2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER) | 978-1-6654-3786-8/22/$31.00 ©2022 IEEE | DOI: 10.1109/SANER53432.2022.00080

# Recommending Code Reviewers for Proprietary Software Projects: A Large Scale Study

Dezhen Kong*‡, Qiuyuan Chen*‡, Lingfeng Bao*§, Chenxing Sun†, Xin Xia*, Shanping Li*
*Zhejiang University, China, {timkong, chenqiuyuan, lingfengbao, shan}@zju.edu.cn, xin.xia@acm.org
†Tencent, China, marssun@tencent.com

***Abstract*—Code review is an important activity in software development, which offers benefits such as improving code quality, reducing defects and distributing knowledge. Tencent, as a giant company, hosts a great number of proprietary software projects that are only open to specific internal developers. Since these proprietary projects receive up to 100,000 of newly submitted code changes per month, it is extremely needed to automatically recommend code reviewers. To this end, we first conduct an empirical study on a large scale of proprietary projects from Tencent, to understand their characteristics and how code reviewer recommendation approaches work on them. Based on the derived findings and implications, we propose a new approach named CAMP that recommends reviewers by considering their collaboration and expertise in multiple projects, to fit the context of proprietary software development. The evaluation results show that CAMP can achieve higher scores on proprietary projects across most metrics than other state-of-the-art approaches, i.e., REVFINDER, CHREV, TIE and COMMENT NETWORK and produce acceptable performance scores for more projects. In addition, we discuss the possible directions of code reviewer recommendation.**

***Index Terms*—reviewer recommendation, code review, proprietary projects**

## I. INTRODUCTION

Code review is an important activity in software development, which is nowadays the best practice in both open source and industrial software projects [1], [2]. The main goal of code review is to improve the overall quality of code changes through manual examination, such as reducing software defects and quality problems of source code [3], [4]. Since the inefficiencies of traditional peer review practices, such as code inspections, which are cumbersome and time-consuming [5], many organizations have adopted a more lightweight and tool-based code review process called Modern Code Review (MCR) [6], [7], which can not only detect defects and quality problems of source code as traditional code review practices but also distribute knowledge and increase team awareness [6].

At Tencent, a giant international company, there are a large number of proprietary projects maintained in internal community. Many teams at Tencent adopt similar MCR workflow as open-source organizations, making a proportion of proprietary projects open to internal developers. Internal developers can contribute code to the projects they are concerned with, and a submitted code change will be merged after one or more code reviewers approve it. In this context, although developers do not build open-source software, the practices in open-source development is used. Considering that there are too many code changes to be reviewed (At Tencent, there are more than 100,000 code changes submitted to their proprietary projects per month), it is not efficient to manually assign proper reviewers for each submitted code change. Therefore, Code Reviewer Recommendation (CRR) tools are needed to automatically find proper reviewers to relieve developer's workload and speed up the code review process [8].

Fortunately, there have been lots of studies dedicated to CRR. Many proposed approaches [8]–[14] mainly consider the expertise of reviewers as a critical factor, and there are also approaches [4], [15]–[18] considering other factors such as collaboration, workload, and knowledge distribution. Most researchers evaluated their approaches on open-source projects, typically those hosted on Gerrit systems and GitHub. A small part of studies, such as [11], [17], [19], [20] evaluated their approaches on a small number of proprietary software projects. However, little effort is paid in systematically evaluating CRR approaches on proprietary projects in a large company, and the characteristics of these projects have not been studied.

To bridge this gap, we first conduct a large-scale empirical study to investigate the characteristics of proprietary projects at Tencent, and how existing state-of-the-art CRR approaches perform on plenty of proprietary projects in a giant commercial company. Based on the derived findings and implications, we then propose a new approach to better recommend code reviewers in the context of proprietary software development. In this work, we try to answer the following research questions:

**RQ1: What are the characteristics of proprietary projects at Tencent?**

We first identify 300 proprietary projects with the most pull requests at Tencent, and retain 163 accessible projects with more than 1,000 historical code changes. After retrieving their review data, we then clean our datasets (*e.g.*, removing robot reviewers and code changes with no commits) and categorize the projects according to their dominant programming languages, since we find projects written in the same language serve different applications and functionalities. Through the quantitative analysis of our collected datasets, we obtain some valuable results which can support our further study. For instance, we find that most code changes in our collected projects are only reviewed by a single person, while those in open-source projects are often reviewed by several or even a group of project members; most developers tend to work

‡Work done while this author was an intern at Tencent.
§Corresponding author.

978-1-6654-3786-8/22/$31.00 ©2022 IEEE
DOI 10.1109/SANER53432.2022.00080

with their dominant collaborators and a proportion of projects are reviewed by only several developers. We also notice that 9.6% of the reviewers have worked on multiple projects (contributing or reviewing code).

**RQ2: How do existing approaches perform on proprietary projects?**

We evaluate four existing state-of-the-art approaches, *i.e.*, REVFINDER [10], TIE [8], CHREV [11] and COMMENT NETWORK [15] on our collected datasets. The results show that for each approach, performance scores on proprietary projects differ greatly. For example, when COMMENT NETWORK is applied, the mean reciprocal rank (MRR) can range from below 0.3 to nearly 1.0. On most proprietary projects (excluding those written in rare languages), collaboration-based approach COMMENT NETWORK achieves higher scores than other approaches in terms of almost all metrics. COMMENT NETWORK also produces the largest reviewer participation rate for most projects. And TIE, which combines text mining and file path similarity performs better than file location-based approaches REVFINDER and CHREV. We also qualitatively analyze our findings.

**RQ3: Can we propose a new approach to better recommend proper reviewers for proprietary projects?**

On the basis of what we acquire in RQ1 and RQ2, *i.e.*, (i) considering the CollAboration as an important factor and (ii) considering developers' Multiple-Project working experience, we propose a new approach that can fit the context of proprietary development named CAMP. Roughly speaking, CAMP recommend reviewers according to developers' collaboration along with expertise on multiple projects. For collaboration, CAMP builds a collaboration network (a directed graph) of all participants in current and relative projects, then divides the network into several communities. To supplement reviewers' expertise, CAMP leverages an identifier splitting algorithm to extract common information from text and file paths of PRs in multiple relative projects. Finally CAMP builds a term list for each candidate reviewer, containing terms in the content of PRs that he/she has participated in. For an incoming PR, CAMP first recommends reviewers who are in the same communities as the author of the PR. If there are not enough reviewers, CAMP then search the whole reviewer list for proper ones measured by the relevance of their expertise.

We then evaluate CAMP on our collected datasets. The results show that on average, our approach outperforms baseline approaches in terms of all metrics we use. CAMP produces acceptable recommendation results for more projects, that is, fewer projects only obtain very low top-5 accuracy. Additionally, CAMP can increase the participation rate and can therefore boost knowledge distribution and team awareness.

In summary, the contributions of our work are two-fold:

1) To the best of our knowledge, we are the first to investigate the characteristics of proprietary projects, and the effectiveness of CRR approaches on them by a large-scale empirical study involving 163 projects in total. We derive several findings and implications, which provides directions for our refined approach.
2) We propose a new approach based on our empirical study. The new approach named CAMP outperforms other baseline methods on proprietary projects also with larger reviewer participation rate. And we further discuss the direction for CRR approaches.

The remainder of the paper is organized as follows. Section II provides the background about modern code review and code reviewer recommendation. Section III and IV present our empirical study on proprietary projects at Tencent. Section V exhibits the details and evaluation results of our approach. Section VI discusses advantages and limitations of our approach. Section VII provides related work of recommendation systems and Section VIII concludes the paper.

## II. BACKGROUND

### A. Modern Code Review Practice

We briefly describe the workflow of MCR at Tencent, which is similar to that on open source platform such as GitHub. First, a developer *forks* the main repository, and makes some code changes to the forked repository. Then he/she can open a *pull request* to ask for adoption of code changes. Other developers, so-called *reviewers*, will be invited to review code changes automatically or manually and they can give some comments. Some privileged reviewers can approve or reject code changes. Notice that reviewers may also be robots, which do some automated tasks like continuous integration (CI) and sanity check. The approved code changes will be merged into the main repository.

Most development teams at Tencent adopt MCR. To determine code reviewers, developers maintain owner lists and necessary reviewer lists in configuration files of projects, containing developers responsible for reviewing code changes. However, on the one hand, it costs a lot of time and effort to maintain configuration files. On the other hand, developers in maintained lists just have the privilege to manage code repositories (*e.g.*, submitting code changes), but not certainly the proper reviewers. To this end, applying CRR algorithms to proprietary projects is a possible way to improve code review practice.

### B. Code Reviewer Recommendation Approaches

There have been a lot of work related to CRR, which can be possibly implemented on proprietary projects. A lot of approaches are expertise-based. Thongtanunam *et al.* proposed REVFINDER [10] to automatically recommend reviewers by leveraging the similarity of changed files between reviews. Reviewers who have participated in PRs that are similar to the incoming code change in terms of file paths are probably to be recommended. Xia *et al.* [8] proposed TIE, integrating file location-based approach and text mining approach to improve the performance of REVFINDER. Ouni *et al.* introduced a search-based approach REVREC [12], to find proper reviewers based on their expertise and collaboration in past reviews. Rahman *et al.* [13] recommend reviewers considering not only the relevant cross-project work history, but but also the experience of a developer in certain specialized technologies

associated with a pull request. Zanjani *et al.* proposed CHREV [11] that measures a reviewer's expertise by his/her review history.

As for approaches not only considering expertise, Yu *et al.* proposed an approach named COMMENT NETWORK [15], which leverages social relations between contributors and reviewers to build a collaboration network, achieving similar performance as traditional approaches. Al-Zubaidi *et al.* proposed WLRREC [4], which aims at maximizing the chance of participating in a code review and minimizing the imbalance of the review workload distribution by using meta-heuristic algorithm. Rebai *et al.* formulated CRR as a multi-objective search problem to balance the conflicting objectives of expertise, availability, and history of collaborations [21]. Mirsaeedi *et al.* proposed CARROT [19] to mitigate turnover in the development process, which can balance expertise, workload, and knowledge distribution.

## III. ANSWER RQ1: DATA RETRIEVAL AND ANALYSIS

In this section, we present our empirical study on the review data collected from Tencent. We aim at finding the characters of pull requests and how developers participate in code review activities. The derived findings can provide supports for further study.

### A. Data Collection

At Tencent, there are a great number of proprietary projects that are open to internal developers, and these projects desire for the functionality of code reviewer recommendation. We first identify 300 projects with the most PRs, and remove those whose repositories do not exist and those without documentation websites. We then retain 163 projects containing more than 1,000 historical code changes from October 2018 to July 2021.

We group the collected projects based on the primary programming language they are written in. We determine the primary programming language of a project by the most source code files written in it. Since TypeScript extends JavaScript by adding types to the language, we regard TypeScript projects as JavaScript projects. For projects that do not contain source code, typically those only storing static datasets or written in rare languages at Tencent (*e.g.*, PHP, Dart and C#), we group them into *Others* and do not use them in our further discussion.

Table I presents the projects used in our study. JavaScript and Java projects make great proportions (29% and 23% respectively). By the way, most of the JavaScript projects are frontend web applications, and most of the Java projects are Android applications. Table I also exhibits mean and standard deviation of some properties of selected projects, including the number of contributors ($\#Contrib$, the number of reviewers ($\#Rev$) and the number of PR ($\#PR$).

We retain the following properties for each PR:

1) *Review ID*. A unique integer or string.
2) *Created time*. The time when the review was created.
3) *Author or committer*. The author or committer of the review.
4) *Changed files*. Changed files in related commits.
5) *Textual content*. Description and commit messages of a PR.
6) *Reviewers*. Actual reviewers of the PR.

We remove PRs which are not in *merged* or *abandoned* state, since *open* changes may not be fully reviewed and get eventual results. Additionally, Among the obtained pull requests, we remove robot participants (typically those developers whose names end with `bot`) and those reviewers in *pending* state. We also remove those PRs with no reviewer or changed files. Finally, we sort reviews for each project in chronological of their created time.

### B. Pull Requests in Proprietary Projects

Since the high complexity of textual content of PRs, we first concern the number of reviewers and changed files in a PR. We define the following two properties applied to a project:

1) $\#Rev_{pr}$: Mean count of reviewers in a PR, which can be computed as:

$$\#Rev_{pr}(P) = \frac{1}{|P|} \sum_{pr \in P} ReviewerCount(pr) \quad (1)$$

2) $\#Files_{pr}$: Mean count of changed files in a PR, which can be computed as:

$$\#Files_{pr}(P) = \frac{1}{|P|} \sum_{pr \in P} FileCount(pr) \quad (2)$$

where $P$ is a project, and $|\cdot|$ is the number of PRs in a project.

We first compute $\#Rev_{pr}$ for each project. Table II shows the distribution of $\#Rev_{pr}$ for each programming language. Medians of $\#Rev_pr$ is near 1.0, indicating that in most projects, code changes are just reviewed by one person on average, while by two or more developers in a small proportion of projects. This probably results from the review policy at Tencent: *a code change can be merged if one necessary reviewer approves it*. Here *necessary reviewers* are mainly designated by the configuration files of a project.

Compared with Tencent, the situation in other commercial companies is different. Rigby *et al.* [22] has reported that in AMD, the median number of each review is 2, and this value is 3 or 4 in some projects of Microsoft. On the other hand, in open-source projects, developers can participate in reviews they are interested in. For example, on OpenStack[1], a code change is reviewed by over three reviewers on average, and up to 40.

> **Finding 1.** On most proprietary projects at Tencent, code changes are reviewed by only one developer on average.

We then count the distribution of $\#Files_{pr}$, and the result is presented in Table III. All groups' median $\#Files_{pr}$ values

[1]We retrieve code review data between August 2020 and January 2021 from https://review.opendev.org using Gerrit REST API.

TABLE I
DETAILS OF PROPRIETARY PROJECTS USED IN OUR STUDY, GROUPED BY LANGUAGE.

| Language | # Project | $\#Contrib$ (mean $\pm$ std) | $\#Rev$ (mean $\pm$ std) | $\#PR$ (mean $\pm$ std) |
|---|---|---|---|---|
| JavaScript (JS) | 48 | 27.0 $\pm$ 21.8 | 34.0 $\pm$ 24.6 | 1993.8 $\pm$ 1411.3 |
| Java | 38 | 33.0 $\pm$ 36.8 | 41.7 $\pm$ 35.4 | 2998.7 $\pm$ 4620.2 |
| C++ | 19 | 43.5 $\pm$ 42.5 | 52.5 $\pm$ 41.9 | 1819.2 $\pm$ 1149.7 |
| Python (Py) | 9 | 28.8 $\pm$ 29.0 | 39.5 $\pm$ 44.8 | 1306.0 $\pm$ 130.2 |
| Go | 19 | 40.0 $\pm$ 35.7 | 40.3 $\pm$ 36.8 | 2359.8 $\pm$ 1300.9 |
| Objective-C (OC) | 17 | 34.8 $\pm$ 46.7 | 40.8 $\pm$ 44.3 | 4099.1 $\pm$ 7267.7 |
| Others | 13 | – | – | – |

* $\#Contrib$, $\#Rev$ and $\#PR$ stands for the number of contributors, reviewers and PRs, respectively.

TABLE II
STATISTICS OF $\#Rev_{pr}$ (PROJECTS ARE GROUP BY LANGUAGE).

| Language | mean | std | min | median | max |
|---|---|---|---|---|---|
| JavaScript (JS) | 1.22 | 0.34 | 1.00 | 1.02 | 2.29 |
| Java | 1.10 | 0.19 | 1.00 | 1.02 | 1.81 |
| C++ | 1.15 | 0.28 | 1.00 | 1.02 | 2.03 |
| Python (Py) | 1.04 | 0.09 | 1.00 | 1.00 | 1.30 |
| Go | 1.35 | 0.43 | 1.00 | 1.06 | 2.07 |
| Objective-C (OC) | 1.12 | 0.20 | 1.01 | 1.04 | 1.82 |

TABLE III
STATISTICS OF $\#Files_{pr}$ (PROJECTS ARE GROUPED BY LANGUAGE).

| Language | mean | std | min | median | max |
|---|---|---|---|---|---|
| JavaScript (JS) | 9.36 | 4.65 | 3.56 | 7.91 | 29.07 |
| Java | 12.41 | 5.39 | 1.73 | 11.84 | 24.40 |
| C++ | 11.52 | 7.27 | 2.89 | 10.75 | 28.26 |
| Python (Py) | 6.54 | 3.38 | 1.60 | 5.14 | 13.15 |
| Go | 10.79 | 7.51 | 4.11 | 7.90 | 37.32 |
| Objective-C (OC) | 14.66 | 6.67 | 5.72 | 14.66 | 31.21 |

are below 15, which means that only few files are changed in most pull requests. However, among all code changes, 2% contain more than 100 files and are difficult for reviewers to inspect. The largest number of contained files is above 2,000. Some PRs with more than 500 but less than 1,000 files are not very complex in fact (*e.g.*, changing a class name that affects many files with reference to it). Anyway, it is burden for some CRR approaches to deal with these huge PRs.

**Finding 2.** Most code changes do not contain too many files, but a small proportion contain a large number of files.

### C. Reviewer Participation in Proprietary Projects

For characteristics of reviewer participation, we first investigate the diversity of reviewers in a project. We measure the diversity of reviewers ($DoR$) for each project using Shannon's entropy [23], which can be computed as follows:

$$DoR(P) = -\sum_{s \in rc(P)} p(s) \cdot \ln p(s) \quad (3)$$

where $P$ is the target project, $rc(\cdot)$ denotes the collection of existing reviewer combinations of $P$, and $p(\cdot)$ stands for probability of a reviewer combination. For example, if 5% of the code changes in $P$ are co-reviewed by $A$ and $B$, and 4% are reviewed by $A$ only, then $p(\{A, B\})$ and $p(\{A\})$ are 0.05 and 0.04, respectively.

We discover that on each group, $DoR$ values vary from project to project. In our datasets, several projects are reviewed just by one or two developers, therefore their $DoR$ values are relatively small (often less than 1.0). We also look into a proportion of projects (30%) whose $DoR$ values are below 2.0 (30%) and found that those projects usually have dominant reviewers, *i.e.*, reviewed by several participants in turn. Except these dominant ones, others usually just review a small part of code changes.

**Finding 3.** A proportion of projects have dominant reviewers, while other projects engage diverse reviewers.

Since many projects are written in the same language or maintained by the same team, it is interesting to investigate whether they share developers. Among 3,967 human reviewers in all internal projects, 382 have participated in over two projects, and 98 have participated in over four projects. We also find that many of those developers participate in a group of projects. For example, developers $D_1$, $D_2$ and $D_3$ all contribute code or review others' code changes in $P_1$, $P_2$ and $P_3$, we say that $\{P_1, P_2, P_3\}$ are a group of frequently appearing projects. We leverage Apriori algorithm [24] to generate frequently appearing projects, finally resulting in 13 groups of projects (involving 98 projects in total) that share common participants.

**Finding 4.** Nearly 10% of the reviewers have worked in multiple projects.

Since we have observed that some developers tend to review certain colleagues' code changes, or submit code changes that are often reviewed by specific colleagues, we use *dominant collaborator* to describe the phenomenon: if a developer $D_a$ has reviewed more than half of the code changes contributed by developer $D_b$, then $D_a$ is $D_b$'s *dominant collaborator*, and

vice versa. We compute the proportion of developers with dominant collaborators (*DDC*) for each project. The result is that over 50% of the total developers in our collected projects are *DDC*. Here we only count the developers who have contributed or reviewed more than 10 code changes.

**Finding 5.** Over half of the developers have dominant collaborators. Developers tend to invite others they are familiar with to review code changes.

## IV. Answer RQ2: Evaluating CRR Approaches

In this section, we mainly concern whether existing approaches can perform well on proprietary projects, and how differently the existing approaches perform along with the reasons behind.

### A. Evaluation Metrics

To evaluate existing methods on collected projects, we use four metrics widely adopted by recommendation system community, *i.e.*, top-k accuracy, MRR, precision and recall. Compared with our work, studies [8], [10] only consider top-k accuracy and MRR as performance metrics, while other studies such as [4], [11], [15] use precision and recall. These evaluation metrics are described as follows.

**Top-k accuracy** is the percentage of reviews where their ground truth code reviewers are ranked in the top $k$ positions in the returned ranked list of reviewers. It can be calculated as follows:

$$Top\text{-}k\ Accuracy = \frac{1}{|S_{pr}|}\sum_{pr \in S_{pr}} isRecomm(pr,k) \quad (4)$$

where $S_{pr}$ is PRs in testing set, $isRecomm(pr,k)$ denotes whether there exists a correct reviewer for PR $pr$ in the first $k$ positions of the recommendation list.

**Mean Reciprocal Rank (MRR)** [25] is a popular metric used in information retrieval and recommendation system. Given a recommendation result for a PR, its reciprocal rank (RR) is the multiplicative inverse of the rank of the first correctly-recommended reviewer in a recommendation list. MRR can be computed as follows:

$$MRR = \frac{1}{|S_{pr}|}\sum_{pr \in S_{pr}} \frac{1}{rank(pr)} \quad (5)$$

where $S_{pr}$ is PRs in testing set, $rank(pr)$ is the rank of the first correctly recommended code reviewer in the ranked list for $pr$.

**Mean Precision (MP).** Here *precision* is the fraction of recommended reviewers that are correct for a code change, which can be computed as follows:

$$precision@k = \frac{1}{|S_{pr}|}\sum_{pr \in S_{pr}} \frac{|Actual(pr) \cap Recomm(pr,k)|}{k} \quad (6)$$

Mean precision averages such measures for all pull requests in the testing set. In the equation above, $S_{pr}$ is PRs for testing, $Actual(pr)$ is the actual reviewers of the PR $pr$, and $Recomm(pr,k)$ denotes the first $k$ reviewers in the recommendation list of $pr$.

**Mean Recall (MR).** Here *recall* is the fraction of ground-truth reviewers that are correctly recommended, which can be calculated as follows:

$$recall@k = \frac{1}{|S_{pr}|}\sum_{pr \in S_{pr}} \frac{|Actual(pr) \cap Recomm(pr,k)|}{|Actual(pr)|} \quad (7)$$

Mean recall (MR) averages such measures for all pull requests in the testing set. In the equation above, $S_{pr}$ is PRs for testing, $Actual(pr)$ is the actual reviewers of the PR $pr$, and $Recomm(pr,k)$ denotes the first $k$ reviewers in the recommendation list of $pr$.

### B. Baseline Approaches

To answer RQ2, we evaluate four CRR approaches, *i.e.*, RevFinder, cHRev, Tie, and Comment Network on our collected projects. Four approaches are presented as follows.

**RevFinder** [10] is a file location-based approach that recommends reviewers by leveraging the similarity of file paths. For an incoming pull request, developers who have reviewed many files similar to those in the pull request are probably to be recommended.

**cHRev** [11] is also a file location-based approach, which considers reviewers' experience and working time on specific files. cHRev outperforms RevFinder on many open-source projects.

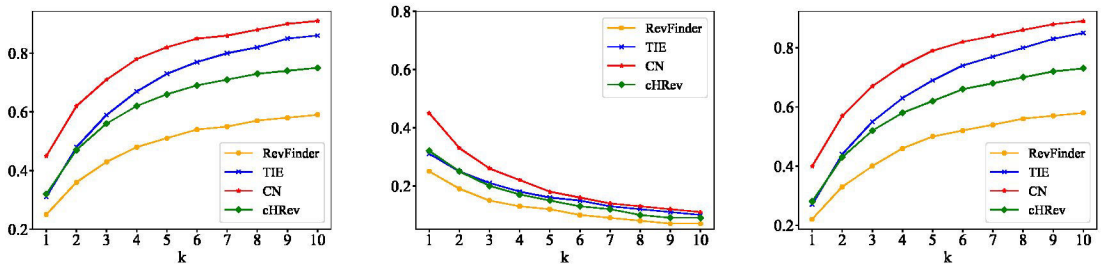
**Tie** [8] combines the file location-based model and text mining model together, and is proved to be one of the most promising CRR approaches, that is, can achieve acceptable performance on all evaluated open-source projects [26].

**Comment Network** [15] is completely a collaboration-based approach that recommends reviewers only considering their collaboration relationship. For a new PR, developers who always review the author's code changes or are the dominant participants of the project are tend to be recommended.

We do not evaluate other approaches that are not expertise-based in our experiments, including workload-aware approach WLRRec [4] and context-aware approach Carrot [19]. The main reason is that these approaches need additional context information, which is somehow difficult to acquire at Tencent, *e.g.*, review workload of each developer (a developer may participate in many development tasks) and the background knowledge of proprietary projects, even though the information may be useful in recommending reviewers.

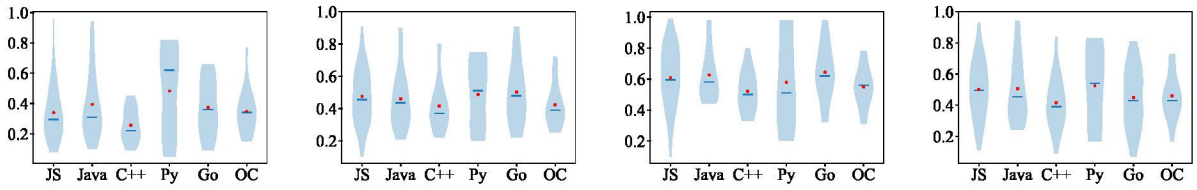
### C. Effectiveness of Existing Approaches

We evaluate four CRR approaches described above on our collected projects. Due to space limitation, we only exhibit the average of the top-k accuracy, mean precision, and mean recall for each approach in Figure 1. Since MRR and top-5 accuracy are more concerned at Tencent (five candidates are provided when the submitter of a code change tries to select reviewers), we also exhibit the distribution of MRR and top-5 accuracy of each group in Figure 2 and 3, respectively.



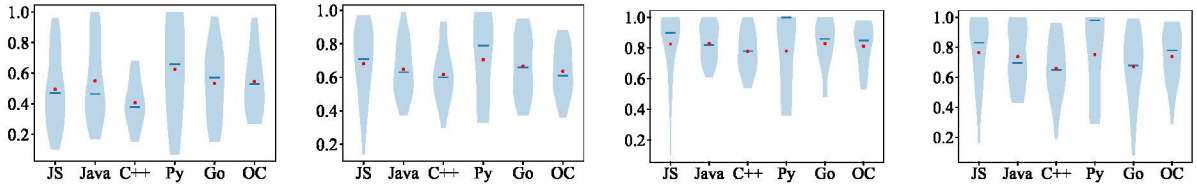

(a) Top-k accuracy (b) Mean precision (MP) (c) Mean recall (MR)

Fig. 1. Averages of top-k accuracy, MP and MR.



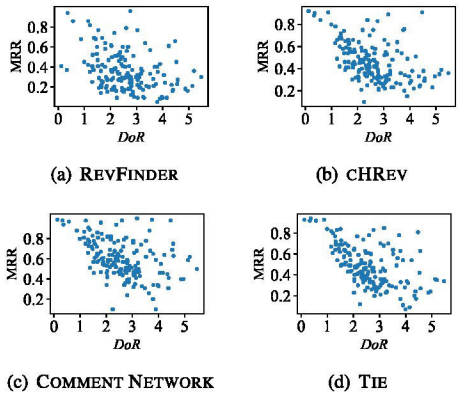

(a) REVFINDER (b) CHREV (c) COMMENT NETWORK (d) TIE

Fig. 2. Distribution of MRR, grouped by language. Red points and blue lines indicate the averages and medians, respectively.




(a) REVFINDER (b) CHREV (c) COMMENT NETWORK (d) TIE

Fig. 3. Distribution of top-5 accuracy, grouped by language. Red points and blue lines indicate the averages and medians, respectively.




(a) REVFINDER (b) CHREV (c) COMMENT NETWORK (d) TIE

Fig. 4. Scatter plot that depicts the relationship between *DoR* and MRR.

From Figure 2 and 3, for most categories, MRR and top-5 accuracy values vary from projects to projects. On JavaScript projects, the difference is more noticeable. It is obvious that on some projects, the top-5 accuracy is still very low (*i.e.*, less than 0.3), which is somewhat not acceptable in practice.

**Finding 6.** Performance of the four evaluated approaches are various among different projects.

To investigate the reason behind various performance metrics, we use intuitive properties of collected projects, *e.g.*, $\#PR$, $\#Contrib$, and the review frequency (measured by submitted code changes per week). We first compute Pearson's correlation between MRR and properties mentioned above. However, the results are very small (absolute values are about 0.01). Then we turn to the relationship between MRR and $DoR$. We present the relationship between reviewer diversity and MRR in scatter plots in Figure 4. Overall, projects with low reviewer diversity can often obtain high MRR values. By contrast, projects with higher reviewer diversity obtain relatively low MRR. We compute the Pearson's correlation

between MRR and *DoR* for each evaluated approach. Correlation coefficients of REVFINDER, CHREV, COMMENT NETWORK and TIE are $-0.37$, $-0.48$, $-0.42$ and $-0.56$, respectively. Overall, MRR and *DoR* are negatively correlated. We mine the three reasons behind, shown below:

1) Nine projects engage only one or two reviewers. Therefore, all kinds of CRR approaches can achieve ideal scores.
2) Some projects, whose *DoR* values fall between 1.0 and 2.0 may have multiple dominant reviewers. Yu *et al.* discovered in their qualitative study that on those projects with multiple dominant reviewers (*i.e.*, where multiple participants review code changes alternatively), expertise-based approaches achieve relatively low scores [15].
3) In some projects, many developers have multiple collaborators. For example, developer $B$, $C$ and $D$ review 30% of code changes submitted by $A$, respectively, causing it relatively difficult to determine which one is to be chosen.

**Finding 7.** Performance scores are overall negatively correlated to reviewer diversity.

*1) Performance Comparison:* Overall, COMMENT NETWORK outperforms other baseline approaches on proprietary projects. Specifically, COMMENT NETWORK achieve the higher scores than other approaches on over 90% of all projects, with 13 projects in Others category excluded. Similarly, on open-source projects, Ouni *et al.* [12] evaluated their collaboration-based approach REVREC on three open-source projects, *i.e.*, Android, QT and OpenStack, and found REVREC outperforms other expertise-based approaches including REVFINDER, CHREV across a number of metrics. They also found that if REVREC only takes collaboration into account, the results are still acceptable. Therefore, it is not too strange that in the context of proprietary software, collaboration-based approach COMMENT NETWORK still performs much better. In our prior findings, there are over 50% of the reviewers have dominant collaborators, which means developers tend to select reviewers with whom they are familiar. It also indicates that proprietary software development may be similar to open-source development in some way. We also apply the Wilcoxon Signed Rank test [27] to our evaluation metrics (including accuracy, precision, recall and MRR) to confirm that the performance of COMMENT NETWORK is significantly better than other approaches, with all $p$-values lower than $10^{-22}$.

As is shown in Figure 1, 2 and 3, TIE achieves higher scores in terms of most metrics than CHREV and REVFINDER, except top-1 accuracy (equivalent to precision@1), which is not very important in practice [13]. We also utilize Wilcoxon Signed Rank test to confirm the performance metrics of TIE (except top-1 accuracy and precision@1) are significantly higher than file location-based approaches, with all $p$-values below $10^{-20}$.

**Finding 8.** Collaboration-based approach outperforms others by a large margin across most metrics, and text mining-based approach TIE completely outperforms file location-based approaches.

Another finding is that when REVFINDER is applied, the performance scores varies by programming languages more obviously. For example, we use Mann Whitney's U test [28] to confirm that the MRR and top-5 accuracy values on C++ projects are significantly less than those projects written in other languages, with all $p$-values lower than 0.05. In the same way, performance metrics on Java projects are significantly lower than those on Python projects. This shows that the performance of REVFINDER is strongly subject to programming languages. By contrast, other approaches are not related to the issue.

**Finding 9.** REVFINDER's performance is more subject to programming languages, compared to other approaches.

## V. ANSWER RQ3: OUR APPROACH

### A. Insights

With help of what we gained from RQ1 and RQ2, we can conclude some directions and targets for our refined approach:

**Insight 1: We'd better consider collaboration of developers as a dominant factor.** On the one hand, we should follow the nature that developers tend to select reviewers they are familiar with. On the other hand, we notice that approaches such as REVFINDER and TIE slow down the recommendation process when they cope with a small proportion of huge PRs, while COMMENT NETWORK do not. What's more, we may supplement developers' expertise in a more proper way, instead of comparing with historical pull requests during a long period.

**Insight 2: We can take into account the multiple-project working experience of a proportion of reviewers.** Nearly 10% of the reviewers have multiple-project working experience. We may extract reviewers' expertise from multiple projects. However, the content of reviewers of different projects are various. Therefore, it is beneficial to extract some general terms from identifiers, including file names and programming tokens in description of a pull request. We can further determine a reviewer's expertise by extracted terms. Also, we need to consider collaboration in multiple projects, since close collaborators may also work together in another project.

### B. A New Approach to Better Recommend Reviewers

Our approach, namely CAMP, consists of *training* phase and *recommending* phase. In training phase, relative projects can be trained together. CAMP builds a collaboration network

from the collaboration history of a seris of relative projects, and then divide the collaboration network into several communities using community detection algorithm. Then CAMP extracts terms from textual content and file paths of all PRs' data in these relative projects, and generate a term list for each candidate reviewer. In recommending phase, CAMP frist recommend the developers in the same community as the author by the weight of the edge between the developer and the author. If there are not enough reviewers, than CAMP extracts terms from the content of a given PR, and computes the relation score of the PR and each reviewer. Then reviewers with high relation scores are recommended.

*1) Training Phase:* Training phase comprises four steps, *i.e.*, constructing relation network, extracting technical terms, removing too common terms and generating term list for reviewers. Steps are described as follows.

**Step 1: Constructing relation network.** Like COMMENT NETWORK, CAMP also builds a relation network from a given project and its relative projects. Two members with collaboration share a undirected edge whose weight is the number of PRs they commonly participate in. If a developer has reviewed another reviewer's code change, or the two developers worked in a common review, we say that they are *with collaboration*. Unlike COMMENT NETWORK, we use undirected graph instead, since the relationship between developers are usually two-way, *e.g.*, if $A$ often review $B$'s code changes, then $B$ will also review $A$'s from time to time. It then divides the network into some communities using *Louvain* algorithm [29] which is one of the widely used community detection approaches. Each community consists of a set of developers.

**Step 2: Extracting terms from pull requests.** We extract items from text and file paths of historical PRs in multiple relative projects. For textual content, we first remove stop words, URLs, email addresses and some unique identifiers, *e.g.*, Commit IDs, Pull Request IDs and related Issue IDs. We do not split words in English dictionary[2]. And for those words that are absent from dictionary and non-English words, we utilize *Samurai* [30] algorithm to split them into simple words. However, we re-define *scoring function* in the splitting algorithm:

$$Score(t) = \alpha\, DocFreq(t) + (1-\alpha)\, RevFreq(t) \qquad (8)$$

where $t$ is the given term, $DocFreq(t)$ is the frequency of $t$ in documentation website of corresponding project, $RevFreq(t)$ is the frequency of $t$ in review datasets, and $\alpha$ is a coefficient, we set it 0.5 in experiments. We crawl HTML content from documentation websites, and only retain plain text. If there is no systematic documentation or the documentation is inaccessible, we ignore it.

For file paths, we split them by trailing slash into file and directory names. Then we also adopt *Samurai* algorithm to split file names into soft words, using the same scoring function defined in Equation 8. After PRs' content has been split into terms, we collect all terms to build a global term list.

**Step 3: Removing common terms.** If anyone of a community $C$ has reviewed a code change containing the term $t$, we say that term $t$ appears in community $C$. Intuitively, if a term appears in most communities, it is probably a general word. If a term appears only in few communities with high frequency, it is possibly a representative technology of a community.

For each term, CAMP computes the number of communities where it appears. We notice that terms such as `from`, `with`, and `commit` probably appear in almost all communities. In other words, these terms can hardly be used to distinguish proper reviewers since almost everyone uses them. Through our preliminary experiments, most terms in a PR appear in less than half of the communities, only a small proportion of terms appear in many communities. Therefore, we remove terms that appear in half of the communities, and finally the rest make up the final term list.

**Step 4: Generating term list for each participant.** For each reviewers, we obtain the PRs in multiple projects he/she has participated in. Among all PRs he/she has participated in, we then count the appearing times of each term in term list obtain from Step 3. Finally this step produces a term list for each participant.

In this way, we do not need to compare the incoming PR with a large number of historical PR records, but only consider reviewers with similar technical terms instead.

*2) Recommending Phase:* In recommending phase, we first recommend reviewers in the same community as the author of a given PR. We rank the reviewers by the undirected weight between them and the author of the PR. If there are not enough reviewers, we then recommend reviewers by their expertise. We regard each incoming PR as a bag of terms, where terms are extracted in the training phase. The relevance score between the incoming PR and a specific reviewer can be computed as follows:

$$Relevance(r, pr) = \sum_{t \in T_{pr} \cap T_r} \frac{Count(t,r) \cdot Count(t,pr)}{CommunityCount(t)} \qquad (9)$$

where $T_{pr}$ is the term list of a PR, $T_r$ is the term list of a reviewer $r$, $Count(t, pr)$ is the times $t$ appears in the PR $pr$, and $Count(t, r)$ is the times $t$ appears in the pull requests which the reviewer $r$ has participated in. $CommunityCount(t)$ denotes the number of community where the term $t$ appears, as defined in Step 3.

### C. Evaluation

We evaluate CAMP on each project's testing set, using the same performance metrics and baseline approaches listed in Section IV-B. Due to space limitation, we only present the average top-k accuracy, precision and recall in Figure 5 and distribution of MRR and top-5 accuracy in Figure 6, respectively. Overall, CAMP outperforms other baseline

[2]We use the dictionary in https://github.com/dwyl/english-words

approaches across almost all metrics. From Figure 6(a), CAMP achieves significantly higher MRR on JavaScript, Java and Go projects than COMMENT NETWORK. As is shown in Figure 6(b), there are less projects that only receive very low top-5 accuracy when recommended by CAMP, that is, CAMP *produces acceptable recommendation results for more projects*. Specifically, there are about 21% of the projects that only receive top-5 accuracy lower than 0.7 when COMMENT NETWORK is adopted, while only 10% when CAMP is applied. Similar to Section IV, we utilize Wilcoxon Signed Rank test [27] (with $p$-values below $10^{-8}$) to confirm that CAMP performs significantly better than existing approaches.

**Observation 1.** CAMP outperforms existing approaches, and can produce acceptable results for more projects.

We further compute the proportion of code changes that receive more accurate recommendation. The results show that up to 29% of pull requests obtain more accurate recommendation results on certain projects. We further discover that over 50% of those PRs with a large number of files get more accurate recommendation by using CAMP. This can prove the effectiveness of considering collaboration and expertise.

We also quantitatively analyze the projects that the existing CRR approaches cannot satisfactorily cope with, but CAMP can produce acceptable results for them, finding that for many code changes, CAMP can recommend more participants close to the contributors, and these participants did not often collaborate with the contributors in the past but have collaborated in other projects. This can confirm the effectiveness of considering multiple-project working experience.

Besides, we adopt *reviewer coverage* ($RC$) to measure the participation rate of reviewers. CAMP can improve reviewer coverage of COMMENT NETWORK, REVFINDER, CHREV and TIE by about 10% (from 73% to 81%), 55% (from 52% to 81%), 8% (from 75% to 81%) and 8% (from 75% to 81%), respectively. CAMP tends to recommend reviewers in the same community and have relative expertise (*e.g.*, having worked on files with a specific token many times). This proves the effectiveness of extracting terms from the content of pull requests. We define $RC$ as follows:

$$RC(P, Rec) = \frac{card(\bigcup_{pr \in P} Rec(pr, k))}{\#Rev(P)} \quad (10)$$

where $P$ denotes a certain project, $Rec$ stands for a recommendation approach and thus $Rec(pr, k)$ denotes the first $k$ recommended reviewers for $pr$, and $\#Rev(P)$ indicates the total number of reviewers in $P$. In our experiments, we let $k$ be 10.

**Observation 2.** CAMP can recommend more diverse reviewers and increase participation rate.

## VI. DISCUSSION

### A. Directions for Code Reviewer Recommendation

**We should not only consider recommendation accuracy, but also actual promotion of software development.** Through our exploratory study and evaluation of our proposed approach, we confirm that our approach can achieve acceptable performance scores on most projects. However, we should also concern *whether reviewer recommendation approaches can distribute knowledge, promote team awareness*. Through our evaluation, our approach CAMP outperforms others in *reviewer coverage*, thus increases the participation rate of internal developers. We believe it is what our approach can benefit internal development.

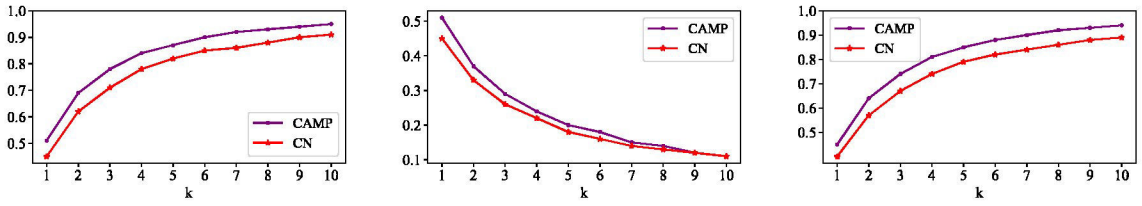
**A wide range of external knowledge should be mined to better recommend reviewers.** Terms in text (description of a PR) and file paths may relate to a wide range of external knowledge, such as documentation of the project and programming conventions. In this work, many proprietary projects lack relative documentation or their documentation is inaccessible. We notice that CARROT [19], a context-aware recommendation approach leverages some metadata, including directory, repositories and file extensions. We suggest that a wider range of knowledge can be considered in CRR, like some recommendation systems in other software engineering tasks (*e.g.*, [31], [32]) making use of knowledge from technical websites like StackOverflow.

### B. Limitation and Threats to Validity

One limitation of CAMP is that it cannot fully outperform COMMENT NETWORK on a small part of projects with a large number of pull requests. The scores in terms of top-k accuracy and precision when $k$ are relatively small. It indicates that for those projects, the expertise of reviewers are difficult to determine. Some pull requests contain too much noise, thus considering expertise factors may lower the performance. On the contrary, although recommending reviewers only by their collaboration cannot achieve very high scores in terms of most metrics, it can yet obtain very high performance on a proportion of projects.

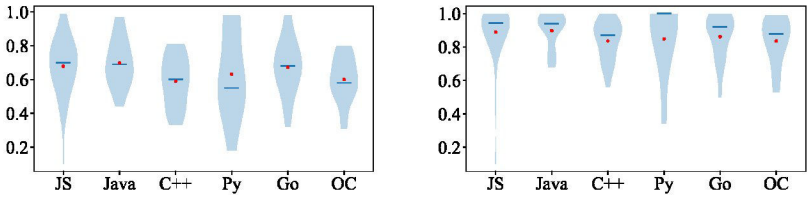
Another limitation of our study is that we do not investigate the universality of our findings and proposed approaches. Although we do empirical study and evaluate our new approach on up to 163 proprietary projects to mitigate threats to external validity, our findings and approach may not be suitable for other proprietary projects at Tencent and other commercial companies. Also, we do not collect many relevant open-source projects, thus we cannot check whether CAMP is suitable for OSS development. It is left for future studies to investigate how CRR approaches perform on proprietary projects in different commercial software companies.

There may also be some errors and biases in our datasets and experiments. First, at Tencent, contributors may make wrong decisions or only receive the default choices when selecting reviewers. Second, we do not collect code changes contributed or reviewed by robots, which may ignore some




(a) Top-k accuracy (b) Mean precision (MP) (c) Mean recall (MR)

Fig. 5. Performance of CAMP compared to COMMENT NETWORK.




(a) MRR (b) Top-5 accuracy

Fig. 6. Distribution of MRR and top-5 accuracy of CAMP. Red points and blue lines indicate the averages and medians, respectively.

critical technical terms. Besides, the identifier splitting method might wrongly deal with out-of-vocabulary words, bringing errors in computing similarity between pull requests.

## VII. RELATED WORK

Recommendation systems (RS) are also widely used in many other software engineering scenarios as well as code reviewer selection. For example, programmers may need library recommenders to effectively use third-party libraries, and API recommenders to look up suitable methods to implement a functionality. Studies in other recommendation systems may benefit CRR research. In this section, we mainly concern library and API method recommendation.

For library recommendation, many approaches leverages collaborative filtering that popular in the field of RS, such as *LibRec* proposed by Thung *et al.* [33] and CROSSREC proposed by Nguyen *et al.* [34]. Besides, Ouni *et al.* formulated library recommendation as a multi-objective search problem [35]. And Chen *et al.* proposed an approach to mine similar libraries making use of the knowledge base extracted from StackOverflow [31].

For API method recommendation, Bruch *et al.* [36] proposed an approach that intelligently suggests method calls considering developing context. Thung *et al.* proposed an API recommendation framework [37] based on both history-based and description-based recommenders. Chan *et al.*'s approach [38] recommends API methods by text phrases as user queries. Knowledge from StackOverflow can also be used in recommendation approaches, *e.g.*, RACK proposed by Rahman *et al.* [32]. Moreover, there exist approaches that recommend code examples given a specific method, such as MUSE presented by Moreno *et al.* [39].

## VIII. CONCLUSION AND FUTURE WORK

Code Reviewer Recommendation (CRR) is an important task for modern software development. Since a large number of submitted code changes, it is necessary to automate CRR to improve development efficiency. In this paper, we first conduct an empirical study on a large scale of proprietary projects from Tencent. We both quantitatively and qualitatively illustrate the characteristics of proprietary projects and the effectiveness of existing CRR approaches. Based on our findings and implications, we propose a new method CAMP that recommends reviewers according to their collaboration and expertise on multiple projects. We evaluate our proposed approach, and the results show that CAMP can obtain higher scores than other baseline approaches across most performance metrics. Additionally, CAMP can mitigate diversity of performance scores on proprietary projects, *i.e.*, offer acceptable performance for more projects. In the future, we also plan to integrate our new approach in development platforms at Tencent and investigate how CAMP works in practice in detail.

## ACKNOWLEDGMENT

The authors would like to thank colleagues at Tencent for providing rich data and technical support. This research is supported by the National Science Foundation of China (No. U20A20173 and No. 6190234).

## REFERENCES

[1] M. Fagan, "Design and code inspections to reduce errors in program development," in *Software pioneers*. Springer, 2002, pp. 575–607.
[2] A. Aurum, H. Petersson, and C. Wohlin, "State-of-the-art: software inspections after 25 years," *Software Testing, Verification and Reliability*, vol. 12, no. 3, pp. 133–154, 2002.

[3] A. F. Ackerman, P. J. Fowler, and R. G. Ebenau, "Software inspections and the industrial production of software," in *Proc. of a Symposium on Software Validation: Inspection-Testing-Verification-Alternatives*. USA: Elsevier North-Holland, Inc., 1984, p. 13–40.

[4] W. H. A. Al-Zubaidi, P. Thongtanunam, H. K. Dam, C. Tantithamthavorn, and A. Ghose, "Workload-aware reviewer recommendation using a multi-objective search-based approach," in *Proceedings of the 16th ACM International Conference on Predictive Models and Data Analytics in Software Engineering*, 2020, pp. 21–30.

[5] F. Shull and C. Seaman, "Inspecting the history of inspections: An example of evidence-based technology diffusion," *IEEE Software*, vol. 25, no. 1, pp. 88–90, 2008.

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

[7] C. Sadowski, E. Söderberg, L. Church, M. Sipko, and A. Bacchelli, "Modern code review: a case study at google," in *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, 2018, pp. 181–190.

[8] 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)*. IEEE, 2015, pp. 261–270.

[9] 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)*. IEEE, 2013, pp. 931–940.

[10] 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)*, 2015, pp. 141–150.

[11] M. B. Zanjani, H. Kagdi, and C. Bird, "Automatically recommending peer reviewers in modern code review," *IEEE Transactions on Software Engineering*, vol. 42, no. 6, pp. 530–543, 2015.

[12] A. Ouni, R. G. Kula, and K. Inoue, "Search-based peer reviewers recommendation in modern code review," in *2016 IEEE International Conference on Software Maintenance and Evolution, ICSME 2016, Raleigh, NC, USA, October 2-7, 2016*. IEEE Computer Society, 2016, pp. 367–377. [Online]. Available: https://doi.org/10.1109/ICSME.2016.65

[13] M. M. Rahman, C. K. Roy, and J. A. Collins, "Correct: code reviewer recommendation in github based on cross-project and technology experience," in *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016 - Companion Volume*, L. K. Dillon, W. Visser, and L. A. Williams, Eds. ACM, 2016, pp. 222–231. [Online]. Available: https://doi.org/10.1145/2889160.2889244

[14] J. Kim and E. Lee, "Understanding review expertise of developers: A reviewer recommendation approach based on latent dirichlet allocation," *Symmetry*, vol. 10, no. 4, p. 114, 2018.

[15] 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*, vol. 74, pp. 204–218, 2016.

[16] H. Ying, L. Chen, T. Liang, and J. Wu, "Earec: leveraging expertise and authority for pull-request reviewer recommendation in github," in *2016 IEEE/ACM 3rd International Workshop on CrowdSourcing in Software Engineering (CSI-SE)*. IEEE, 2016, pp. 29–35.

[17] E. Mirsaeedi and P. C. Rigby, "Mitigating turnover with code review recommendation: balancing expertise, workload, and knowledge distribution," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 1183–1195.

[18] A. Chueshev, J. Lawall, R. Bendraou, and T. Ziadi, "Expanding the number of reviewers in open-source projects by recommending appropriate developers," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2020, pp. 499–510.

[19] A. Strand, M. Gunnarson, R. Britto, and M. Usman, "Using a context-aware approach to recommend code reviewers: findings from an industrial case study," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice*, 2020, pp. 1–10.

[20] S. Asthana, R. Kumar, R. Bhagwan, C. Bird, C. Bansal, C. Maddila, S. Mehta, and B. Ashok, "Whodo: automating reviewer suggestions at scale," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 937–945.

[21] S. Rebai, A. Amich, S. Molaei, M. Kessentini, and R. Kazman, "Multi-objective code reviewer recommendations: balancing expertise, availability and collaborations," *Automated Software Engineering*, vol. 27, no. 3, pp. 301–328, 2020.

[22] 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*, 2013, pp. 202–212.

[23] C. E. Shannon, "A mathematical theory of communication," *The Bell system technical journal*, vol. 27, no. 3, pp. 379–423, 1948.

[24] R. Agrawal, R. Srikant *et al.*, "Fast algorithms for mining association rules," in *Proc. 20th int. conf. very large data bases, VLDB*, vol. 1215. Citeseer, 1994, pp. 487–499.

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

[26] Y. Hu, J. Wang, J. Hou, S. Li, and Q. Wang, "Is there a" golden" rule for code reviewer recommendation?:—an experimental evaluation," in *2020 IEEE 20th International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 2020, pp. 497–508.

[27] F. Wilcoxon, "Individual comparisons by ranking methods," in *Breakthroughs in statistics*. Springer, 1992, pp. 196–202.

[28] H. B. Mann and D. R. Whitney, "On a test of whether one of two random variables is stochastically larger than the other," *The annals of mathematical statistics*, pp. 50–60, 1947.

[29] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre, "Fast unfolding of communities in large networks," *Journal of statistical mechanics: theory and experiment*, vol. 2008, no. 10, p. P10008, 2008.

[30] E. Enslen, E. Hill, L. Pollock, and K. Vijay-Shanker, "Mining source code to automatically split identifiers for software analysis," in *2009 6th IEEE International Working Conference on Mining Software Repositories*. IEEE, 2009, pp. 71–80.

[31] C. Chen, S. Gao, and Z. Xing, "Mining analogical libraries in q&a discussions–incorporating relational and categorical knowledge into word embedding," in *2016 IEEE 23rd international conference on software analysis, evolution, and reengineering (SANER)*, vol. 1. IEEE, 2016, pp. 338–348.

[32] M. M. Rahman, C. K. Roy, and D. Lo, "Rack: Automatic api recommendation using crowdsourced knowledge," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1. IEEE, 2016, pp. 349–359.

[33] F. Thung, D. Lo, and J. Lawall, "Automated library recommendation," in *2013 20th Working conference on reverse engineering (WCRE)*. IEEE, 2013, pp. 182–191.

[34] P. T. Nguyen, J. Di Rocco, D. Di Ruscio, and M. Di Penta, "Crossrec: Supporting software developers by recommending third-party libraries," *Journal of Systems and Software*, vol. 161, p. 110460, 2020.

[35] A. Ouni, R. G. Kula, M. Kessentini, T. Ishio, D. M. German, and K. Inoue, "Search-based software library recommendation using multi-objective optimization," *Information and Software Technology*, vol. 83, pp. 55–75, 2017.

[36] M. Bruch, M. Monperrus, and M. Mezini, "Learning from examples to improve code completion systems," in *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering*, 2009, pp. 213–222.

[37] F. Thung, S. Wang, D. Lo, and J. Lawall, "Automatic recommendation of api methods from feature requests," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2013, pp. 290–300.

[38] W.-K. Chan, H. Cheng, and D. Lo, "Searching connected api subgraph via text phrases," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, 2012, pp. 1–11.

[39] L. Moreno, G. Bavota, M. Di Penta, R. Oliveto, and A. Marcus, "How can i use this method?" in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 880–890.