Mitigating Turnover with Code Review Recommendation: Balancing Expertise, Workload, and Knowledge Distribution

Ehsan Mirsaeedi

A Thesis

in

The Department

of

Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of

Master of Computer Science (Computer Science) at

Concordia University

Montréal, Québec, Canada

September 2019

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: **Seyyed Ehsan Edin Mirsaeedi Farahani**

Entitled: **Mitigating Turnover with Code Review Recommendation: Balancing Expertise, Workload, and Knowledge Distribution**

and submitted in partial fulfillment of the requirements for the degree of

Master of Computer Science (Computer Science)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the Final Examining Committee:

_____ Chair
Dr. Essam Mansour

_____ Examiner
Dr. Emad Shihab

_____ Examiner
Dr. Jinqiu Yang

_____ Supervisor
Dr. Peter C. Rigby

Approved by _____
Dr. Lata Narayanan, Chair
Department of Computer Science and Software Engineering

September 2019

_____
Dr. Amir Asif, Dean
Faculty of Engineering and Computer Science

Abstract

Mitigating Turnover with Code Review Recommendation: Balancing Expertise, Workload, and Knowledge Distribution

Seyyed Ehsan Edin Mirsaeedi Farahani

Developer turnover is inevitable on software projects and leads to knowledge loss, a reduction in productivity, and an increase in defects. Mitigation strategies to deal with turnover tend to disrupt and increase workloads for developers. In this work, we suggest that through code review recommendation we can distribute knowledge and mitigate turnover with minimal impact on the development process. We evaluate review recommenders in the context of ensuring expertise during review, *Expertise*, reducing the review workload of the core team, *CoreWorkload*, and reducing the Files at Risk to turnover, *FaR*. We find that prior work that assigns reviewers based on file ownership concentrates knowledge on a small group of core developers increasing risk of knowledge loss from turnover by up to 65%. We propose learning and retention aware review recommenders that when combined are effective at reducing the risk of turnover by -29% but they unacceptably reduce the overall expertise during reviews by -26%. We develop the `Sofia` recommender that suggest experts when none of the files under review are hoarded by developers, but distributes knowledge when files are at risk. In this way, we are able to simultaneously increase expertise during review with a ΔExpertise of 6%, with a negligible impact on workload of ΔCoreWorkload of 0.09%, and reduce the files at risk by ΔFaR -28%. `Sofia` is integrated into GitHub pull requests allowing developers to select an appropriate expert or "learner" based on the context of the review. We release the `Sofia` bot as well as the code and data for replication purposes.

Acknowledgments

I would like to take this opportunity to show my gratitude towards the people who have played an indispensable role in this memorable journey.

Foremost, I would like to express my sincere gratitude and respect towards my thesis supervisor, Dr. Peter Rigby. This work would not have been possible without his guidance, support and encouragement. His undying patience and guidance have helped me in all phases of this journey, from carrying out research to writing this thesis. Sincerely, I could not have asked for a better supervisor.

I would like to thank Concordia University for providing me with an opportunity to be a part of it.

Last but not the least, I would like to thank my parents and my wife for their love and constant support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Turnover on software projects is frequent and inevitable and leads to the loss of knowledge when developers leave a project [16, 78]. Turnover incurs substantial economic cost in recruiting and training new employees [59, 64], it reduces the productivity of development teams [39, 59], it leads to the loss of critical tacit knowledge [38, 58, 59], and has been shown to increase the number of defects in a product and reduce overall product quality [30, 58, 59].

Recent works have tried to mitigate the adverse impact of turnover through increasing knowledge retention by predicting leavers [16, 25, 45], planning for succession [58, 63, 78, 86], documenting knowledge, and persisting knowledge on StackOverflow and other internal QA forums [63, 70]. However, these mitigation practices often require organizational changes and additional developer effort especially by those who are expert enough to answer questions and write documentation [70].

In this work, we show that code review can mitigate turnover risk because it naturally distributes knowledge by exposing developers to new code during reviews. Prior work interviewed developers and showed that code review is an opportunity for learning and it plays a vital role in distributing knowledge [14, 22, 35, 75, 80, 87]. Furthermore, studies have quantified the knowledge gained during code review [73, 80] and shown that developers review code in modules they have not modified [87]. In contrast to other turnover mitigation strategies, code review is a common and well-established practice in teams that does not require teams and individuals to alter their current workflow.

In this work, we enhance code review's inherent knowledge sharing potential by developing review recommenders to distribute knowledge and use simulations to show that they mitigate turnover

risk. In contrast, existing review recommenders [15, 37, 40, 69, 89, 92, 94, 95] are solely focused on finding expert reviewers and disregard the role of code review in distributing knowledge among developers. These recommenders result in expertise concentration because the evaluation benchmark is how many of the actual developers who performed the review were recommended. Interviewed developers state that these recommenders suggest obvious candidates and do not provide additional value [42].

To evaluate recommenders from other perspectives, we introduce three outcome measures that interviews with developers indicated as important aspects of code review [14, 35]: *Expertise*, *Core-Workload*, and *FaR*. The first outcome ensures that expertise remains high for finding defects during review. The second, ensures that the core developers are not unreasonably overworked due to always being the top recommendation. The third outcome measures the number of files that are at risk to turnover, *FaR*, to ensure that knowledge is adequately distributed during review. We run simulations on the historical reviews of five large projects to understand how recommenders affect each outcome. For completeness, we also calculate Mean Reciprocal Rank, MRR, to understand how well each recommender predicts the developers who actually performed the review.

Research Questions

RQ1, Review and Turnover: What is the reduction in files at risk to turnover when both authors and reviewers are considered knowledgeable?

Recent studies have quantified knowledge loss from turnover on the basis of the commits that each developer has authored [61, 78]. However, the knowledge transfer that occurs during code review is widely documented with prior work showing that review promotes team awareness, transparency, and shared code ownership [14, 75, 80, 87]. We modify the previous turnover measure to consider both authors of code as well as reviewers to be knowledgeable and recalculate the number of files that are at risk, *FaR*. With only authors being considered knowledgeable on average 37% of the total files are at risk to turnover. When we consider both authors and reviewers to be knowledgeable *FaR* drops to 15%. Code review naturally distributes knowledge.

RQ2, Ownership: Does recommending reviewers based on code ownership reduce the

number of files at risk to turnover?

Studies show that teams tend to assign reviews to the owners of files under review [35, 80] and experts who have modified or reviewed the files in the past [15,42]. We implement simple ownership recommenders that suggest reviewers based on the files that developers have modified or reviewed in the past.

We show that assigning reviewers based on prior commits, *AuthorshipRec*, or prior reviews, *RevOwnRec*, increases expertise by 11.29% and 15.17%, respectively, while increases turnover risk, *FaR*, by 25.25% and 65.19%. We conclude that concentrating expertise on the top developers make projects susceptible to knowledge loss from turnover.

RQ3, cHRev: Does a state-of-the-art recommender reduce the number of files at risk to turnover?

We review the literature on review recommenders and find that most mine historical review information. Unfortunately, we did not find working implementations or replication packages for any of the existing recommenders. For comparison purposes, we re-implement cHRev which has been shown to outperform other recommenders [95]. When re-evaluate *cHRev* on our outcome measures, we find that like the ownership recommenders, cHRev increases the level of expertise by 11.11%, and has the added benefit of reducing *CoreWorkload* by -3.49%. Unfortunately, cHRev concentrates knowledge and increases the risk of knowledge loss through turnover by 4.15%.

RQ4, Learning and Retention: Can we reduce the number of files at risk to turnover by developing learning and retention aware review recommenders?

We propose two knowledge aware proxies for estimating knowledge distribution and retention. *LearnRec* ensures that a developer who has not reviewed or modified all of the files currently under review will be proposed. *RetentionRec* recommender ensures that non-transient developer who have commitment to the project are recommended. Assigning learners through *LearnRec* substantially reduces *Expertise*, -35.13%, but counter-intuitively it makes the project drastically more susceptible to knowledge loss from turnover as less committed developers are recommended, ΔFaR of 63.04%. Suggesting committed developers through *RetentionRec* is the most successful strategy in ensuring experts, 16.59%, during review, but has the greatest increase in *CoreWorkload*, 29.42%.

RQ5, `Sofia`: Can we combine recommenders to balance *Expertise*, *CoreWorkload*, and

FaR? Each of the previous recommenders has a focus and cannot simultaneously balance the outcomes. Our final recommender, `Sofia`, assigns either experts or learners based on the files under review. It uses cHRev when the files under review are not at risk and uses *TurnoverRec* when few developers know about the files under review. This multi focus strategy improves all outcomes simultaneously. `Sofia` increases the level of expertise during review by 6.27%, while having a minor impact of 0.09% on *CoreWorkload* and reduces turnover risk with a ΔFaR of -28.27%.

We integrated `Sofia` to make recommendations for GitHub pull requests and to recommend both expert and "learning" developers. The `Sofia` source code [57] is publicly available along with the data in a replication package [56]

This thesis is organized as follows. In Chapter 3, we provide the study background as well as defining our measures, review recommender, scoring functions, and simulation methodology. In Chapter 4, we describe the projects under study. In Chapter 5, we present results for each of our research questions. In Chapter 5.6, we describe the `Sofia` bot which integrates into GitHub pull request. In Chapter 5.7, we discuss threats to validity. In Chapters 6 and 6.4, we discuss our findings in the context of the existing literature and conclude the thesis.

Chapter 2

Survey of the Literature

The goal of this thesis is to spread knowledge during code review to mitigate knowledge loss from turnover through reviewer recommendation. Since a key objective of code review is to find defects, we also need to ensure that experts who have authored and reviewed the files in the past are recommended. We break our survey of the literature into five categories.

- Turnover
- Code Ownership
- Expert Recommenders
- Code Review
- Reviewer Recommender

2.1 Turnover

During the evolution of a project, the contributing team evolves as employees join, leave, or change roles. Turnover refers to the phenomenon of continuous cycle of hiring and departing employees. Previous research has unveiled the impact of turnover on the productivity of teams and the sustainability of companies and projects in a wide variety of industries. The most common perspective is that turnover of employees negatively impacts the performance of organizations [38]. Departures damage the established social network and work environment [26] and deprive the project of

leaver's experience and knowledge [38, 90]. A contrasting perspective is that turnover is beneficial for sustainability of projects by allowing removing negative or dissatisfied employees [43]. Furthermore, studies show that new employees bring novel ideas, knowledge, and experiences which helps with renewing the knowledge base of team and preventing stagnation [84].

In software engineering field, turnover is perceived as on of the biggest source of risk by managers [21]. Prior works have examined the reasons why developer leave, developed patterns of turnover, and estimated the risk that projects face.

2.1.1 Turnover Patterns

Lambert *et al.* [44] find that workers with high job satisfaction are less likely to leave. For instance, employees who were involved in the early stages of the project have have higher retention rates and job satisfaction. These employees tend to have more flexibility in determining their organizational role [45, 81]. Martensen *et al.* [48] show that roles that fulfill the employees' need for creativity and innovation experience have lower turnover rates. Constantinou and Mens [25] attempt to predict why developers leave a project. They find that developers who do not have strong technical and social activity intensity, as measured by commits and engagement in discussions, are more likely to leave. Similarly, according to Sharma *et al.* [83] past activity, age, and size of a project as well as developer tenure are important factors in predicting leavers.

2.1.2 Turnover Impact

The impact of turnover on software engineering projects has been the focus of prior studies. A survey with practitioners show that controlling turnover is necessary for the success of industrial projects [36]. On the contrary, Foucalt *et al.* [30] find that a high turnover rate in open source systems does not negatively affect the success of projects while it may increase defects in modules. Mockus [59] finds that departures of developers increase the number of defects. However, he also found that newcomers are not culprits the increased defects because they are not assigned to important tasks.

Izquierdo *et al.* [39] propose a quantitative method for measuring the knowledge loss due to developer turnover. They investigate the evolution of orphaned code lines authored by developers

who left the team. They show that while in some projects, developers maintain code introduced by leavers, in others, they seek to remove the orphaned code.

Studies on Truck Factor quantify the severity of knowledge concentration by measuring turnover-induced knowledge loss as the percentage of files without active owners [72, 96]. The truck factor measures the worst case loss, not its likelihood. Rigby *et al.* [78] use a Monte Carlo simulation to estimate the worst impact of knowledge loss and its likelihood. Using file abandonment, they quantify the risk of developer turnover by adapting the Value at Risk from financial risk management and calculate probability that a certain level of loss will be occurred, *i.e.* Knowledge at Risk. Through historical simulation they find that projects are susceptible to turnover-induced losses that are more than five times larger than the expected loss. The methodology of Rigby *et al.* in computing knowledge loss have been replicated in other studies [61, 79].

To mitigate the impact of knowledge loss from turnover and promote knowledge sharing, proactive knowledge exchange mechanisms have been suggested, such as using knowledge management systems and succession planning [58, 63, 70]. For example, Rigby *et al.* [78] develop a succession measure that ranks possible successors based on the number of files that they have authored that have co-changed with abandoned files. Their approach can recommend correct successors 34% to 48% of the time. Mockus [58] propose different measures for investigating the impact of succession on organizations. Based on the chronology and traces of developer activities that are available through the version control and organization directory systems, they show larger projects, overloaded mentors, and offshoring succession substantially reduce the productivity ratio.

2.2 Code Ownership

The research literature contains substantial works on understand and quantifying code ownership for both empirical understand of development as well as task and review recommendation [19, 31, 60, 62, 87].

As teams and codebases grow, a controversial question arises [34]: should each of the team members own a specific part of the system? Or should the team collectively own everything? Code ownership reflects the notion of responsibility. In other words, it indicates how many developers are

currently contributing to the code and whether there is a single individual who can be considered the primary contributor.

The collective code ownership model has been advocated by the Agile community [17] where schedules and responsibilities are set such that all developers have a chance to work across different modules [62]. Advocates of the collective code ownership model claim that allowing developers to work on any part of the code base 1) increases the agility and productivity of the team [2] 2) increases the chance of catching defects because of the number of collaborators [71] 3) distributes knowledge and reduces the truck factor.

In contrast to the belief of agile practitioners, several researches provide evidence suggesting that having many distinct authors on code artifacts leads to unfocused and defect-prone contributions [20, 66].

2.2.1 Measuring Code Ownership

The existing literature uses expertise as a proxy for measuring ownership. Proposed code ownership heuristics are measured at different granularities. McDonald and Ackerman [49] find that developers use previous experience as the primary criterion for determining experts. They report that developers often look into the code change history to identify who has contributed to a particular file and assume the last person to make a change is most likely the expert. The "Line 10 Rule" and Git blame allow developers to determine the last developer that changed each file and line, respectively [3].

Mockus and Herbsleb [60] measure expertise based on the number of tasks a developer has completed. They define experience atoms to measure several types of expertise based on technology, change type, and software version. They extract data from software management systems to identify a developer's work items. Then, by linking code artifacts to work items they calculate the type of expertise a particular developer has about each part of the system.

Bird *et al.* [20] define ownership at the module level. The ownership of a module is assigned based on the number of commits by a developer relative to the total number of commits made to the module. Their assumption is that each exposure to a code artifact is a learning experience and extends the developer's knowledge and expertise. They consider a developer to be a major

contributor of a module if his or her ownership ratio is above 5%. Greiler *et al.* [35] uses the same ownership definition as Bird *et al.* but at the level of files instead of large modules. They also change the ownership threshold to 50%.

Rahman and Devanbu [68] measures ownership as the number of lines authored by a specific developer. They define authorship ratio of developer a in buggy code b to be the number of lines authored by a over the total number of lines of b. Similarly, Gibra *et al.* [32] define file ownership as the percentage of lines owned by developers.

Schuler and Zimmermann [82] introduce usage expertise as number of commits with code that use a particular module by calling its APIs. The intuition behind their work is that developers also accumulate expertise by calling methods because they know when and how a method can be used.

There are other studies that go beyond code authorship in order to measure ownership. Fritz *et al.* [31] estimate expertise using the Degree of Knowledge model which is built upon the authorship and the interactions of developers with code elements. Their heuristic has four major components which are first authorship, number of deliveries, acceptances, and developer interactions through an IDE.

Rigby and Bird [75] measure expertise of a developer by counting the number of files they have modified or reviewed in the past. Thongtanunam *et al.* [87] report that in addition to authoring changes, developers greatly contribute to the evolution of code artifacts by reviewing code. Drawing on this idea, they adapt the popular commit-based code ownership heuristics of Bird *et al.* [20] to be review aware. They define review specific ownership ratio of developer D in module M to be the number of reviews D has done over the total review contributions made to M.

2.3 Expert recommenders

In the previous section, we gave the background on ownership. In this section, we discuss how ownership and expertise can be used to recommend developers. Everyday a hundreds of new bugs are submitted that need to be traiged and fixed quickly [91]. Finding relevant experts who have enough knowledge and experience to effectively fix bugs and implement new features is difficult on large teams. In OSS projects, bug report triagers are responsible for manually analyzing tasks and bug reports to developers [49]. Manual diagnosis takes time and effort and the assignment is not

always precise because the root cause is not always clear to triagers. Expert recommenders have been proposed in the literature to automatically assign bugs to experts [49].

McDonald and Ackerman [49] propose Expertise Recommender as a general software architecture suitable to locating expertise in a range of different collaborative domains.

Mockus and Herbsleb [60] propose Expertise Browser (ExB) that uses data from issue management systems to find people with desired expertise. Builting upon experience atoms, users of the tool can see how expertise is distributed over the system. ExB facilitates finding developers with either broad or specific expertise.

Anvik *et al.* [13] present a semi-automated approach to the assignment of a bug reports with a precision between 50% and 64%. They train a supervised machine learning model using textual information available from summary and description fields of existing bug reports.

Kagdi *et al.* [41] produced a tool called xFinder to recommend a ranked list of developers who have sufficient expertise fix a bug or perform a development task. xFinder mines the information available in version control systems to find the potential experts for a specific file or package using the number of commits, recency, and workdays of developers. xFinder's accuracy is between 43% and 82%.

Wu *et al.* [91] present a two layer approach called DREX to recommend a group of developers who may have potential expertise in resolving reported bugs. When a new bug report arrives, they employ the K-Nearest-Neighbor search technique to find similar historical bug reports. They rank candidates using the frequency of comments and social relationships in similar historical bugs.

Linares *et al.* [46] propose a technique that does not need to mine historical information of version control and issue tracking systems. Using an information retrieval technique, they query the codebase to locate the relevant files from the textual description of a bug report. Next, potential candidates are ranked based on the number of relevant files they have authored. In this approach, authors are found in the header of files instead of searching through commit history. They report their approach is on a par with xFinder [41] and a previously ML technique published by Anvick *et al.* [13].

Xia *et al.* [93] propose a composite method named DevRec to score candidates based on two analyses. First, in the bug report analysis, it finds the most appropriate developers using a multi-label

ML-KNN technique which finds the K-nearest bug reports. Second, in developer analysis, it scores developers of similar bug reports based on their distance to the newly submitted bug. Developer distance is computed based on the characteristics of the bugs a developer has resolved. DevRec is shown to outperform DREX by improving the average recall@5 and recall@10 scores by 165.38% and 89.36%, respectively.

Shokripour *et al.* [85] provide a simple yet effective two-phase approach to recommend developers. They use textual information of code files, commit messages, comments, and previously fixed bugs to construct an index mapping between nouns and files. When a new bug report is submitted, possible affected files are located based on a simple term-weighting scheme and the previous developers are selected according to "Line-10 Rule". They achieve an accuracy of 89.41% and 59.76% when recommending five developers for the Eclipse and Mozilla projects, respectively.

2.4 Code Review

Code review is an effective quality assurance strategy that identifies defects [14, 27, 73, 75, 80] and also spreads knowledge among the developers involved in a review [14, 22, 35, 75, 80]. Our goal is to reduce turnover risk through review. In this section, we survey the code review literature.

Code review is a common quality assurance mechanism employed in both open source and industrial settings. The proliferation of code review tools such as Gerrit, CodeFlow, and GitHub have strongly promoted the less formal modern code review [73, 75] which is a lightweight but efficient replacement for formal software inspection meetings prescribed by Fagan [27]. The asynchronous nature of modern code review forms an organic, developer driven, community-wide collaboration mechanisms that is easily scalable in large distributed teams [77].

2.4.1 Empirical studies on Code Review

Formal software inspections formalized by Fagan [27] have been perceived as a valuable method to improve the quality of industrial software projects. The formal inspection prescribe periodic meetings where the manager is responsible for assigning experts to be involved in reviews and participants are expected to study the change before gathering to discuss it. The single goal of

inspection meetings is to find defects. Software inspections effective in finding 60%-90% of defects, as well as educating developers to avoid making same mistakes [29].

Porter *et al.* [67] create statistical models to understand the factors that effect the efficacy of code review. They report that the expertise of participants along with the change size and meeting intervals are an important predictor of review effectiveness. Their result show that two reviewers perform as well as a larger group. Another study conducted by Perry *et al.* [65] state that synchronous meetings are indeed unnecessary because meeting-less inspections discover the same number of defects.

Rigby *et al.* [76] report that code review in open source environments and formal inspection meetings as envisioned by Fagan in 1976 [27] have little in common. Using the data gathered from 25 successful open source projects, they create statistical models of review efficacy and found that instead of having rigid and formal policies, open source review practices evolved organically based on the needs of the team. Open source teams embrace code reviews that are conducted asynchronously, empower experts, provide timely feedback on small changes, and give developers enough autonomy to work on their area of expertise.

Rigby *et al.* [74] show that in successful open source projects code review is conducted early and frequently by a small group of self-selected experts to ensure the quality proposed changes which are small and independent. Interestingly, they observe that discussion are not necessarily localized. In fact, a large proportion of the reviews discussed the abstract or global implications of the contribution.

Rigby and Bird [75] report that despite differences among projects, many of the characteristics of the review process have independently converged to similar values which may indicate general principles of code review. For instance, having two reviewers is enough to find the optimal number of defects, change size should be small, and reviews need to be done early and frequently. Also, They identify a new trend in software teams moving toward employing a review tool instead of email-based communications. Important to our research, they report the focal point of reviews is shifting from a defect finding activity to a group problem solving activity in which the author and reviewers discuss the code and design decisions in depth. This knowledge sharing practice is shown to substantially increases the number of files that developers know about by 100%, 122%, 155% in Bing, Office, and SQL, respectively. Another study conducted by Sadowski *et al.* [80] confirms

these convergent practices on Google projects. At Google, knowledge spreading is part of the educational motivation for code review. Furthermore, they show that the number of files developers learn through reviews is close to the number of files they learn about through authorship. Furthermore, the workload of expert reviewers and temporary absences are reported to be an ongoing concern at Google.

Rigby and Storey [77] conduct a qualitative study to investigate the broadcast-based reviews practiced in open source communities. Among their findings, they report that developers decide to review a patch according to their area of expertise and their relation with the author of the change.

Bacchelli and Bird [14] study Microsoft developer and report that while finding defects and improving code quality are the main motivation for code review, there are other side benefits such as knowledge transfer, share code ownership, and team awareness. Distribution of knowledge helps developers learn about different parts of the code and ensures that more than one person has expertise.

Another large-scale study conducted by Greiler *et al.* [35] shows that authors of changes are concerned with 1) getting feedback in timely manner 2) finding appropriate or willing reviewers. On the other hand, reviewers state that it is hard to find time for performing requested reviews because understanding the change's purpose and motivation is difficult and time-consuming. Similar to the findings by Rigby and Bird [75] study participants explicitly recommended using two reviewers. They suggest reviewers should be selected based on code expertise, ownership, or educational purposes. They also note that feedback from experts can lead to delays from lack of availability and also fewer fewer opportunities for knowledge dissemination. They conclude that requesting less experienced reviewers can increase review speed and balance the teams workload.

Bosu *et al.* [23] find that reviewers who have changed or reviewed a file before provide more useful feedback. In addition, they report that larger changes receive less comments that will be of value to the author of the change. In another study, Bosu *et al.* [22] find that while open source contributors see code review as an opportunity for impression formation, Microsoft developers consider knowledge dissemination to be more important. Their survey results show that eliminating functional defects is only the third most important reason for code reviews, after maintainability and knowledge sharing. Reviewers state that the reputation of author and their relationship is an

important factor for accepting a review request. Moreover, reviewers prefer to review code changes that are closely related to their areas of work or expertise in order to minimize review time.

2.5 Reviewer Recommenders

Identifying the right reviewers for a given change is a critical step in the code review process [14, 27, 35]. Inappropriate selection of reviewers slows down the review process [89] or lower the quality of inspection [14]. Reviewer recommenders consist of an algorithm that suggests the optimal reviewers for a given change. The expertise of candidates with code under review is the main proxy for estimating suitability, which is measured through analysis of developers prior code changes and review participation. The performance of existing reviewer recommenders have been measured by evaluating how accurate these approaches can produce recommendations that match actual reviewers of historical reviews. This evaluation is based upon the assumption that actual reviewers were among the best candidates to review a change [42].

An early recommender was proposed by Jeong *et al.* [40] to help authors of changes find best reviewers. They build a Bayesian ML-Based recommendation model using a variety of features such as file name, module name, file keywords, author, and bug priority. Their model achieves an accuracy between 66%-70% in top 5 recommendations. A feature sensitivity analysis reveal that the author of the patch and file names are the most dominant features in constructing the prediction model.

Balachandran [15] proposes a reviewer recommender tool called ReviewBot which finds relevant experts based on the lines of code they have reviewed in the past. For each modified line in a new review request, ReviewBot assigns a score to previous review requests which changed that line. Then the score is propagated to all developers involved in reviews. A decay factor is incorporated to reduce the influence of older review requests. The top 5 accuracy of ReviewBot is between 80.85% and 92.31%.

Thongtanunam *et al.* [89] propose a file location-based recommender called RevFinder based on the assumption that files and folders with similar paths are maintained by a similar set of developers. When a new review requests arrives, they calculate how its file paths are similar to previously

reviewed files paths. Using four different string comparison techniques, RevFinder assigns a score to participants of previous review requests. RevFinder has a top 10 accuracy of 79%.

Xia *et al.* [92] propose a hybrid approach called TIE which extends RevFinder by constructing a text mining model that is trained using the textual content of the description field of review requests. TIE achieves average top 5 and top 10 of 79% and 85%, respectively.

Yu *et al.* [94] present a reviewer recommender which suggests reviewers based on their social relations. The intuition behind their work is that developers who share common interests have more commenting interactions together. By mining historical comments, they construct a weighted graph called Comment Network (CN) to model social relations. They find that the CN based model performs as well as the file location based approach which achieves a precision and recall of 26.5% and 77% of recall for top 10 recommendation. However, a mixed approach of CN model and ML-based bug triaging model can achieve 33.8% of precision and 79.0% of recall at top 10 recommendation.

Lipcak and Rossi [47] conduct a large-scale study to compare the performance of RevFinder [89] and the Bayesian based recommender devised by Jeong *et al.* [40]. Their result reveals that no algorithm can outperform the other one for all 51 studied projects.

Zanjani *et al.* [95] propose cHRev which uses the frequency and recency of reviews along with the number of work days as three metrics for ranking expert reviewers. They consider the number of comments a reviewer has made to a file as a proxy for their expertise about that file. To measure effort, they count the number of work days that a developer has devoted to reviewing a file. They also give recent reviews a higher weight. According to their results, cHRev outperforms RevFinder and ReviewBot in terms of recall and precision. cHRev achieves a precision between 29% and 41% and a recall between 67% and 87% for top 5 recommendation. Moreover, they find that inclusion of authorship expertise in reviewer recommender model may adversely affect the precision and recall values.

Kovalenco *et al.* [42] conduct a series of interviews and surveys at Microsoft and Jetbrains to understand if reviewer recommenders help in practice. They find that although recommendations are perceived as relevant, they rarely provide value to developers, *i.e.* the recommended reviewers are obvious. They argue that recommenders should consider a broader set of metrics. For instance, reviewer recommenders could optimize for other goals such as distributing knowledge by

not solely focusing on experts, but also by recommending less experienced people. Similarly, Sadowski *et al.* [80] reports that Detecting the right reviewer is not problematic in practice at Google because ownership is strong in Google's code base.

In the next chapter we build upon the measures discussed in the literature to develop a review recommender that is both developer expertise, developer workload, and turnover aware.

Chapter 3

Background and Definitions

In this section we introduce the background on ownership, review recommenders, and knowledge loss and show the manner in which each has been quantified in the past. We will subsequently use these measures as the basis on which to expand reviewer recommendation in a scoring function that will also be knowledge aware.

3.1 The Ownership Recommenders

The influence of code ownership on code quality has been extensively investigated in the literature [19, 20, 30, 68, 87]. Ownership is a human factor that helps with finding knowledgeable developers that can be accountable for a particular part of code or task [60]. *Developer Recommenders* use ownership to automatically assign tasks to experts [41]. Researchers have used a wide range of granularity, from lines [31,32,68] to modules [19,20], to estimate ownership of developers. Studies on code review find that code owners are usually selected to review changes [14, 35, 80]. In this work we develop two simple scoring functions for review recommendation based on ownership.

***AuthorshipRec*.** Bird *et al.* [19, 20] defines the code ownership for a developer in a module as the ratio of commits the developer has made relative to the total commits made to that component. Our *AuthorshipRec* scores a developer, D, as a candidate reviewer based on the number of commits he or she has made to the files under review, R, divided by the total number of commits made to these files.

$$\text{AuthorshipRec}(D, R) = \frac{\text{CommitsForFilesUnderReview}(D, R)}{\sum_{d}^{Devs} \text{CommitsForFilesUnderReview}(d, R)} \quad (1)$$

RevOwnRec. Thongtanunam *et al.* [87] devise a review aware ownership metric based on the files that a developer has reviewed. Intuitively, reviewers who have reviewed the changed files or modules in the past, will be good candidate reviewers. To recommend reviewers, we score the number of times a candidate has reviewed the files in the past divided by the total number of times the files have been reviewed.

$$\text{RevOwnRec}(D, R) = \frac{\text{ReviewsOfFilesUnderReview}(D, R)}{\sum_{d}^{Devs} \text{ReviewsOfFilesUnderReview}(d, R)} \quad (2)$$

3.2 The *cHRev* Recommender

There is a large literature on review recommendation [15,37,69,89,92,94,95]. We note that we did not find a replication package or recommender implementation for any of these works [47]. We only re-implement cHRev [95] because it includes a wide range of factors in its recommendation and has a higher accuracy than the other review recommenders such as RevFinder [89].

cHRev scores candidate reviewers by the expertise, frequency, and recency of their past reviews. First, cHRev takes the number of comments made by a candidate on a file as a proxy for expertise. Second, cHRev considers the number of work days a developer has worked on a file as a proxy for measuring effort. Third, cHRev weights recent reviews more highly.

cHRev defines the $xFactor(D, F)$ as the measure of the expertise for a developer D on a file F. C_f, W_f, and T_f respectively show the number of review comments contributed by D for F, the number of work days D has dedicated on contributing comments on F, and the last day that D worked on F. To provide a denominator, $C_{f'}$,$W_{f'}$, and $T_{f'}$ indicate the total number of comments made on F, the total number of work days spent on commenting on F, and the time of the most recent comment on F, respectively.

$$xFactor(D, F) = \frac{C_f}{C_{f'}} + \frac{W_f}{W_{f'}} + \frac{1}{|T_f - T_{f'}| + 1} \quad (3)$$

To compute the score of a candidate reviewer for a given code review, they sum up the $xFactor(D, F)$ that the candidate, D, has on the files in the change, F.

3.3 The Turnover Mitigating Recommenders

The focus of existing recommenders on experts disregards the other benefits of code review such as knowledge sharing. Rigby and Bird [75] report that code review increases the number of files developers see by between 100% and 150%. In this work, we speculate that code review can be effective in mitigating the turnover-induced knowledge loss. Based upon this idea, we design reviewer recommenders that either distribute or retain knowledge.

3.3.1 Distributing Knowledge

We then define a candidate's knowledge of review request as the number of files under review that a candidate has modified or reviewed in the past divided by the total number of files under review.

$$\text{ReviewerKnows}(D, R) = \frac{\text{NumCommitOrReviewedFiles}(D, R)}{\text{NumFilesUnderReview}(R)} \tag{4}$$

Equation 4, assigns developers with knowledge of the code under review and ensures expert opinions but concentrates the knowledge of these files exacerbating the risk from turnover.

LearnRec. To distribute knowledge among the developers, we inverse the ReviewerKnows(D, R) function to understand how many new files a developer will gain knowledge of if he or she is assigned the review. We limit the recommender to only display candidates that know about at least one file under review. We then score the remaining reviewers using the *LearnRec* recommender to maximize learning through the scoring function:

$$\textit{LearnRec}(D, R) = 1 - \text{Knowledge}(D, R) \tag{5}$$

3.3.2 Developer Retention

Developers who have made substantial recent contributions to a project have demonstrated a high degree of commitment to the project [25, 83]. In contrast, assigning a review to a developer who is transient and will likely leave the project is antithetical to the goal of retaining project knowledge. We define commitment and contribution consistency measures to recommend reviewers with a high potential of remaining on the project, *i.e.* high retention potential. In contrast to the previous measures which are at the pull request or review level, the retention is done at a project-wide level.

ContributionRatio. We measure the contribution of potential of a developer, D, by the number of reviews and commits he or she has made in the last year divided by all the commits and reviews on the project.

$$\text{ContributionRatio}_{365}(D) = \frac{\text{TotalCommitReview}_{365}(D)}{\sum_{d}^{\text{Devs}} \text{TotalCommitReview}_{365}(d)} \tag{6}$$

ConsistencyRatio. It is common for developers to make substantial contributions to a feature and leave the project after the feature is complete. To avoid assigning reviews to transient developers, we define the $\text{ConsistencyRatio}_{364}(D)$ as the proportion of months a developer has been active in the last year.

$$\text{ConsistencyRatio}_{365}(D) = \frac{\text{ActiveMonths}_{365}(D)}{12} \tag{7}$$

***RetentionRec*.** We develop *RetentionRec* that suggests reviewers who who are unlikely to leave the project. The scoring function for a candidate review, D is

$$\textit{RetentionRec}(D) = \text{ConsistencyRatio}_{365}(D) * \text{ContributionRatio}_{365}(D) \tag{8}$$

3.3.3 Distribution and Retention Combined

***TurnoverRec*.** To ensure that knowledge is distributed among developers who are likely to remain on the project, we define the *TurnoverRec* recommender scoring function for a developer and review as

$$\textit{TurnoverRec}(D, R) = \textit{LearnRec}(D, R) * \textit{RetentionRec}(D) \tag{9}$$

`Sofia`: *TurnoverRec* and *cHRev* Combined When the files under review have many developers who know about them, it is best to suggest an expert. In contrast, when the number of knowledgeable developers is low, knowledge should be distributed among the development team. Our final recommender, `Sofia`, combines the *cHRev*, which is designed to find recent experts and *TurnoverRec*, which is defined to distribute knowledge among developers who have high retention potential. Given the function $Knowledgeable(f)$ that returns the set of developers who have modified or reviewed file f, `Sofia`(D, R) selects either a cHRev(D, R) score or a *TurnoverRec*(D, R) score as defined in the cases below:

$$\begin{cases} \textit{cHRev}(D, R), & \text{if } |Knowledgeable(f)| \leq d, \text{ any } f \mid f \in R \\ \textit{TurnoverRec}(D, R), & \text{otherwise} \end{cases} \tag{10}$$

We consider files that have no knowledgeable developers or that are hoarded by a single developer to be at risk. As result, we consider a review that has a file with 0, 1, or 2 knowledgeable developers to have a potential for knowledge loss from turnover and so distribute knowledge and set $d = 2$.

3.4 Simulation and Evaluation

To evaluate reviewer recommenders, prior works made recommendations for each exiting review and compared their result against the actual reviewers who performed the review [15, 37, 69, 89, 92, 94, 95]. To compare with the actual reviewers, we use the Mean Reciprocal Rank (MRR) and evaluate each recommender. MRR is the average of the inverse rank of the highest ranked correct recommendation. For example, if a correct recommendation is on average the third recommendation, the score would be $1/3$.

A criticism of prior works can be found in Kovalenco *et al.*'s [42] interviews with developers

who state that the recommenders rarely provide additional value because they suggest obvious expert candidate reviewers. This problem is inherent in the outcome measure, which assumes that the actual reviewers were the best, *i.e.* "correct" reviewers. Kovalenco *et al.* [42] suggests that we need to account for other perspectives and outcomes beyond simply attempting to predict the actual reviewers.

To evaluate the impact of reviewer recommendation on diverse outcomes, we perform simulations. Simulation requires us to replace the actual reviewer with a recommended reviewer and to evaluate the outcomes over a period of time. The simulation involves sequentially making recommendations for each review on a project. To train each recommender, we use the entire history prior to the the review. The recommenders consider the files under review and according to the formulas defined in Sections 3.1, 3.2, and 3.3, they randomly replace one of the actual reviewers with the top recommended reviewer. For example, if DevA actually reviewed the files, but is replaced with top recommended DevB, then the knowledge from the review will be attributed to DevB, not DevA, for future recommendation and for outcome measurement. We only randomly replace one developer to avoid disrupting the peer review process and because Kovalenco *et al.* [42] showed that developers usually already know at least one expert review candidate.

To evaluate how each recommender changes the project, we measure three outcomes: the degree of reviewer *Expertise*, reviewer *CoreWorkload*, and the number of files at risk to turnover, *FaR*. These measures incorporate the reasons interviewed developers conduct review [14, 35]. We measure the change in the outcomes over the standard quarterly period [61, 78]. Each measure is calculated as a percentage change relative to the actual reviewers who performed the review. For example, if a recommender replaces an expert reviewer with a non-expert "learner," we would expect the measures to report a percentage decrease in expertise and a percentage increase in the knowledge distribution of the development team. We define each outcome measure below.

Expertise. Having high expertise ensures having high quality code review [14, 23, 28]. We measure the *Expertise* for a review as the proportion of files under review that the selected reviewers have modified or reviewed in the past, *i.e.* the union of the files that the reviewers know about. We sum the expertise across the reviews per quarter, Q.

$$Expertise(Q) = \sum_{R}^{Reviews(Q)} \frac{\text{FilesReviewersKnow(R)}}{\text{FilesUnderReview(R)}} \tag{11}$$

CoreWorkload. To ensure high retention potential of reviewed files, a naive recommender could suggest only core developers who are both experts and are committed to the project. Such a recommender would lead to a drastic increase in the core developer workload. To measure the workload, we find the 10 reviewers who have performed the most reviews in a quarter, Top10Reviewers(Q), and sum the total number of reviews that this top 10 group performed:

$$\textit{CoreWorkload}(Q) = \sum_{D}^{\text{Top10Reviewers}(Q)} \text{NumReviews}(D, Q) \tag{12}$$

FaR. We need to quantify the project's exposure to turnover from knowledge loss. Building on Rigby *et al.*'s [78] definition of knowledge loss we define the quarterly Files at Risk, *FaR*, as the number of files that are known by zero or one active developers. Given the function ActiveDevs(Q, f) that returns the set of developers who have modified or reviewed the file, f, and have not left the project at the end of the quarter, Q, we define *FaR*(Q):

$$\textit{FaR}(Q) = \{\, f \mid f \in \text{Files} \,, |\text{ActiveDevs}(Q, f)| \leq 1\} \tag{13}$$

The raw outcome measures do not facilitate easy interpretation or comparison. We report the percentage change for a recommender relative to the actual reviewers.

We use the Equations 14,15, and 16 to report the percentage change of *Expertise*, *CoreWorkload*, and *FaR*, respectively.

$$\Delta\text{Expertise}(Q) = (\frac{\text{Simulated}\textit{Expertise}(Q)}{\text{Actual}\textit{Expertise}(Q)} - 1) * 100 \tag{14}$$

$$\Delta\text{CoreWorkload}(Q) = (\frac{\text{Simulated}\textit{CoreWorkload}(Q)}{\text{Actual}\textit{CoreWorkload}(Q)} - 1) * 100 \tag{15}$$

$$\Delta\text{FaR}(Q) = (\frac{\text{Simulated}\textit{FaR}(Q)}{\text{Actual}\textit{FaR}(Q)} - 1) * 100 \tag{16}$$

The simulation results for an *ideal reviewer recommender* increases *Expertise* during review with a positive percentage change in ΔExpertise, reduces *CoreWorkload* with a negative percentage change in ΔCoreWorkload, and reduces the number of files at risk, *FaR*, with a negative percentage change in ΔFaR.

Chapter 4

Project Selection and Data

We explicitly select well-established large projects with many completed code reviews. On smaller projects, reviewer recommendation is less meaningful as the potential set of reviews is small and the developers are often aware of the entire team. To select projects, we first query the GitHub torrent dataset to find projects with more than 10K pull requests [33, 52]. We then apply the following manual selection criteria:

(1) We need existing reviews, so 25% or more of the commits must be reviewed.

(2) We need to simulate across time, so the project must be 4 or more years old.

(3) We need diverse knowledge and modules, so we ensure there are at least 10K files.

Five projects met our selection criteria. Of these projects, CoreFX, CoreCLR, and Roslyn are led by industry but are available under an open source license and are developed in the open on GitHub. Rust and Kubernetes are community driven OSS projects. Table 4.1, provides the summary statistics including the number of files, pull requests, and commits. Our replication package contains a link to the project data [56].

We briefly describe each of the selected projects.

Rust is a general purpose programming language which is syntactically similar to C++. In contrast to C++, Rust provides memory management and garbage collection. Rust is developed by The Rust Project [10].

Table 4.1: Size of projects under study. We explicitly select for large, long-lived projects.

Name	Total Files	Reviewed PRs	Years	Developers
CoreFX	16,015	13,499	5	985
CoreCLR	15,199	10,250	4	698
Roslyn	12,313	8,646	5	469
Rust	12,472	17,499	9	2,720
Kubernetes	12,792	32,400	5	2,617

CoreCLR is the .NET platform execution engine and runs across hardware architectures. It provides a wide range of capabilities such as garbage collection, compilation to machine code, and low-level types. All the .NET languages are built on top of the CoreCLR engine and run inside its execution context. CoreCLR development is led by Microsoft [5].

CoreFX provides the core functionalities for .NET based languages including Collections, File System, and Networking. CoreFX development is led by Microsoft [6].

Roslyn is a compiler platform for C# and Visual Basic. In contrast to black-box traditional compilers, Roslyn exposes a rich set of APIs for doing static code analysis and building code analyzers. Roslyn development is led by Microsoft [9].

Kubernetes is an open-source container-orchestration platform. It facilitates automatic management, deployment, and scaling of microservices. Kubernetes developed is led by the Cloud Native Computing Foundation [7].

4.1 Gathering Data

We gather authorship commit data from git and review data from GitHub. We clone the repositories to extract all commits and corresponding changes. On GitHub, reviews are conducted in pull-requests that allow the authors and reviewers to discuss each change [34]. In this study, we consider an individual to be a reviewer of a pull-request if he or she writes a review comment on a file, asks for further changes from the author, or approves/rejects the pull request. To gather and clean the required data, we developed a post-processing pipeline which we make publicly available [56].

Unifying Developer Names. When a developer makes commits using his or her GitHub username we can link this with the email address they use in the git commit. In some cases, the author

commits without using a GitHub username and we use a name unifying approach that employs edit distances to match the git email names with GitHub usernames. This approach is similar to Bird's [18] and Canfora's [24].

Leavers. Robillard *et al.* [79] shows that using the last commit as an indicator for departure of developers draws some risks. Based on this finding, at the end of each quarter, we consider the knowledge of a developer to be inaccessible if he or she has no contribution in the subsequent four quarters. We exclude the last quarter of projects from analysis to ensure that we do not mistakenly label a developer as a leaver if they have gone on vacation for a month more.

Excluding mega commits. Rigby *et al.* [78] argue that commits with hundreds of file changes are too large to be fully comprehended by the author. In manual analysis of mega commits and review requests, we find that they tend to be superficial changes including renaming a folder, renaming a function throughout the source code, changing commented trademarks of files, or importing a large chunk of code from a different source control system to git. We do not associate any knowledge to the author or reviewer of changes with 100 or more files.

In this work, we limit our study of knowledge to code files, including *.cs*, *.java*, and *.scala*. Our replication package contains the full list of file types [56]. We also exclude changes made by bots, review comments that are made after the code has been merged, unmerged pull-requests, and files that were committed without review.

Chapter 5

Results

In this Chapter, we discuss the results for our research questions relating to (1) an empirical study of knowledge distribution during review, (2) recommendations based on ownership, (3) recommendations based on the state-of-the-art, *cHRev*, (4) learning and retention aware recommenders, and (5) `Sofia` which combines the best recommenders. We make three notes: First, we note that RQ1 does not involve simulation and is an empirical result based on the actual reviews and commits. Second, we note that the MRR outcomes does not involve simulation and instead reports how accurately the recommender predicts the actual reviewers. Third, simulations are run for each recommender and we note the changes in ΔExpertise, ΔCoreWorkload, and ΔFaR as a percentage difference relative to the actual values for each project. Table 5.2 shows the average for each outcome across all projects.

There is a table for each outcome: Table 5.3 shows ΔExpertise, Table 5.4 shows ΔCoreWorkload, Table 5.5 shows ΔFaR, and Table 5.6 shows MRR. Table 5.2 shows the average for each outcome across all projects.

5.1 RQ1: Review and Turnover

What is the reduction in files at risk to turnover when both authors and reviewers are considered knowledgeable? Recent studies have quantified knowledge loss from turnover on the basis of the commits that each developer has made [61,78]. The assumption in these works, is that knowledge is

only attained through writing code. However, the knowledge transfer that occurs during code review is widely documented with prior work showing that review promotes team awareness, transparency, and shared code ownership [14,75,80,87]. Rigby and Bird [75] quantified the additional knowledge attained during review and reported that code review exposes developers to between 100% and 150% more files than they edit. Thongtanunam *et al.* [87] added that developers who have not made any changes to a module contributed by reviewing 21% to 39% of the code changes in the module. In this section, we consider both authors of code as well as reviewers to be knowledgeable and calculate the number of files that are at risk when turnover occurs.

To assess the extent that the project is at risk to knowledge loss from turnover, we measure *FaR*, see Equation 13, which measures the number of files that have zero or one active developers at the end of each quarter. To mirror prior works, we calculate the FaR_{author} which only considers authors to be knowledgeable [61, 78]. We then calculate *FaR*, which considers both authors and reviewers as knowledgeable.

Table 5.1 reports the proportion of files at risk relative to the total files on the project. The median raw value per quarter of FaR_{author} is 7,899, 3,749, 5,780, 3,134, and 5,598 files for CoreFX, CoreCLR, Roslyn, Rust, and Kubernetes, respectively. As a percentage of the codebase, between 24% and 49% of the files are at risk of abandonment. In contrast, when both the author and the reviewer are considered knowledgeable, the median raw value per quarter of *FaR* is 2,243, 2,117, 2,070, 2,059, 2,087, respectively. As a percentage of the codebase, between 14% and 16% of the files are at risk of abandonment. As a percentage increase in files at risk for *FaR* relative to FaR_{author} we see that 69.10%, 45.27%, 61.90%, 28.44%, and 65.27% fewer files are at risk of abandonment for CoreFX, CoreCLR, Roslyn, Rust, and Kubernetes, respectively. We conclude that considering reviewers to be knowledgeable of the files they review drastically reduces *FaR* and gives a clearer picture of the risk a project is at to turnover than prior works that only considered authors to be knowledgeable [61, 78].

> When only authors are considered knowledgeable an average of 37.74% of files are at risk to turnover. When reviewers are also considered knowledgeable the *FaR* average is 15.20%. There is substantial knowledge distribution during code review.

Table 5.1: The proportion of total files that are at risk to turnover. When only authors are considered knowledgeable the proportion of files at risk is drastically higher than when both authors and reviewers are considered knowledgeable.

FaR	CoreFX	CoreCLR	Roslyn	Rust	Kubernetes	Avg
Author	48.23%	24.66%	46.94%	25.12%	43.76%	37.74%
Auth & Rev	12.44%	13.92%	16.81%	16.50%	16.31%	15.20%

5.2 RQ2 Ownership

Does recommending reviewers based on code ownership reduce the number of files at risk to turnover?

Studies show that teams tend to assign reviews to the owners of files under review [35, 80] and experts who have modified or reviewed the files in the past [15, 42]. In this research question, we run simulations to show how recommending reviewers based on ownership affects project outcome measures.

***AuthorshipRec*.** Prior works have adapted developer task recommenders [41, 49, 60] that use historical authorship data to recommend reviewers [37,95]. We partially reproduce these authorship recommendations by using the scoring function defined in Equation 1. We use the simulation method described in Section 3.4 and evaluate the impact of *AuthorshipRec* on MRR, ΔExpertise, ΔCoreWorkload, and ΔFaR. The average values are shown in Table 5.2.

AuthorshipRec is successful in predicting the reviewers who actually performed the review with an MRR of 0.59, 0.54, 0.48, 0.44, and 0.41 for CoreFX, CoreCLR, Roslyn, Rust, and Kubernetes, respectively. The average across all projects is 0.49. This implies that on average the actual reviewer is ranked 2.04.

From the simulations, we see that assigning reviewers based on their commit ownership, *i.e.* authorship, increases the *Expertise* in reviews by 7.26%, 5.97%, 19.57%, 10.89%, and 12.77%, respectively, with an average of 11.29% across the projects. The *CoreWorkload* increases for Rust by 7.50%, while it is reduced by -11.30%, -4.74%, -6.91%, and -2.95% for the other projects, with an average of -3.68%. Although *Expertise* is high for each review, *FaR* has risen across all projects by 28.05%, 12.00%, 36.23%, 33.51%, and 14.48%, with an average of 25.25%.

Developers who have authored the files under review are clearly experts. However, suggesting

past authors as reviewers concentrates the knowledge of these files and puts the project at greater risk to turnover as non-authors are not suggested as reviewers.

RevOwnRec. The majority of review recommenders have used historical review data, *i.e.* who has reviewed which files or modules in the past, to recommend reviewers [15,40,89,92,94]. We partially reproduce these review ownership results by using the scoring function defined in Equation 2. We use the simulation methodology and outcome measures as described above.

RevOwnRec is slightly less successful at predicting the reviewers who actually performed the review with an MRR of 0.53, 0.50, 0.42, 0.46, and 0.37 for CoreFX, CoreCLR, Roslyn, Rust, and Kubernetes, respectively. The average across all projects is 0.45. which means the actual reviewer rank is averaged to 2.22.

From the simulations, we see that assigning reviewers based on the files they have reviewed in the past increases review *Expertise* by 12.99%, 10.14%, 22.12%, 13.33%, and 17.31% respectively, with an average of 15.17% across projects. These individuals tend to be top reviewers and we see a corresponding increase in *CoreWorkload* of 11.81%, 21.62%, 10.97%, 16.14%, and 40.93%, with an average of 20.29%. Despite the high utilization of expert reviewers, this recommender has the largest increase in files at risk with ΔFaR values of 9.29%, 51.24%, 159.42%, 105.98%, and 0.04%, with an average of 65.19%.

> Recommending reviewers based on the files they have reviewed in the past ensures expertise during review (average increase of 15.17%), but increases the workload of the top reviewers by on average 20.29% and differ from the set of actual reviewers with an average MRR of 0.45. Concentrating expertise on the top developers substantially increases the risk of knowledge loss when turnover occurs on average by 65.19%.

5.3 RQ3 *cHRev* Recommender

Does a state-of-the-art recommender reduce the number of files at risk to turnover?

cHRev builds upon prior work that leverages information in past reviews [41], but also accounts for the number of days a candidate reviewer has worked on a file, and the recency of this work

(See Section 3.2 for further details). *cHRev* has been show to outperform the other review-based recommenders, including RevFinder [87]. In this research question, we re-implement this state-of-the-art recommender and re-evaluate it. We use the simulation method described in Section 3.4 and evaluate the impact of *cHRev* on MRR, ΔExpertise, ΔCoreWorkload, and ΔFaR.

In the original cHRev paper, the authors report an average MRR of .67 across four projects [95]. On our projects, cHRev has an MRR of 0.64, 0.59, 0.49, 0.50, and 0.42, for CoreFX, CoreCLR, Roslyn, Rust, and Kubernetes, respectively. The average is 0.52. This implies that on average the actual reviewer is ranked 1.92. Although the MRR is lower in our reproduction than in the original study, we note that for MRR cHRev outperforms all of the other recommenders we consider.

From the simulations, we see that like the ownership recommenders, *cHRev* increases the *Expertise* in reviews by 9.84%, 7.27%, 16.45%, 8.22%, and 13.81%, respectively, with an average of 11.11% across projects. However, unlike *RevOwnRec*, it reduces the load on top reviewers. The corresponding values for ΔCoreWorkload are -5.93%, -2.35%, -0.51%, -2.19%, and -6.47% with an average of -3.49%. *cHRev* concentrates knowledge and increases the project's risk to turnover with a *FaR* increase of 6.46%, 13.85%, 4.43%, 10.28% in CoreFX, CoreCLR, Roslyn, and Rust, respectively and for Kubernetes the ΔFaR is reduced at -14.24%. The average of ΔFaR across all projects is 4.15%.

> *cHRev* remains accurate in suggesting actual reviewers with an MRR of 0.52. It increases the degree of *Expertise* during review by 11.11%, while reducing the *CoreWorkload* on the top reviewers by -3.49%. However, the risk of turnover increases with an average ΔFaR of 4.15%.

5.4 RQ4: Learning and Retention

Can we reduce the number of files at risk to turnover by developing learning and retention aware review recommenders?

The previous research questions have demonstrated that existing review recommenders concentrate knowledge on experts increasing the risk of knowledge loss from turnover. Furthermore, in

Table 5.2: The average of outcome measures across the projects. MRR is shown for replication purposes. Individual project outcomes are discussed in the paper text. The ideal recommender increases expertise (positive ΔExpertise), reduces workload (negative ΔCoreWorkload), and reduces files at risk to turnover (negative ΔFaR).

Recommender	Average Across Projects			
	MRR	ΔExpertise	ΔCoreWorkload	ΔFaR
AuthorshipRec	0.49	11.29%	-3.68%	25.25%
RevOwnRec	0.45	15.17%	20.29%	65.19%
cHRev	0.52	11.11%	-3.49%	4.15%
LearnRec	0.12	-35.13	-39.51%	63.04%
RetentionRec	0.39	16.59%	29.42%	-15.91%
TurnoverRec	0.19	-26.55%	1.07%	-29.54%
`Sofia`	0.43	6.27%	0.09%	-28.27%

two large industrial settings, Kovalenco *et al.* [42] interviewed developers and found that suggesting prior review experts tends to recommend reviewers that are obvious to the author of the change. They state that making obvious recommendations leads to a lack of use of recommenders. They envision a new research path for next generation of recommenders that go beyond suggesting experts. In this research question, we investigate how we can mitigate turnover-induced loss and disseminate knowledge using learning and retention measures.

***LearnRec*.** Without review recommenders, development teams naturally distribute knowledge during review by assigning reviewers who would benefit by learning about the files under review [14, 22, 80]. Building on this idea, in Section 3.3.1, we defined a scoring function that determines how many files a candidate reviewer will learn about. We ensure that the candidate knows at least one of the files that is under review. In this way, we spread knowledge, but ensure that the reviewer has some relevant knowledge. We use the simulation method described in Section 3.4 and evaluate the impact of *LearnRec* on MRR, ΔExpertise, ΔCoreWorkload, and ΔFaR with the average outcomes shown in Table 5.2.

LearnRec does a poor job of predicting the reviewers who actually performed the review with an MRR of 0.18, 0.14, 0.12, 0.11, and 0.09 for CoreFX, CoreCLR, Roslyn, Rust, and Kubernetes, respectively. The average across all projects is 0.12. This implies that on average the actual reviewer is ranked 8.33. However, the goal of this recommender was to ensure that developers learn and this shows that it suggests unexpected reviewers.

From the simulations, we see a substantial decrease in *Expertise*: -34.91%, -32.76%, -24.35%,

-50.34%, and -33.33%, respectively, with an average of -35.13% across all projects. The *CoreWorkload* is drastically reduced as fewer expert reviewers are assigned reviews: -38.07%, -38.53%, -35.68%, -49.86%, and -35.45%, with an average of -39.51%. The goal of this measure is to distribute knowledge and reduce turnover. Counter-intuitively we see an increase in the files at risk with ΔFaR values of 16.26%, 22.31%, 119.32%, 108.72%, 48.61% with an average of 63.04%. By selecting non-experts, *LearnRec* recommends transient developers who have less commitment to the project.

> The recommendations substantially differ from actual reviewers, MRR 0.12. *LearnRec* substantially reduces *Expertise*, -35.13%, but suggests learners reducing the *CoreWorkload* by -39.51%. Counter-intuitively it makes the project drastically more susceptible to knowledge loss from turnover because it assigns reviews to learners who are less committed to the project, ΔFaR of 63.04%.

RetentionRec. Assigning reviews to transient developers may distribute knowledge, but does not reduce turnover. In Section 3.3.2, we define a measure that captures how frequently developers contribute to the project and the number of months in the last year that they are active. We ensure that the candidate knows at least one of the files that is under review. Our goal is to assign reviews to committed developers. We use the same simulation methodology and outcome measures.

RetentionRec does similarly to *RevOwnRec* at predicting the reviewers who actually performed the review with an MRR of 0.57, 0.44, 0.31, 0.42, and 0.25 for CoreFX, CoreCLR, Roslyn, Rust, and Kubernetes, respectively. The average across all projects is 0.39. This implies that on average the actual reviewer is ranked 2.56.

From the simulations, we see an increase in *Expertise* of 13.84%, 10.94%, 24.80%, 24.13%, and 19.24%, respectively, with an average of 16.59%. These percentages are highest for any recommender outperforming ownership recommenders at ensuring expertise during review. We see a corresponding increase in *CoreWorkload* of 23.03%, 35.34%, 20.73%, 20.18%, and 47.82% with an average of 29.42%. However, unlike the ownership and cHRev recommenders, we see a reduction in the files at risk with ΔFaR values of -28.45%, -4.60%, -22.73%, -7.33%, and -16.47% with an average of -15.91%. Clearly *RetentionRec* selects committed developers who are unlikely to leave

the project.

> *RetentionRec* is the most successful in ensuring experts, 16.59%, during review, while reducing the risk of knowledge loss from turnover, -15.91%. However, by focusing on the most committed developers it also has the greatest increase in *CoreWorkload*, 29.42%. The MRR of 0.39 indicates that the actual reviewers are more diverse than the recommendations.

TurnoverRec. We showed that distributing knowledge through *LearnRec* does not alleviate knowledge loss and *RetentionRec* increases the *CoreWorkload*. We combine these approaches to distribute knowledge but to distribute it among individuals who have a higher retention potential. Through Equation 9, we defined *TurnoverRec* that multiplies the learning measure by the retention measure. Again we ensure that each candidate knows about at least one file. We use the same simulation methodology and outcomes.

TurnoverRec does a poor job of predicting the reviewers who actually performed the review with an MRR of 0.29, 0.20, 0.18, 0.19, and 0.12 for CoreFX, CoreCLR, Roslyn, Rust, and Kubernetes, respectively. The average across all projects is 0.19. This implies that on average the actual reviewer is ranked 5.26.

From the simulations, we see that similar to *LearnRec* recommender, the *Expertise* has decreased by -27.41%, -24.91%, -14.05%, -34.22%, and -25.93%, respectively, with an average of -26.55%. However, in terms of *CoreWorkload* there is only a slight increase of 5.98%, 5.52%, and 0.50% in CoreFX, CoreCLR, and Kubernetes and a reduction in Roslyn and Rust by -0.12% and -6.52% with an average of 1.07%. The files at risk are reduced with a ΔFaR of -34.95%, -14.20%, -41.70%, -24.32%, and -32.53% with an average of -29.54%.

> *TurnoverRec* combines learning and retention recommenders and has the greatest reduction in turnover risk, ΔFaR-29.54. However, there is a substantial cost in the reduction of *Expertise*, -26.55%, and a minor increase in *CoreWorkload*, 1.07. The low MRR value of 0.19 indicates that developers naturally focus on reviewers with greater expertise than *TurnoverRec*.

Table 5.3: Change in Expertise. Compared to the reviewers who actually performed the review, a *positive* values indicate an increase in expertise with the recommended reviewers.

Recommender	Δ*Expertise*				
	CoreFX	CoreCLR	Roslyn	Rust	Kubernetes
AuthorshipRec	7.26%	5.97%	19.57%	10.89%	12.77%
RevOwnRec	12.99%	10.14%	22.12%	13.33%	17.31%
cHRev	9.84%	7.27%	16.45%	8.22%	13.81%
LearnRec	-34.91%	-32.76%	-24.35%	-50.34%	-33.33%
RetentionRec	13.84%	10.94%	24.80%	14.13%	19.24%
TurnoverRec	-27.41%	-24.91%	-14.05%	-34.22%	-25.93%
`Sofia`	4.69%	3.23%	8.04%	5.82%	9.58%

Table 5.4: Change in *CoreWorkload*. Compared to the reviewers who actually performed the review, a *negative* value is an improvement and indicates that the recommended reviewers reduce the workload on the core team.

Recommender	Δ*CoreWorkload*				
	CoreFX	CoreCLR	Roslyn	Rust	Kubernetes
AuthorshipRec	-11.30%	-4.74%	-6.91%	7.50%	-2.95%
RevOwnRec	11.81%	21.62%	10.97%	16.14%	40.93%
cHRev	-5.93%	-2.35%	-0.51%	-2.19%	-6.47%
LearnRec	-38.07%	-38.53%	-35.68%	-49.86%	-35.45%
RetentionRec	23.03%	35.34%	20.73%	20.18%	47.82%
TurnoverRec	5.98%	5.52%	-0.12%	-6.52%	0.50%
`Sofia`	-0.27%	-5.89%	5.09%	0.43%	1.12%

Table 5.5: Change in *FaR*. Compared to the reviewers who actually performed the review, a *negative* value is an improvement and indicates that the recommended reviewers reduce the number of files at risk.

Recommender	Δ*FaR*				
	CoreFX	CoreCLR	Roslyn	Rust	Kubernetes
AuthorshipRec	28.05%	12.00%	36.23%	35.51%	14.48%
RevOwnRec	9.29%	51.24%	159.42%	105.98%	0.04%
cHRev	6.46%	13.85%	4.43%	10.28%	-14.24%
LearnRec	16.26%	22.31%	119.32%	108.72%	48.61%
RetentionRec	-28.45%	-4.60%	-22.73%	-7.33%	-16.47%
TurnoverRec	-34.95%	-14.20%	-41.70%	-24.32%	-32.53%
`Sofia`	-34.46%	-12.42%	-41.56%	-19.92%	-33.02%

Table 5.6: Mean Reciprocal Rank, MRR, for each recommender. An MRR of $1/3$ indicates that on average the third ranked recommended reviewer actually performed the review.

Recommender	CoreFX	CoreCLR	Roslyn	Rust	Kubernetes
AuthorshipRec	0.59	0.54	0.48	0.44	0.41
RevOwnRec	0.53	0.50	0.42	0.46	0.37
cHRev	0.64	0.59	0.49	0.50	0.42
LearnRec	0.18	0.14	0.12	0.11	0.09
RetentionRec	0.57	0.44	0.31	0.42	0.25
TurnoverRec	0.29	0.20	0.18	0.19	0.12
`Sofia`	0.54	0.48	0.39	0.39	0.36

5.5 RQ5 *Sofia*

Can we combine recommenders to balance *Expertise*, *CoreWorkload*, and *FaR*?

Not all reviews contain files that are at risk of abandonment. As a result, we do not need to distribute knowledge on these files because there is already a sufficient number of developers to mitigate knowledge loss from developer turnover. In Equation 10, we define `Sofia` that distributes knowledge during review using *TurnoverRec* when there are files at risk of abandonment. In contrast, when all the files have active developers, `Sofia` uses the *cHRev* scoring function to suggest recent experts. Of the 13,690, 10,256, 10,388, 17,810, and 32,260 reviewed pull request on CoreFX, CoreCLR, Roslyn, Rust, and Kubernetes around $1/4$, 25.18%, 26.13%, 29.82%, 29.41%, and 17.17%, contain files at risk. The remaining pull requests use *cHRev* recommendations to ensure concentrated expertise. We use the simulation method described in Section 3.4 and evaluate the impact of `Sofia` on MRR, ΔExpertise, ΔCoreWorkload, and ΔFaR with average outcomes shown in Table 5.2.

`Sofia` does a good job of predicting the reviewers who actually performed the review with an MRR of 0.54, 0.48, 0.39, 0.39, and 0.36 for CoreFX, CoreCLR, Roslyn, Rust, and Kubernetes, respectively. The average across all projects is 0.43. This implies that on average the actual reviewer is ranked 2.32.

From the simulations, we see that by only distributing knowledge when files are at risk and otherwise suggesting experts, `Sofia` inherits the best characteristics of *TurnoverRec* and *cHRev*. The *Expertise* goes up by 4.69%, 3.32%, 8.04%, 5.82%, and 9.58%, respectively, with an average of 6.27%. In terms of *CoreWorkload*, we see a reduction of -0.27% and -5.89% in CoreFX and

CoreCLR, an increase in Roslyn of 5.09% and a slight increase of 0.43% and 1.12% for Rust and Kubernetes. The average of ΔCoreWorkload is minor at 0.09%. `Sofia` distributes knowledge to developers who have a high retention potential and reduces the risk of turnover as measured by a decrease in ΔFaR of -34.46%, 12.42%, -41.56%, -19.92%, and -33.02%, with an average of -28.27%.

> The `Sofia` recommender distributes knowledge when there are files under review that are at risk of abandonment and suggests experts when all files already have multiple knowledgeable developers. This strategy allows us to increase the level of *Expertise* during review, 6.27%, while having a minor impact on *CoreWorkload*, 0.09%, and substantially reducing the number of files at risk by -28.27%. `Sofia` also does a reasonable job of predicting the actual reviewers with an MRR of 0.43.

5.6 The `Sofia` Bot on GitHub

Code review is known to have multiple purpose and outcomes from finding defects to distributing knowledge [14, 22, 35, 75, 80]. Our tool design allows developers to make an informed selection balancing the need for experts and learners. We created a GitHub application [57] that will recommend reviewers based on the combination of *cHRev* with *TurnoverRec* as the `Sofia` bot. Feedback from developers showed that the rationale behind a review recommendation is required [42]. For the `Sofia` bot we display simplified measures to complement a developer's intuition and domain expertise on who should review the pull request.

Implementation. The `Sofia` repository with the source code and the straightforward installation instructions are publicly available [4]. Once installed `Sofia` processes the entire history of the project to be able to recommend reviewers. `Sofia` uses GitHub webhooks to scan submitted commits and reviewed pull requests to keep recommendations up-to-date. `Sofia` can operate in two modes: fully automated or list candidates. In the fully automated mode, for each pull request, `Sofia` assigns the top scoring candidate to perform the review.

In Figure 5.1 `Sofia` displays a list of candidates when the pull request is created or when the `Sofia suggest` command is issued (Box A in figure). The `Sofia` bot displays the ranked list of

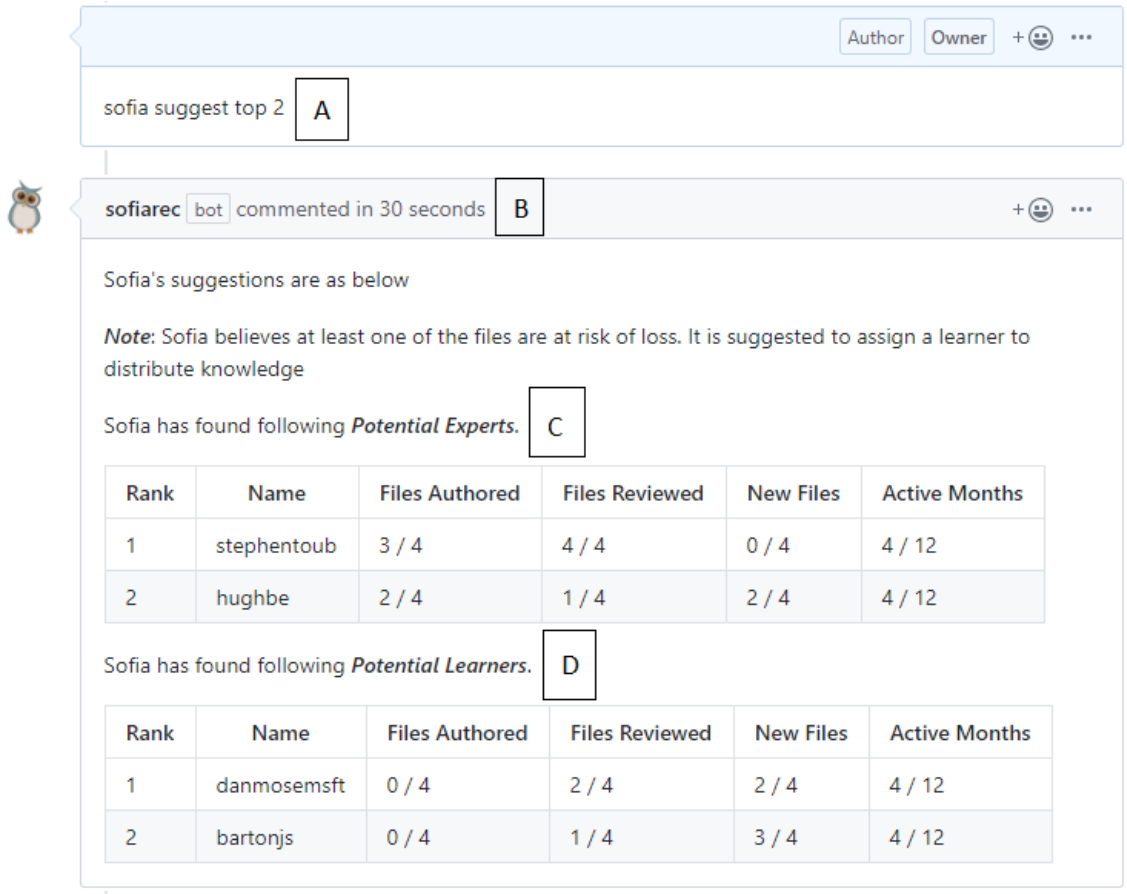

Figure 5.1: An example of `Sofia` recommending both learners and experts for the CoreFX project.

potential reviewers (Box B). In Box C in the figure, the author can select the person with the highest expertise. Or if learning is more important they can select the developer who would learn about the most files (Box D). The author can also issue the `Sofia suggest learners` or `Sofia suggest experts` if he or she is only interested in a particular type of candidate.

To help with tool adoption, the displayed measures are designed to be quick and easy to interpret by pull request authors and are major simplifications of the scoring functions defined in Section 3.3. The ownership dimension maps to the "Files Authored" and the "Files Reviewed" fields which are simplified to show the proportion of files under review that the candidate has authored or reviewed in the past, respectively. Learning maps to the "New Files" field which is simplified to the number of files that the candidate would learn about, *i.e.* they have not modified or reviewed. Retention potential maps to the "Active Months" field which is simplified to the proportion of months that the developer has been active in the previous year.

The goal of our tool is to compliment a developer's intuition. For example, if a developer feels that high expertise is required, he or she might choose the top candidate in Figure 5.1 Box C, "stephentoub," who has in the past modified $3/4$ of the files under review, has reviewed all of the files under review, and has been active in 5 months in the last year. `Sofia` will warn developers that in a review there is at least one file at risk (top of Box B). The developer may then select the best "learner" review candidate from Box D, "danmosemsft." Although he has never modified the files under review, he has reviewed 2 of 4 and has also been been active in 5 of the last 12 months. Finally, "hughbe" has both expertise and would also learn about new files. He has authored 2 of 4 of the files under review, reviewed 1 of the files, and would learn about 2 new files. He has also been active 5 of the last 12 months. `Sofia` makes recommendations, but provides a simple rationale for each review candidate allowing the developer to select the best reviewer given their intuition and the review context.

5.7 Threats to Validity

Generalizability. We selected large and successful open source software projects that were led by either industry or a community. On smaller projects, there is no need for reviewer recommendation because the list of candidates is small and obvious to all developers. Future work is necessary to validate our results in other development contexts.

Construct Validity. Following prior works on review recommendation [89, 95], ownership [31, 35, 75], and turnover [61, 78], we use the source code file as the unit of knowledge. Knowledge is contained in other documents and at other unit levels. We leave these investigations to future work. We have also provided formulas for each of our measures and scoring functions to facilitate replication.

The knowledge acquired by a reviewer will be different from the knowledge of the author. The author will usually know more of the details, while an expert reviewer may know more about other modules and dependencies. In this work, we consider both authors and reviewers to be knowledgeable and able to work on the files when turnover occurs. Future work is required to understand the different types of knowledge that authors and reviewers have.

Replication and Reproducability. Existing recommenders including ReviewBot [15], RevFinder [87], and cHRev [95] do not provide a replication package or source code for their recommenders. As a result, we re-implemented cHRev for comparison because it outperform other state-of-the-art recommenders. We also implemented simple authorship and ownership recommenders. Comparing each recommender with existing baseline recommenders reduces the threat of internal validity. We make all of our code, data, and GitHub `Sofia` bot available for future researchers as well as for use on software projects [56].

Chapter 6

Discussion, Literature, and Conclusion

We position our findings in the research literature. We discuss how we advance our understanding of code review practice, mitigation of turnover risk through *FaR*, and evaluate reviewer recommender systems on diverse outcome measures.

6.1 Understanding Code Review Practice

Fagan [28] introduced software inspections in 1976 with a detailed experiment that conclusively showed that inspection found defects earlier in the design process and that unreviewed design artifacts lead to defects that slipped through to latter stages increased overall effort. In the subsequent 40 years, code review has been extensively studied. Early works focused on examining the process [27, 28]. However, Porter *et al.* [67] demonstrated that process was much less of a factor than ensuring expertise during review. Current code review practice favors a lightweight process that focuses on expert discussion of changes to the system [14, 22, 23, 75, 77] that still improves software quality [50, 76]. We show that *RetentionRec* has the highest ΔExpertise among all expert recommenders with an average of 16.59%. *RevOwnRec* and *AuthorshipRec* that focus on ownership have an average of 15.17% and 11.29%, respectively. We also found that focusing on learners will reduce *Expertise* by up to -26.55%.

Recent works that interview reviewers, find that experts tend to be overloaded with their review workloads [35, 80] and that it is often difficult to find an available expert reviewer [35, 77, 89].

Moreover, it has shown that high overall workload could lead to poor review participation [88] and requesting feedback from experts can lead to delays from lack of availability and also fewer opportunities for knowledge dissemination [35]. We show that the relationship between *Expertise* and *CoreWorkload* is not straightforward. For instance, *cHRev* and *AuthorshipRec* improve the *Expertise* while at the same time reduce the *CoreWorkload* by -3.49% and -3.68% on average, respectively. On the other hand, *TurnoverRec* drastically reduces *Expertise* by -26.55% while increases the *CoreWorkload* by 1.07% and `Sofia` improves the *Expertise* with a negligible change of 0.09% in *CoreWorkload*.

6.2 Turnover-Induced Knowledge Loss and Mitigation

Turnover deprives the project of the leaver's experience and knowledge [38, 90] and has been shown to increase the number of defects [59]. Previous research has quantified the knowledge loss from turnover and shown that projects with very high turnover are susceptible to as much as five times the expected loss [61, 78]. However, these works considered authorship as the only way of gaining knowledge about files.

In contrast with prior work, we include the knowledge gained from conducting reviews into the turnover risk calculations because interviews with developers show that code review is an opportunity for learning and it plays a vital role in distributing knowledge [14, 22, 35, 75, 80]. Two separate studies quantified the knowledge gained during code review and showed that at both Google [80] and Microsoft [75] code review doubles the number of files that developers know. Furthermore, Thongtanunam *et al.* [87] showed that reviewers of modules are often not authors of the module [87]. In Section 5.1, our empirical results show that review naturally reduces turnover risk. We show that when only authors are considered knowledgeable an average of 37% of the total files are at risk. When both authors and reviewers are considered knowledgeable the average *FaR* is 15%. This reduction in far shows that substantial knowledge is attained during code review.

In this work, we design recommenders that explicitly distribute knowledge by suggesting reviewers who would learn about the files under review. We show that by distributing knowledge among developers who have a higher retention potential, there is a *FaR* reduction of -29.54% and

-28.27% for *TurnoverRec* and `Sofia`, which outperforms cHRev which increases *FaR* by 4.15%. The advantage of using code review in mitigating knowledge loss is that it adds little additional effort because code review is already a common practice on software teams. In contrast, prior works on turnover mitigation suggest increasing documentation with blogs, formalizing the process of documenting bugs in issue trackers, and participating in StackOverflow and internal QA forums [63, 70]. Each strategy requires additional developer effort especially for developers who are expert enough to answer questions and write documentation.

6.3 Recommenders

Identifying the right reviewers for a given change is a challenging and critical step in the code review process [14, 15, 28, 35, 89, 95]. Inappropriate selection of reviewers can slow down the review process [89] or lower the quality of inspection [14, 23]. The research on reviewer recommenders focus on the problem of automatically assigning review requests to the expert developers who are most likely to provide better feedback [15, 37, 40, 89, 92, 94, 95].

Advanced recommenders have been proposed which are built upon machine learning [40], text mining [92], and social relation graphs [94]. However, these papers do not provide public implementation of their recommenders. Re-implementing and testing these recommenders against our outcome measures is beyond the scope we set for this paper. We hope future work will examine these recommenders, and we release all our code and data to facilitate replication and advancement of review recommenders [55, 56].

The existing recommenders have been evaluated using accuracy metrics such as *Top-K* and *MRR* that measure how accurately the recommendations match the actual developers that were involved in a review. This evaluation is based upon the assumption that actual reviewers were among the best candidates to review a change [42]. However, it is reported that the focus on accuracy rarely provides additional value for developers because the recommendations are obvious [42]. Furthermore, in teams with strong code ownership, finding relevant experts is not problematic [80]. For replication completeness we calculated MRR. Our results confirm Kovalenco *et al.*'s findings that a broader perspective is needed when evaluating recommenders. We showed that recommenders with similar

MRR values may have entirely different impact on *Expertise*, *CoreWorkload*, and *FaR*. For instance, *RevOwnRec* and *RetentionRec* have a difference of 0.06 in MRR while the difference between their ΔFaR is 81.10%. *LearnRec* and *TurnoverRec* have a difference of 0.07 in MRR while the difference between their ΔCoreWorkload and ΔFaR is 40.58% and 92.58%.

6.4 Concluding Remarks

In this study, we provide a novel evaluation framework for reviewer recommenders based their impact on *Expertise*, *CoreWorkload*, and Files at Risk to turnover (*FaR*). We show that selecting reviewers solely based on ownership, expertise, or learning proxy measures does not balance all three outcomes and leads to a knowledge concentration, low knowledge retention, or low expertise.

The outcome of this work is `Sofia` that combines the state-of-the-art expert recommender, *cHRev*, with the learning and retention recommender, *TurnoverRec*. This bi-functional recommender adapts itself to the context of the review. It distributes knowledge when there are files under review that are at risk to turnover, but otherwise suggests experts. Through simulation we show that `Sofia` is the only recommender that balances the three outcomes simultaneously. This strategy allows us to increase the level of *Expertise* during review by 6.27%, while having a minor impact on workload, ΔCoreWorkload 0.09%, and reducing the number of files at risk with a ΔFaR of -28.27%.

We release `Sofia` bot as an open source software that fully integrates with GitHub pull requests and provides reviewer recommendations. The recommendations complement a developer's intuition and experience by providing simple rationale for each review candidate, such as showing how active a candidate has been, how many files he or she would learn about if they performed the review, and how many of the files under review they have modified or reviewed in the past. To the best of our knowledge, existing reviewer recommenders including Microsoft's CodeFlow [75] and Google's Gerrit [80] do not explicitly recommend reviewers based on distributing knowledge to reduce turnover. Future work is necessary to fully evaluate `Sofia` in practice.

In answering our research questions we make 8 contributions.

First, prior works assume that the developer who actually performed the review are the best

candidates and report Top-K and MRR style outcomes. We question the validity of this approach as prior works have shown that surveyed developers feel recommendations are obvious and not useful [42]. In contrast, we introduce the outcome measures of *Expertise*, *CoreWorkload*, and *FaR*, which allow us to determine the change in expertise, workload, and the number of files that are at risk. These outcomes account for factors beyond the normal "find the best" expert evaluation. Section 3 provides the formal definition for each measure.

Second, we remove the assumptions of turnover research that only authors of code are knowledgeable [61, 78]. In Section 5.1, we empirically showed that When only authors are considered knowledgeable an average of 37.74% of files are at risk to turnover. When reviewers are also considered knowledgeable the *FaR* average is 15.20%. There is substantial knowledge distribution during code review.

Third, we examine the impact of recommending reviewers with high authorship and review ownership of the files under review. In Section 5.2, we show that recommending reviewers based on the files they have reviewed in the past ensures expertise during review (average increase of 15.17%), but increases the workload of the top reviewers by on average 20.29% and differ from the set of actual reviewers with an average MRR of .45. Concentrating expertise on the top developers substantially increases the risk of knowledge loss when turnover occurs on average by 65.19%.

Fourth, we re-implement and re-evaluate the state-of-the-art review recommender, cHRev [95]. In Section 5.3, we find that *cHRev* remains accurate in suggesting actual reviewers with an MRR of 0.52. It increases the degree of *Expertise* during review by 11.11%, while reducing the *CoreWorkload* on the top reviewers by -3.49%. However, the risk of turnover increases with an average ΔFaR of 4.15%.

Fifth, we make recommendations that focus on spreading knowledge. In Section 5.4 we find that recommendations of *LearnRec* substantially differ from actual reviewers, MRR 0.12. *LearnRec* substantially reduces *Expertise*, -35.13%, but suggests learners reducing the *CoreWorkload* by -39.51%. Counter-intuitively it makes the project drastically more susceptible to knowledge loss from turnover because it assigns reviews to learners who are less committed to the project, ΔFaR of 63.04%.

Sixth, we make recommendations based on the retention potential of each candidate reviewer. In

Section 5.4, we show that *RetentionRec* is the most successful in ensuring experts, 16.59%, during review, while reducing the risk of knowledge loss from turnover, -15.91%. However, by focusing on the most committed developers it also has the greatest increase in *CoreWorkload*, 29.42%. The MRR of .39 indicates that the actual reviewers are more diverse than the recommendations.

Seventh, we introduce the `Sofia` review recommender in Section 5.5, The `Sofia` recommender distributes knowledge when there are files under review that are at risk of abandonment and suggests experts when all files already have multiple knowledgeable developers. This strategy allows us to increase the level of *Expertise* during review, 6.27%, while having a minor impact on *CoreWorkload*, 0.09%, and substantially reducing the number of files at risk by -28.27%. `Sofia` also does a reasonable job of predicting the actual reviewers with an MRR of .43.

Eighth, we implement `Sofia` into GitHub pull request. `Sofia` makes ranks candidates and complements developer experience and intuition by displaying simple measures, including how many files the candidate is familiar with, how many new files he or she will learn about, and how active the developer has been in the last year.

We have shown that existing reviewer recommenders tend to concentrate knowledge on experts. This concentration of knowledge exacerbates turnover risk. We evaluate review recommenders with three criteria *Expertise*, *CoreWorkload*, and *FaR*. These outcomes provide a nuanced view and the potential to expand what is considered a valuable review recommendation. We hope that future work will replicate our study and that software teams will use our code review recommendations to reduce their exposure to knowledge loss from turnover.

Appendix A

Tools

In this study, we implemented two tools named RelationalGit and *Sofia*, respectively. At the time of writing this theses, RelationalGit has been downloaded more than 7,700 times.

It is worth to note that apart from RelationalGit and *Sofia*, we have developed two other open source libraries named Octokit.Extensions [53] and Octokit.Bot [54] which have been downloaded more than 3000 times. Octokit.Extensions [51] makes gathering huge amount of information from GitHub possible by extending the popular Octokit.net library to mitigate the GitHub throttling policy and transient HTTP related errors. Octokit.Bot [8] is an application framework helps with creating GitHub Apps using C# and ASP.NET Core. This library takes care of lots of boiler plate code and lets you focus on nothing except your domain.

A.1 RelationalGit

RelationalGit [55, 56] gathers code changes and review histories of software projects. It mines git data structure to extract commits, blames, changes, and developers of a codebase. Using GitHub APIs, it collects pull requests and all related data such as files, reviewers, and comments.

Beside being a data extraction tool, RelationalGit consists of a historical simulation functionality. Through simulations, one can understand how using different reviewer recommendation strategies affects project's knowledge distribution, workload of developers, and expertise of reviews.

A.1.1 Installation

RelationalGit is a cross platform *dotnet tool* application written using C# and .NET Core. Users can install RelationalGit through command-line interface such as bash, cmd, or PowerShell.

.NET Core

.NET Core is a prerequisite for installing RelationalGit. You can install .NET Core Runtime from *https://dotnet.microsoft.com/*. After installation run the *dotnet version* through command-line to make sure the framework is installed.

PowerShell Core

PowerShell Core is a cross platform command-line interface. Behind the scenes, RelationalGit uses PowerShell Core to call git APIs for extracting blames. You can install PowerShell Core from here [11].

Database

RelationalGit stores data in a relational database to make further analysis easier. As of now, the only supported database is Microsoft SQL Server. Microsoft SQL Server is a cross platform database which is free for academic purposes and can handle large amounts of data.

The recommended way for installing Microsoft SQL Server is to install its docker image [12] on your system which makes installation as easy as executing a single line of code.

dotnet tool

To install RelationalGit, you just need to run the following command.

dotnet tool install –global RelationalGit.

Previous versions are available at *https://www.nuget.org/packages/RelationalGit*.

A.1.2 Configuration File

RelationalGit uses a JSON configuration file to read the connection string and required parameters of commands. By default, RelationalGit reads a file named *relationalgit.json* located in user's document folder.

(1) **ConnectionStrings.** Has a child element called *RelationalGit* which holds the connection string value required for connecting to the database.

(2) **Mining.** Consists of several child elements that are required for execution of commands. We explain each parameter at following sections where we explain each command individually.

If you will to use a different location for the configuration file, you have to explicitly pass its location using the *–conf-path* parameter.

dotnet-rgit –cmd NAME_OF_COMMAND –conf-path "C:/Users/Administrator/Desktop/relationalgit.json"

A.1.3 Git Commands

Once you have installed RelationalGit, there are variety of commands at your disposal to extract data from a repository you have cloned.

get-git-commits

Extracts the commits from the repository referenced by the *RepositoryPath* parameter in the configuration file or the *repo-path* argument. It populates the Commits and CommitRelationships tables.

dotnet-rgit –cmd get-git-commits –repo-path "PATH_TO_REPO"

get-git-commits-changes

Description. Extracts the commits from the repository.

Database. It populates the *CommitChanges* table of the database.

Arguments. The repository path and the name of the branch is required. This command gets changes of commits accessible from the branch.

Table A.1: Arguments of get-git-commits-changes

	Repository Path	Branch
Configuration File	RepositoryPath	GitBranch
Command Arg	repo-path	git-branch

Sample.

dotnet-rgit –cmd get-git-commits-changes –repo-path "PATH_TO_REPO" –git-branch master

alias-git-names

Description. Normalizes the name of authors based on their name, email address, and GitHub login.

Database. It populates the *AliasedDeveloperNames* table of the database.

Sample.

dotnet-rgit –cmd alias-git-names

periodize-git-commit

Description. Breaks the project's history into periods.

Database. It populates the *Periods* table of the database.

Arguments. The period type and period length are required. As of now, the only supported period type is month.

Table A.2: Arguments of periodize-git-commit

	Period Type	Period Length
Configuration File	PeriodType	PeriodLength
Command Arg	period-type	period-length

Sample. Break the project history into quarters.

dotnet-rgit –cmd periodize-git-commit –period-type month –period-length 3

get-git-commit-blames-for-periods

Description. Extracts files and blames of the last commit of each period.

Database. It populates the *CommittedBlob* and *CommitBlobBlames* tables.

Arguments. The required arguments are repository path, branch, extensions of files, period Ids, excluded paths, and whether to do blaming operation.

Table A.3: Arguments of get-git-commit-blames-for-periods

	Repository Path	Branch	Extensions	Periods	Exclusions	Blaming
Configuration File	RepositoryPath	GitBranch	Extensions	BlamePeriods	ExcludedBlamePaths	ExtractBlames
Command Arg	repo-path	git-branch	extensions	blame-periods	exclude-blame-path	extract-blames

Sample. Extract blames of .cs and .java files for all periods.

dotnet-rgit –cmd get-git-commit-blames-for-periods –repo-path PATH_To_REPO –git-branch master –extensions .cs .java –extract-blames true

Sample. Extract blames of .cs and .java files for period 1 and 2.

dotnet-rgit –cmd get-git-commit-blames-for-periods –repo-path PATH_To_REPO –git-branch master –extensions .cs .java –extract-blames true –blame-periods 1 2

Sample. Extract blames of .cs and .java files all periods. Exclude files located at lib and third-party folders.

dotnet-rgit –cmd get-git-commit-blames-for-periods –repo-path PATH_To_REPO –git-branch master –extensions .cs .java –extract-blames true –exclude-blame-path "/lib/*" "*/thirdparty/*"*

apply-git-aliased

Description. Assigns authors' unique name to commits, blobs and blames. You need to execute alias-git-names before running this command.

Database. It updates *Commits*, *CommittedBlob*, and *CommitBlobBlames* tables.

Sample.

dotnet-rgit –cmd apply-git-aliased

ignore-mega-commits

Description. turn on the ignore flag of mega commits and their associated blames. Also, commits and blames authored by mega developers are marked as ignored.

Database. It updates *Commits*, *CommittedBlob*, and *CommitBlobBlames* tables.

Arguments. The size of mega commits and the list of mega developers are required.

Table A.4: Arguments of ignore-mega-commits

	Mega Commit Size	Mega Developer
Configuration File	MegaCommitSize	MegaDevelopers
Command Arg	mega-commit-size	mega-devs

Sample. Ignore commits which contain more than 100 changes or are authored by dotnet-bot and botbot developers.

dotnet-rgit –cmd ignore-mega-commits –mega-devs dotnet-bot botbot –mega-commit-size 100

A.1.4 GitHub Commands

GitHub Commands gather review related data from GitHub. These commands all require the same set of arguments which are GitHub owner, Repository Name, Access Token, and Branch Name.

By default, GitHub restricts API calls to 60 requests per hour. This limit will be 5000 requests per hour if you obtain a personal access token [1].

Table A.5: Arguments of GitHub Commands

	Owner	Repository	Token	Branch
Configuration File	GitHubOwner	GitHubRepo	GitHubToken	GitBranch
Command Arg	github-owner	github-repo	github-token	git-branch

get-github-pullrequests

Description. retrieves the list of all pull requests.

Database. It populates the *PullRequests* table.

Sample.

dotnet-rgit –cmd get-github-pullrequests

get-github-pullrequest-reviewers

Description. Gets the list of reviewers assigned to pull requests.

Database. It populates the *PullRequestReviewers* table.

Sample.

dotnet-rgit –cmd get-github-pullrequest-reviewers

get-github-pullrequest-reviewer-comments

Description. retrieves the list of inline comments made on pull requests.

Database. It populates the *PullRequestReviewerComments* table.

Sample.

dotnet-rgit –cmd get-github-pullrequest-reviewer-comments

get-pullrequest-issue-comments

Description. retrieve the list of comments made on the discussion thread of pull requests.

Database. It populates the *IssueComments* table.

Sample.

dotnet-rgit –cmd get-pullrequest-issue-comments

get-github-pullrequests-files

Description. retrieves the files of pull requests.

Database. It populates the *PullRequestFiles* table.

Sample.

dotnet-rgit –cmd get-github-pullrequests-files

map-git-github-names

Description. links GitHub logins to the corresponding normalized unique author names.

Database. It populates the *GitHubGitUsers* table.

Sample.

dotnet-rgit –cmd map-git-github-names

A.1.5 Historical Simulations Command

Through historical simulations, we can evaluate the effectiveness of different approaches to reviewer recommendation. In these simulation, we change the actual reviewers of pull requests

with recommendations generated by a given recommender. After simulation, we can query the database to see how expertise, workload, and knowledge distribution change.

We have implemented 9 different recommenders in RelationalGit which are:

(1) **nothing.** Returns no recommendations. Using this recommender we can understand how having no reviewers affect projects.

(2) **reviewers-actual.** Does not change the actual reviewers. Using this recommender we can understand the actual expertise, workload, and knowledge distribution of projects.

(3) **bird.** Is an implementation of the state-of-the-art reviewer recommender called cHRev [95]

(4) **commit.** Recommends reviewers based on their commit ownership.

(5) **review.** Recommends reviewers based on their review ownership.

(6) **persist.** Recommends reviewers based on the likelihood of their retention.

(7) **spreading.** Recommends reviewers who no less that others about files under change.

(8) **persist-spreading.** Recommends reviewers based on the combination of spreading and retention factors.

(9) **sofia.** Recommends reviewers based on the combination of cHRev and persist-spreading algorithms.

The arguments required for the simulation commands are:

(1) **First Period.** Indicates the first period that the simulation change historical reviews. For example, if first periods is 5, then reviews conducted before period 5 will not be changed.

(2) **Size of Mega Pull Requests.** Is a threshold value which tell the simulator to ignore mega size pull requests.

(3) **Reviewer Replacement Strategy.** Specifies how to replace actual reviewers with recommended ones.

(4) **Recommender Algorithm.** Specifies the recommender algorithm for generating reviewer recommendations.

The following sample command does the historical simulation started from the first periods, uses the `Sofia` recommender, ignores pull requests with more than 100 files. As for the replacement strategy 1) it does not add a reviewer to pull requests with no reviewers 2) for other pull requests randomly replaces one of the actual reviewers with the top candidate.

dotnet-rgit –cmd compute-loss –simulation-first-period 1 –mega-pr-size 100 –save-strategy sofia –pullRequests-reviewer-selection "0:nothing-nothing,-:replacerandom-1"

A.2 `Sofia`: GitHub Application

We implemented a `Sofia` reviewer recommender as a free GitHub Application [57] that can be installed on GitHub repositories [4]. Users can ask `Sofia` to suggest experts, learners, or both.

A.2.1 Gathering Historical Data

Once the application is installed on the repository, you need to ask `Sofia` to gather all historical commits and reviews. This information is required for building a knowledge model which represent who knows about which file. To initiate a scanning process, user has to open an issue with its body filled with `Sofia` *scan branch master* command.

This operation takes around 3 hours for large repositories. After gathering historical information, `Sofia` keeps its knowledge model up to date by listening to commit and reviews event sent through GitHub webhooks.

A.2.2 Ask for Recommendations

Users can ask `Sofia` to return a list of reviewer candidates according to expertise or learning opportunities.

To ask for expert reviewers, you need to issue *"*`Sofia` *suggest experts"* command by making a new comment on the pull request discussion thread.

To spread knowledge and promote learning opportunities, you need to issue *"*`Sofia` ***suggest learners"*** command by making a new comment on the pull request discussion thread.

By issuing *"*`Sofia` ***suggest"*** command, *`Sofia`* returns both of experts and learners. It also inform user if any of the files have less than 3 knowledgeable developers.

Bibliography

[1] Creating a personal access token for the command line. https://help.github.com/en/enterprise/2.17/user/articles/creating-a-personal-access-token-for-the-command-line. [Online; accessed 11-August-2019].

[2] Fowler: Code ownership. https://martinfowler.com/bliki/CodeOwnership.html. [Online; accessed 11-August-2019].

[3] Git blame. https://git-scm.com/docs/git-blame. [Online; accessed 11-August-2019].

[4] Github app: Sofiarec. https://github.com/apps/sofiarec. [Online; accessed 11-August-2019].

[5] Github coreclr. https://github.com/dotnet/coreclr. [Online; accessed 11-August-2019].

[6] Github corefx. https://github.com/dotnet/corefx. [Online; accessed 11-August-2019].

[7] Github kuberenetes. https://github.com/kubernetes/kubernetes. [Online; accessed 11-August-2019].

[8] Github: Octokit.bot. https://github.com/mirsaeedi/octokit.net.bot. [Online; accessed 11-August-2019].

[9] Github roslyn. https://github.com/dotnet/roslyn. [Online; accessed 11-August-2019].

[10] Github rust. https://github.com/rust-lang/rust. [Online; accessed 11-August-2019].

[11] Installing various versions of powershell. https://docs.microsoft.com/en-us/powershell/scripting/install/installing-powershell?view=powershell-6. [Online; accessed 11-August-2019].

[12] Quickstart: Run sql server container images with docker. https://docs.microsoft.com/en-us/sql/linux/quickstart-install-connect-docker?view=sql-server-2017&pivots=cs1-bash. [Online; accessed 11-August-2019].

[13] J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In *Proceedings of the 28th international conference on Software engineering*, pages 361–370. ACM, 2006.

[14] A. Bacchelli and C. Bird. Expectations, outcomes, and challenges of modern code review. In *Proceedings of the 2013 international conference on software engineering*, pages 712–721. IEEE Press, 2013.

[15] V. Balachandran. Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 931–940. IEEE Press, 2013.

[16] L. Bao, Z. Xing, X. Xia, D. Lo, and S. Li. Who will leave the company?: a large-scale industry study of developer turnover by mining monthly work report. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 170–181. IEEE, 2017.

[17] K. Beck and C. Andres. Extreme programming explained: Embrace change. 2-nd edition. *Addison-Illl-iesley Professional, Zttlld*, 2005.

[18] C. Bird, A. Gourley, P. Devanbu, M. Gertz, and A. Swaminathan. Mining email social networks. In *Proceedings of the 2006 international workshop on Mining software repositories*, pages 137–143. ACM, 2006.

[19] C. Bird, N. Nagappan, B. Murphy, H. Gall, and P. Devanbu. An analysis of the effect of code ownership on software quality across windows, eclipse, and firefox. *Technical report, University of California*, 2010.

[20] C. Bird, N. Nagappan, B. Murphy, H. Gall, and P. Devanbu. Don't touch my code!: examining the effects of ownership on software quality. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 4–14. ACM, 2011.

[21] B. W. Boehm and T. DeMarco. Software risk management. *IEEE software*, (3):17–19, 1997.

[22] A. Bosu, J. C. Carver, C. Bird, J. Orbeck, and C. Chockley. Process aspects and social dynamics of contemporary code review: Insights from open source development and industrial practice at microsoft. *IEEE Transactions on Software Engineering*, 43(1):56–75, 2016.

[23] 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.

[24] G. Canfora, M. Di Penta, R. Oliveto, and S. Panichella. Who is going to mentor newcomers in open source projects? In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 44. ACM, 2012.

[25] E. Constantinou and T. Mens. An empirical comparison of developer retention in the rubygems and npm software ecosystems. *Innovations in Systems and Software Engineering*, 13(2-3):101–115, 2017.

[26] G. G. Dess and J. D. Shaw. Voluntary turnover, social capital, and organizational performance. *Academy of management review*, 26(3):446–456, 2001.

[27] M. Fagan. Design and code inspections to reduce errors in program development. In *Software pioneers*, pages 575–607. Springer, 2002.

[28] M. E. Fagan. Design and Code Inspections to Reduce Errors in Program Development. *IBM Systems Journal*, 15(3):182–211, 1976.

[29] M. E. Fagan. Advances in software inspections. In *Pioneers and Their Contributions to Software Engineering*, pages 335–360. Springer, 2001.

[30] M. Foucault, M. Palyart, X. Blanc, G. C. Murphy, and J.-R. Falleri. Impact of developer turnover on quality in open-source software. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 829–841. ACM, 2015.

[31] T. Fritz, G. C. Murphy, and E. Hill. Does a programmer's activity indicate knowledge of code? In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 341–350. ACM, 2007.

[32] T. Girba, A. Kuhn, M. Seeberger, and S. Ducasse. How developers drive software evolution. In *Eighth International Workshop on Principles of Software Evolution (IWPSE'05)*, pages 113–122. IEEE, 2005.

[33] G. Gousios. The ghtorent dataset and tool suite. In *Proceedings of the 10th working conference on mining software repositories*, pages 233–236. IEEE Press, 2013.

[34] G. Gousios, M. Pinzger, and A. v. Deursen. An exploratory study of the pull-based software development model. In *Proceedings of the 36th International Conference on Software Engineering*, pages 345–355. ACM, 2014.

[35] M. Greiler, C. Bird, M.-A. Storey, L. MacLeod, and J. Czerwonka. Code reviewing in the trenches: Understanding challenges, best practices and tool needs. 2016.

[36] T. Hall, S. Beecham, J. Verner, and D. Wilson. The impact of staff turnover on software projects: the importance of understanding what makes software practitioners tick. In *Proceedings of the 2008 ACM SIGMIS CPR conference on Computer personnel doctoral consortium and research*, pages 30–39. ACM, 2008.

[37] C. Hannebauer, M. Patalas, S. Stünkel, and V. Gruhn. Automatically recommending code reviewers based on their expertise: An empirical comparison. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 99–110. ACM, 2016.

[38] M. A. Huselid. The impact of human resource management practices on turnover, productivity, and corporate financial performance. *Academy of management journal*, 38(3):635–672, 1995.

[39] D. Izquierdo-Cortazar, G. Robles, F. Ortega, and J. M. Gonzalez-Barahona. Using software archaeology to measure knowledge loss in software projects due to developer turnover. In *2009 42nd Hawaii International Conference on System Sciences*, pages 1–10. IEEE, 2009.

[40] G. Jeong, S. Kim, T. Zimmermann, and K. Yi. Improving code review by predicting reviewers and acceptance of patches. *Research on software analysis for error-free computing center Tech-Memo (ROSAEC MEMO 2009-006)*, pages 1–18, 2009.

[41] H. Kagdi, M. Hammad, and J. I. Maletic. Who can help me with this source code change? In *2008 IEEE International Conference on Software Maintenance*, pages 157–166. IEEE, 2008.

[42] V. Kovalenko, N. Tintarev, E. Pasynkov, C. Bird, and A. Bacchelli. Does reviewer recommendation help developers? *IEEE Transactions on Software Engineering*, 2018.

[43] D. Krackhardt and L. W. Porter. When friends leave: A structural analysis of the relationship between turnover and stayers' attitudes. *Administrative science quarterly*, pages 242–261, 1985.

[44] E. G. Lambert, N. L. Hogan, and S. M. Barton. The impact of job satisfaction on turnover intent: a test of a structural measurement model using a national sample of workers. *The Social Science Journal*, 38(2):233–250, 2001.

[45] B. Lin, G. Robles, and A. Serebrenik. Developer turnover in global, industrial open source projects: Insights from applying survival analysis. In *2017 IEEE 12th International Conference on Global Software Engineering (ICGSE)*, pages 66–75. IEEE, 2017.

[46] M. Linares-Vásquez, K. Hossen, H. Dang, H. Kagdi, M. Gethers, and D. Poshyvanyk. Triaging incoming change requests: Bug or commit history, or code authorship? In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 451–460. IEEE, 2012.

[47] J. Lipcak and B. Rossi. A large-scale study on source code reviewer recommendation. In *2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 378–387. IEEE, 2018.

[48] A. Martensen, L. Grønholdt, et al. Internal marketing: a study of employee loyalty, its determinants and consequences. *Innovative Marketing*, 2(4):92–116, 2006.

[49] D. W. McDonald and M. S. Ackerman. Expertise recommender: a flexible recommendation system and architecture. In *Proceedings of the 2000 ACM conference on Computer supported cooperative work*, pages 231–240. ACM, 2000.

[50] S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan. An Empirical Study of the Impact of Modern Code Review Practices on Software Quality. *Empirical Software Engineering*, 21(5):2146–2189, 2016.

[51] E. Mirsaeedi. Github: Octokit.extionsions. https://github.com/mirsaeedi/octokit.net.extensions. [Online; accessed 11-August-2019].

[52] E. Mirsaeedi. Google bigquery for selecting projects. https://bigquery.cloud.google.com/savedquery/283702471694:7738d44c7a02401e8137708e304c7da2. [Online; accessed 11-August-2019].

[53] E. Mirsaeedi. Nuget: Octokit.bot. https://www.nuget.org/packages/Octokit.Extensions/. [Online; accessed 11-August-2019].

[54] E. Mirsaeedi. nuget: Octokit.extensions. https://www.nuget.org/packages/Octokit.Bot. [Online; accessed 11-August-2019].

[55] E. Mirsaeedi. Relationalgit - download. https://www.nuget.org/packages/RelationalGit. [Online; accessed 11-August-2019].

[56] E. Mirsaeedi. Relationalgit - replication package. https://github.com/cesel/relationalgit. [Online; accessed 11-August-2019].

[57] E. Mirsaeedi. Sofia. https://github.com/cesel/Sofia. [Online; accessed 11-August-2019].

[58] A. Mockus. Succession: Measuring transfer of code and developer productivity. In *Proceedings of the 31st International Conference on Software Engineering*, pages 67–77. IEEE Computer Society, 2009.

[59] A. Mockus. Organizational volatility and its effects on software defects. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 117–126. ACM, 2010.

[60] A. Mockus and J. D. Herbsleb. Expertise browser: a quantitative approach to identifying expertise. In *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*, pages 503–512. IEEE, 2002.

[61] M. Nassif and M. P. Robillard. Revisiting turnover-induced knowledge loss in software projects. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 261–272. IEEE, 2017.

[62] M. E. Nordberg. Managing code ownership. *IEEE software*, 20(2):26–33, 2003.

[63] L. G. Pee, A. Kankanhalli, G. W. Tan, and G. Tham. Mitigating the impact of member turnover in information systems development projects. *IEEE Transactions on Engineering Management*, 61(4):702–716, 2014.

[64] N. Pekala. Holding on to top talent. *Journal of Property management*, 66(5):22–22, 2001.

[65] D. E. Perry, A. Porter, M. W. Wade, L. G. Votta, and J. Perpich. Reducing inspection interval in large-scale software development. *IEEE Transactions on Software Engineering*, 28(7):695–705, 2002.

[66] M. Pinzger, N. Nagappan, and B. Murphy. Can developer-module networks predict failures? In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 2–12. ACM, 2008.

[67] A. Porter, H. Siy, A. Mockus, and L. Votta. Understanding the sources of variation in software inspections. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 7(1):41–79, 1998.

[68] F. Rahman and P. Devanbu. Ownership, experience and defects: a fine-grained study of authorship. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 491–500. ACM, 2011.

[69] 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. IEEE, 2016.

[70] M. Rashid, P. M. Clarke, and R. V. OConnor. Exploring knowledge loss in open source software (oss) projects. In *International conference on software process improvement and capability determination*, pages 481–495. Springer, 2017.

[71] E. Raymond. The cathedral and the bazaar: Musings on linuxand open source by an accidental revolutionary, 2002.

[72] F. Ricca, A. Marchetto, and M. Torchiano. On the difficulty of computing the truck factor. In *International Conference on Product Focused Software Process Improvement*, pages 337–351. Springer, 2011.

[73] P. Rigby, B. Cleary, F. Painchaud, M.-A. Storey, and D. German. Contemporary peer review in action: Lessons from open source development. *IEEE software*, 29(6):56–61, 2012.

[74] P. Rigby, D. German, and M.-A. Storey. Open source software peer review practices. In *2008 ACM/IEEE 30th International Conference on Software Engineering*.

[75] 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. ACM, 2013.

[76] P. C. Rigby, D. M. German, L. Cowen, and M.-A. Storey. Peer review on open-source software projects: Parameters, statistical models, and theory. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 23(4):35, 2014.

[77] P. C. Rigby and M.-A. Storey. Understanding broadcast based peer review on open source software projects. In *2011 33rd International Conference on Software Engineering (ICSE)*, pages 541–550. IEEE, 2011.

[78] P. C. Rigby, Y. C. Zhu, S. M. Donadelli, and A. Mockus. Quantifying and mitigating turnover-induced knowledge loss: Case studies of chrome and a project at avaya. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 1006–1016, May 2016.

[79] M. P. Robillard, M. Nassif, and S. McIntosh. Threats of aggregating software repository data. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 508–518. IEEE, 2018.

[80] 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*, pages 181–190. ACM, 2018.

[81] A. Schilling, S. Laumer, and T. Weitzel. Who will remain? an evaluation of actual person-job and person-team fit to predict developer retention in floss projects. In *2012 45th Hawaii International Conference on System Sciences*, pages 3446–3455. IEEE, 2012.

[82] D. Schuler and T. Zimmermann. Mining usage expertise from version archives. In *Proceedings of the 2008 international working conference on Mining software repositories*, pages 121–124. ACM, 2008.

[83] P. N. Sharma, J. Hulland, and S. Daniel. Examining turnover in open source software projects using logistic hierarchical linear modeling approach. In *IFIP International Conference on Open Source Systems*, pages 331–337. Springer, 2012.

[84] J. D. Shaw, N. Gupta, and J. E. Delery. Alternative conceptualizations of the relationship between voluntary turnover and organizational performance. *Academy of management journal*, 48(1):50–68, 2005.

[85] R. Shokripour, J. Anvik, Z. M. Kasirun, and S. Zamani. Why so complicated? simple term filtering and weighting for location-based bug report assignment recommendation. In *2013 10th Working Conference on Mining Software Repositories (MSR)*, pages 2–11. IEEE, 2013.

[86] M. Stovel and N. Bontis. Voluntary turnover: knowledge management–friend or foe? *Journal of intellectual Capital*, 3(3):303–322, 2002.

[87] P. Thongtanunam, S. McIntosh, A. E. Hassan, and H. Iida. Revisiting code ownership and its relationship with software quality in the scope of modern code review. In *Proceedings of the 38th international conference on software engineering*, pages 1039–1050. ACM, 2016.

[88] P. Thongtanunam, S. McIntosh, A. E. Hassan, and H. Iida. Review Participation in Modern Code Review: An Empirical Study of the Android, Qt, and OpenStack Projects. *Empirical Software Engineering*, 22(2):768–817, 2017.

[89] 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.

[90] Z. Ton and R. S. Huckman. Managing the impact of employee turnover on performance: The role of process conformance. *Organization Science*, 19(1):56–68, 2008.

[91] W. Wu, W. Zhang, Y. Yang, and Q. Wang. Drex: Developer recommendation with k-nearest-neighbor search and expertise ranking. In *2011 18th Asia-Pacific Software Engineering Conference*, pages 389–396. IEEE, 2011.

[92] 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.

[93] X. Xia, D. Lo, X. Wang, and B. Zhou. Accurate developer recommendation for bug resolution. In *2013 20th Working Conference on Reverse Engineering (WCRE)*, pages 72–81. IEEE, 2013.

[94] 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.

[95] M. B. Zanjani, H. Kagdi, and C. Bird. Automatically recommending peer reviewers in modern code review. *IEEE Trans. Softw. Eng.*, 42(6):530–543, June 2016.

[96] N. Zazworka, K. Stapel, E. Knauss, F. Shull, V. R. Basili, and K. Schneider. Are developers complying with the process: an xp study. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, page 14. ACM, 2010.