Pasynkov, C. Bird, and A. Bacchelli. Does reviewer recommendation help developers? *IEEE Transactions on Software Engineering*, 46(7):710–731, 2018.

[25] A. Lee and J. C. Carver. Are one-time contributors different? a comparison to core and periphery developers in floss repositories. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–10. IEEE, 2017.

[26] J. Liang and O. Mizuno. Analyzing involvements of reviewers through mining a code review repository. In *2011 Joint Conference of the 21st International Workshop on Software Measurement and the 6th International Conference on Software Process and Product Measurement*, pages 126–132. IEEE, 2011.

[27] R. Paul, A. Bosu, and K. Z. Sultana. Expressions of sentiments during code reviews: Male vs. female. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 26–37. IEEE, 2019.

[28] M. M. Rahman, C. K. Roy, and J. A. Collins. Correct: Code reviewer recommendation in github based on cross-project and technology experience. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, pages 222–231, 2016.

[29] M. M. Rahman, C. K. Roy, and R. G. Kula. Predicting usefulness of code review comments using textual features and developer experience. In *2017*

IEEE/ACM 14th International Conference on Mining Software Repositories (MSR), pages 215–226. IEEE, 2017.

[30] R. Ranawana and V. Palade. Multi-classifier systems: Review and a roadmap for developers. *International journal of hybrid intelligent systems*, 3(1):35–61, 2006.

[31] S. Rao and A. Kak. Retrieval from software libraries for bug localization: a comparative study of generic and composite text models. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, pages 43–52, 2011.

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

[33] S. Ruangwan, P. Thongtanunam, A. Ihara, and K. Matsumoto. The impact of human factors on the participation decision of reviewers in modern code review. *Empirical Software Engineering*, 24(2):973–1016, 2019.

[34] C. Sadowski, E. Söderberg, L. Church, M. Sipko, and A. Bacchelli. Modern code review: a case study at google. pages 181–190, 05 2018.

[35] D. Spadini, M. Aniche, M.-A. Storey, M. Bruntink, and A. Bacchelli. When testing meets code review: Why and how developers review tests. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 677–687. IEEE, 2018.

[36] S. Suthaharan. Support vector machine. In *Machine learning models and algorithms for big data classification*, pages 207–235. Springer, 2016.

[37] A. Tamrawi, T. T. Nguyen, J. M. Al-Kofahi, and T. N. Nguyen. Fuzzy set and cache-based approach for bug triaging. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 365–375, 2011.

[38] P. Thongtanunam, S. McIntosh, A. E. Hassan, and H. Iida. Investigating code review practices in defective files: An empirical study of the qt system. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 168–179. IEEE, 2015.

[39] P. Thongtanunam, S. McIntosh, A. E. Hassan, and H. Iida. Review participation in modern code review. *Empirical Software Engineering*, 22(2):768–817, 2017.

[40] P. Thongtanunam, C. Tantithamthavorn, R. G. Kula, N. Yoshida, H. Iida, and K.-i. Matsumoto. Who should review my code? a file location-based code-reviewer recommendation approach for modern code review. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 141–150. IEEE, 2015.

[41] R. Wen, D. Gilbert, M. G. Roche, and S. McIntosh. Blimp tracer: Integrating build impact analysis with code review. In *2018 IEEE International conference on software maintenance and evolution (ICSME)*, pages 685–694. IEEE, 2018.

[42] X. Xia, D. Lo, X. Wang, and X. Yang. Who should review this change?: Putting text and file location analyses together for more accurate recommendations. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 261–270. IEEE, 2015.

[43] X. Xia, D. Lo, X. Wang, C. Zhang, and X. Wang. Cross-language bug localization. In *Proceedings of the 22nd International Conference on Program Comprehension*, pages 275–278, 2014.

[44] C. Yang, X. Zhang, L. Zeng, Q. Fan, G. Yin, and H. Wang. An empirical study of reviewer recommendation in pull-based development model. In *Proceedings of the 9th Asia-Pacific Symposium on Internetware*, pages 1–6, 2017.

[45] C. Yang, X.-h. Zhang, L.-b. Zeng, Q. Fan, T. Wang, Y. Yu, G. Yin, and H.-m. Wang. Revrec: A two-layer reviewer recommendation algorithm in pull-based development model. *Journal of Central South University*, 25(5):1129–1143, 2018.

[46] Y. Yu, H. Wang, G. Yin, and T. Wang. Reviewer recommendation for pull-requests in github: What can we learn from code review and bug assignment? *Information and Software Technology*, 74:204–218, 2016.

[47] J. Zhou, H. Zhang, and D. Lo. Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 14–24. IEEE, 2012.