Balance Expertise, Workload and Turnover into Code Review Recommendation

Fahimeh Hajari

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

July 2022

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: **Fahimeh Hajari**

Entitled: **Balance Expertise, Workload and Turnover into Code Review Recommendation**

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
Weiyi Shang

_____ Examiner
Dr. Emad Shihab

_____ Examiner
Dr. Weiyi Shang

_____ Supervisor
Dr. Peter C. Rigby

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

September 2022

_____
Dr. Mourad Debbabi, Dean
Gina Cody School of Engineering and Computer Science

Abstract

Balance Expertise, Workload and Turnover into Code Review Recommendation

Fahimeh Hajari

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 while more evenly distributing review workload. We conduct empirical investigations to understand the natural concentration of review workload and the degree of knowledge spreading that is inherent in code review. Even though the review workload is highly concentrated, with the top 20% of reviewers doing 80.19% of reviews, code review naturally spreads knowledge and reduces the files at risk to turnover from 79.79% to 32.59%. To balance the review workload, reduce the Files at Risk to turnover, *FaR*, and maintain high levels of *Expertise* during the review, we evaluate existing code review recommenders and develop novel recommenders. We find that prior work that assigns reviewers based on file ownership concentrates knowledge on a small group of core developers, increasing the risk of knowledge loss from turnover by up to 65.19%. Recent work, *WhoDo*, that considers developer workload, assigns developers that are not sufficiently committed to the project and increases *FaR* by 40.97%. We propose learning and retention aware review recommenders that when combined are effective at reducing the risk of turnover by -29.54%, but they unacceptably reduce the overall expertise during reviews by -25.30%. Combining recommenders, we develop the *SofiaWL* recommender that suggests experts with low active review workload when none of the files under review are hoarded by developers, but distribute knowledge when files are at risk to turnover. In this way, we can simultaneously increase expertise during review with an ΔExpertise of 3.20%, reduce

workload concentration, ΔGiniWork by -12.00%, and reduce the files at risk, ΔFaR, by -23.92%.

We then focus on the *Risky File* that have zero or one knowledgeable developers. We randomly replace one of the actual reviewers with a suggested developer using *TurnoverRec* when we have a risky file in the pull request. In this approach, we can increase *Expertise* substantially in comparison to *TurnoverRec* and reduce *FaR* by -25.14%.

For the FaR^{++}, we add a learner to the actual reviewers when we have a risky file in the pull request. We reduce *FaR* by -83.88% but increase the number of review by 13.14%. To reduce the additional workload in *AwareFaR*, we only add reviewers when there are abandoned files, this decreases *FaR* by -37.51% and only increases the number of reviews by 34.24%.

Our data results and scripts are available in our replication package. [1]

[1] Replication package available at https://github.com/fahimeh1368/Thesis

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 do research.

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

Contents

List of Figures viii

List of Tables ix

1 Introduction 1

2 Factoring Expertise, Workload, and Turnover into Code Review Recommendation 3

2.1 Introduction . . . 3

2.1.1 Extension and Research Questions . . . 5

2.2 Background and Definitions . . . 8

2.2.1 The Ownership Recommenders . . . 8

2.2.2 The Turnover Mitigating Recommenders . . . 10

2.2.3 Workload Aware Recommenders . . . 13

2.2.4 Simulation and Evaluation Measures . . . 15

2.3 Project Selection and Data . . . 18

2.3.1 Gathering Data . . . 19

2.4 Results . . . 20

2.4.1 Empirical, RQ1: Review and Turnover . . . 20

2.4.2 Simulation, RQ2: Ownership Aware . . . 22

2.4.3 Simulation, RQ3: Turnover Aware . . . 24

2.4.4 Simulation, *Sofia*: Ownership and Turnover Aware . . . 27

2.4.5 Empirical, RQ4: Review Workload . . . 28

2.4.6 Simulation, RQ5: Workload Aware . . . 30
2.4.7 Simulation, RQ6: Ownership, Turnover, and Workload Aware . . . 31
2.5 Threats to Validity . . . 33
2.6 Discussion and Literature . . . 35
2.6.1 Understanding Code Review Practice . . . 35
2.6.2 Review Workload . . . 35
2.6.3 Turnover-Induced Knowledge Loss and Mitigation . . . 36
2.6.4 Recommenders . . . 37
2.7 Conclusion . . . 38

3 Adding a reviewer to spread knowledge **40**
3.1 Research Questions . . . 40
3.2 Background and Definitions . . . 42
3.3 Project Selection and Data . . . 44
3.4 Results . . . 44
3.4.1 RQ1, Recommenders^{++}: Which recommender suggest the best additional reviewer? . . . 44
3.4.2 RQ2, Two-weeks' Notice: What is the impact on *FaR* if we know in advance that someone will leave? . . . 45
3.4.3 RQ3 *FaR Replacement*: How well does *TurnoverRec* perform when we only replace reviewers on *Risky Pull Requests*? . . . 46
3.4.4 RQ4 *AwareFaR*: How well does the combination of *FaR*$^{++}$ when there are abandonded files, and *FaR Replacement* when there are only hoarded files work? . . . 47
3.5 Threats to Validity . . . 48
3.6 Discussion and Concluding Remarks . . . 49

4 Conclusion **51**

Bibliography **52**

List of Figures

Figure 2.1 An example illustrating the *GiniWork* measure using data from Kubernetes. We see that Actual Workload is highly skewed with the top 20% of reviewers doing over 80% of the work. The x = y line represents a Perfectly Equal Workload. The Gini coefficient of workload, *GiniWork*, is the area of A divided by the area of B. . 17

Figure 2.2 Inverted Lorenz curves of the review workload per quarter. No project has a normal or even distribution where 20% of the work is done by 20% of the development team. We see that review effort is highly concentrated with 20% of the developers performing around 80% of the reviews. 30

Figure 2.3 On the Rust project, 20% of the reviewers do 84.08% of the reviews (Actual Workload). We see that by suggesting past reviewers *RetentionRec* concentrates workload, and 1.5% of the reviewers do 59.19% of the reviews. In contrast, WhoDo distributes workload more evenly, with 20% of the developers doing 59.23% of the reviews. For *SofiaWL*, which also balances expertise and turnover risk, workload at 20% is 68.50%. 31

Figure 3.1 Left graph shows *FaR* in different approaches over five projects and the right one shows the amount of added work 50

List of Tables

Table 2.1 Systems under study 19

Table 2.2 Impact of reviews on *FaR* 21

Table 2.3 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 ΔGiniWork), and reduces files at risk to turnover (negative ΔFaR). . 22

Table 2.4 We vary the minimum number of knowledge developers per file: $k = 1, \ldots, 8$. In the paper, we use $k = 2$ because this is the first time more than 1 developer knows about a file, and it also provides the best balance in outcomes. 34

Table 3.1 The average change in *Expertise*, *FaR*, Gini and AddedPull Compared to the reviewers who actually performed the review in *FaR*$^{++}$ approach which adds reviewer when there is a *Risky File* in the pull request. AddedPull is a percentage of pull requests which are risky, and we add a reviewer. 45

Table 3.2 Outcome measures in different projects for *FaR*$^{++}$ by using *TurnoverRec* recommender to add a learner in the *Risky Pull Requests*. 45

Table 3.3 *FaR Replacement* approach outcome measures over five projects with *TurnoverRec* recommender. Replacement of actual reviewers with a suggested reviewer when there is a *Risky File* in a pull request. 47

Table 3.4 The outcome measure for *TurnoverRec* recommender with replacement of suggested learner in all the pull request to share the knowledge and there is a negative *Expertise* and decrease of *FaR* because of distribution of knowledge. 47

Table 3.5 The outcome measure in comparison to the reality when we add new reviewer when we have two-week notice leaver which cause a *Risky File* in a pull request. . 47
Table 3.6 *AwareFaR* outcome measures over five projects. Add learner when we have an abandoned file and replace a reviewer when we have a hoarder file in a pull request. 48

Chapter 1

Introduction

Code review is an important quality assurance practice used widely in both open source and in industry. In this thesis, we build upon prior works to understand code review workload of developers.

The first chapter of this thesis is an extension paper of Mirsaeedi and Rigby [33], our contribution in this chapter is that we use Gini coefficient instead of the review workload of the top 10 developers to provide a fuller understand of workload. We re-evaluate existing recommenders by three measures, *Expertise*, *FaR* and *GiniWork*. Higher *Expertise* but lower *FaR* and *GiniWork* is ideal in a recommender. We also replicate and evaluate *WhoDo* recommender, which is used in Microsoft[1]. Finally, we introduce a novel recommender "*SofiaWL* " that combine *WhoDo* recommender and *TurnoverRec* to improve workload measure of Sofia recommender, which is introduced by Mirsaeedi and Rigby and tries to balance turnover loss knowledge and *Expertise*.

The second chapter of this thesis is entirely novel. We change the simulation methodology of Mirsaeedi and Rigby. They randomly replace one of actual reviewers in all the pull requests by the top suggested reviewer and then evaluate the recommender. We want to change this approach, so, when we have *Risky File* which has zero or one knowledgeable developer, we add a learner reviewer to distribute knowledge among the team. In this approach, FaR^{++}, we decrease *FaR* substantially but add more workload to the team because when we have *Risky File* we need one more reviewer. Since we add reviewers, we measure how many additional reviews need to be performed through the added pull measure. Added Pull is the percentage of pull requests that we add a new reviewer.

Furthermore, in *FaR Replacement*, we randomly replace one of actual reviewers by a suggested reviewer when we have *Risky File*, not in all pull requests. This approach has better results in comparison to *TurnoverRec*. The *Expertise* improves from -25.30% to -1.28% because we decrease the number of learners in the *TurnoverRec* and just use learner when we have *Risky File*. Finally, we combine these approaches and get a balanced result. We use FaR^{++} when we have a *Risky File* in a pull request that no one know about that and when we have a *Risky File* that one developer knows about that replace randomly one of the actual reviewers with the suggested reviewer. We could reduce *FaR* more than *TurnoverRec* with 1.86% of added pull.

Chapter 2

Factoring Expertise, Workload, and Turnover into Code Review Recommendation

This chapter is a verbatim copy of the paper submitted to TSE: IEEE Transaction on Software Engineering. I performed the extension and major revisions for this TSE paper. Further description of my contributions are described in the text.

2.1 Introduction

Turnover on software projects is frequent and inevitable and leads to the loss of knowledge when developers leave a project [4, 49]. Turnover incurs substantial economic cost in recruiting and training new employees [40, 35], it reduces the productivity of development teams [35, 23], it leads to the loss of critical tacit knowledge [34, 35, 22], and has been shown to increase the number of defects in a product and reduce overall product quality [34, 14, 35].

Recent works have tried to mitigate the adverse impact of turnover through increasing knowledge retention by predicting leavers [29, 4, 11], planning for succession [53, 39, 49, 34], documenting knowledge, and persisting knowledge on StackOverflow and other internal QA forums [44, 39]. 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 [44].

In this work, we show that code review can mitigate turnover risk because it naturally distributes knowledge by exposing developers to code they have not authored 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 [46, 51, 2, 19, 7, 54]. Furthermore, studies have quantified the potential knowledge gained during code review [45, 51] and shown that developers review code in modules they have not modified [54]. 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 [3, 56, 60, 58, 59, 24, 21, 43] 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 measures 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 [27].

The second goal of this work is to make recommenders aware of developer workload. Core developers tend to be both committed and expert [47, 26], and are ideal review candidates to increase expertise and reduce the files at risk to turnover. However, core developers already have a high workload and a recommender that increases the workload of the top reviewers would be detrimental to the project. Recent work at Microsoft introduced the WhoDo recommender that factors developer review workload to ensure that expert developers are not overwhelmed by reviews [1]. We reimplement the WhoDo recommender and evaluate it on open source projects. We then develop a recommender, *SofiaWL*, that suggest experts that currently have low active workloads and learners who spread knowledge.

To evaluate recommenders we introduce three outcomes *Expertise*, *GiniWork*, and *FaR*. The first outcome ensures that expertise remains high for finding defects during review, *i.e.* the reviewers know about the files under review. The second, ensures that the top reviewers are not unreasonably

overworked due to always being the top recommendation, *i.e.* the change in area under the review workload distribution curve or Gini coefficient. The third outcome measures the number of files that are at risk to turnover, *FaR*, to ensure that knowledge is adequately distributed during review, *i.e.* how many people know about each file. 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.

2.1.1 Extension and Research Questions

This work is an extension over Mirsaeedi and Rigby's [33] previous work with the novel focus on code review workload. We adapt the economic measure of the Gini coefficient and the Lorenz curve to understand the concentration of reviewer workload (rather than the concentration of money) in methodology Section 2.2.4. We then re-evaluate the existing recommenders that are not aware of workload in Research Questions 2 and 3 to understand if they distribute or concentration reviewer workload. Research Questions 4, 5, and 6 are entirely new to this paper: we empirically describe reviewer workload, evaluate workload aware recommendation, and ensure that we continue to balance *Expertise* and reduce *FaR*. We provide a detailed discussion of reviewer workload and workload measures in Section 2.6.

We divide our research questions into two types: those that are purely *empirical* in nature and evaluate the actual state of the software project, and those that use *simulation* to evaluate the quality of each recommender on our outcome measures. We have six questions in total, and we provide the motivation for each question and a summary of the results below.

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

Our first research question is empirical in nature and quantifies the reduction in *FaR* from considering both authors and reviewers as knowledgeable. Recent studies have quantified knowledge loss from turnover on the basis of the commits that each developer has authored [49, 38]. 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 [54, 46, 51, 2].

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 79.79% of the total files are at risk to turnover. When we consider both authors and reviewers to be knowledgeable *FaR* drops to 32.59%. Code review naturally distributes knowledge.

Simulation RQ2, Ownership Aware: Does recommending reviewers based on code and review file 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 [51, 19] and experts who have modified or reviewed the files in the past [3, 27]. We implement 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%. For comparison purposes, we also re-implement cHRev which has been shown to outperform other recommenders [60]. When we 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 slightly reducing *GiniWork* by -1.71%. We conclude that concentrating expertise on the top developers make projects more susceptible to knowledge loss from turnover.

Simulation RQ3, Turnover Aware: 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 the degree of learning, *i.e.* knowledge distribution, and the retention potential of each reviewer. *LearnRec* ensures that a developer who has not reviewed or modified all of the files currently under review will be proposed. *Retention-Rec* recommender ensures that non-transient developers who have commitment to the project are recommended. When learning and retention factors are combined, *TurnoverRec*, it has the greatest reduction in turnover risk, ΔFaR -29.54, with a negligible impact on *GiniWork*, -0.28%. However, there is a substantial cost in the reduction of *Expertise*, -26.55%

We combined awareness of ownership with learning and retention by assigns either experts or

learners based on the files under review. *Sofia* [33] uses cHRev when the files under review are not at risk to turnover and uses *TurnoverRec* when few developers know about the files under review. *Sofia* increases the level of expertise during review by 6.27%, has a negligible impact on *GiniWork*, 0.59%, and reduces turnover risk with a ΔFaR of -28.27%.

Empirical RQ4, Review Workload: How is the review workload distributed across developers?

Recent works that interview reviewers, find that experts tend to be overloaded with their review workloads [51, 19] and that it is often difficult to find an available expert reviewer [19, 48, 56]. Before we design recommenders to balance workload, we want to understand the distribution of review effort on the projects understudy.

We replicate three studies that quantified review workload at different time scales: monthly [47], weekly [26], and daily [1]. We find that the median reviewers performs between 2 to 4 reviews per month. The review workload on open source projects is highly skewed, with the top reviewers at the 95th percentile reviewing between 25 to 51 monthly reviews, 11 to 24 weekly reviews, and 5 to 8 daily reviews. At a quarterly level, we see that the top 20% of reviewers accounts for an average of 80.19% of all reviews, a clear Pareto distribution of review workload. We demonstrate the concentration of review workload through an adapted Gini coefficient, *GiniWork*, and Lorenz curves.

Simulation RQ5, Workload Aware: WhoDo is designed to be workload aware, but can it also balance *Expertise*, *GiniWork*, and *FaR*?

In prior work, review recommenders focus on finding experts and do not measure the impact of workload [3, 56, 60, 58, 24, 21]. The review recommender recently introduced at Microsoft, WhoDo, considers the ownership of prior commits and reviews and the average daily workload of each developer [1]. They use the change in *average* workload as an outcome measure but do not consider the impact of WhoDo on ΔExpertise, ΔFaR, or the overall distribution of workload, ΔGiniWork. We re-implement the code for WhoDo and evaluate it in the context of open source projects on our multi-objective outcome measures. WhoDo increases *Expertise*, 4.45%, and outperforms the existing recommenders on *GiniWork*, -19.16%. Unfortunately, WhoDo drastically increases FaR by 40.97% which puts many more files at risk to developer turnover.

Simulation RQ6: Ownership, Turnover, and Workload Aware: Can we combine the recommenders to balance *Expertise*, *GiniWork*, and *FaR*?

The ownership based recommenders are aware of the files that are recently authored or reviewed by developers (*e.g.,* cHRev [60]), WhoDo [1] is aware of developer workload, and *Sofia* [33] is aware of files that are potentially at risk to turnover. We combine these recommenders to create a novel recommender that is aware of the ownership, workload, and knowledge distribution. When there are risky files in the pull requests we use *TurnoverRec* to distribute knowledge, when no files are at risk to turnover we use the *WhoDo* recommender to find an expert developer with a low workload.

With SofiaWorkLoad, (*SofiaWL*), *Expertise* is increased by 3.20%, *GiniWork* is reduced by -12.00%, and *FaR* is reduced by -23.92%. While there are recommenders that can outperform *SofiaWL* on individual outcomes only *SofiaWL* can balance all three outcomes simultaneously.

This thesis is organized as follows. In Section 2.2, we provide the study background as well as defining our measures, review recommenders, scoring functions, and simulation methodology. In Section 2.3, we describe the projects under study and project data. In Section 2.4, we present results for each of our research questions. In Section 2.5, we discuss threats to validity. In Sections 2.6 and 2.7, we discuss our findings in the context of the existing literature and conclude the thesis.

2.2 Background and Definitions

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

2.2.1 The Ownership Recommenders

The influence of code ownership on code quality has been extensively investigated in the literature[6, 42, 14, 54]. Ownership is a human factor that helps with finding knowledgeable developers that can

be accountable for a particular part of code or task [37]. *Developer Recommenders* use ownership to automatically assign tasks to experts [25]. Researchers have used a wide range of granularity, from lines [15, 16, 42] to modules [6], to estimate ownership of developers. Studies on code review find that code owners are usually selected to review changes [51, 2, 19]. In this work we develop two simple scoring functions for review recommendation based on ownership.

AuthorshipRec. Bird *et al.* [6] define 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.

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

RevOwnRec. Thongtanunam *et al.* [54] 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.

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

cHRev. There is a large literature on review recommendation [60, 3, 56, 58, 59, 43, 21, 30]. We note that we did not find a replication package or recommender implementation for any of these works. We re-implement cHRev [60] because it includes a wide range of factors in its recommendation and has a higher accuracy than the other review recommenders such as RevFinder [56].

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.

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

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.

2.2.2 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 [46] report that code review increases the number of files developers see by between 100% and 150%. We speculate that code review can be effective in mitigating the turnover-induced knowledge loss. Based upon this idea, we design reviewer recommenders that spread knowledge to developers with a high retention potential.

Distributing Knowledge

We 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.

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 learn about. 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:

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

Developer Retention

Developers who have made substantial recent contributions to a project have demonstrated a high degree of commitment to the project[11, 52]. 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. We use the retention potential for a one-year period, because prior work showed that developers are considered learners up to one year, and that the number of commits they make plateaus at one year [61]. In practice, the development team can adjust this ratio based on their turnover rate.

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 $\text{ConsistencyRatio}_{365}(D)$ 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 are unlikely to leave the project. The scoring function for a candidate review, D is

RetentionRec$(D) =$

$$C1 \cdot \text{ConsistencyRatio}_{365}(D) * C2 \cdot \text{ContributionRatio}_{365}(D) \tag{8}$$

$C1$ and $C2$ are constant coefficients and can weight consistency and contribution. In our simulations, we consider them to be equally important, $C1 = C2 = 1$. In practice, developers could add weights to this equation to emphasize, for example, consistency over total contributions.

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

$$\textit{TurnoverRec}(D, R) = C1 \cdot \textit{LearnRec}(D, R) * C2 \cdot \textit{RetentionRec}(D) \tag{9}$$

The importance of learning over retention for a project can be adjusted based on the weights, $C1$ and $C2$. In this simulation, we consider learning and retention equally important and set $C1 = C2 = 1$. Project managers can adjust these weights based on their specific goals.

Sofia: *TurnoverRec* and *cHRev* combined.

The *Sofia* recommender distributes knowledge when there are files under review that are abandoned or hoarded, *TurnoverRec*, and suggests experts, *cHRev*, when all files already have multiple knowledgeable developers. The equation shows the *Sofia* scoring function:

Sofia$(D, R) =$

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

If there are zero knowledgeable people for a file, then the file is an abandoned file, and if there is only one knowledgeable developer for a file, then the file is hoarded by one person. We set $k = 2$ as the threshold for *TurnoverRec* because this is the smallest number where there is some distribution of knowledge. When there are more than two knowledgeable developers, we recommend an expert with *cHRev*.

2.2.3 Workload Aware Recommenders

***WhoDo*.** The *WhoDo* recommender was developed at Microsoft by Asthana *et al.* [1], and it recommends reviewers based on their workload and recent expertise. The review recommender scoring function is defined as

$$\begin{aligned} \text{Score}(D) = C1 \cdot \sum_{f \in F} n_{\text{change}}(D, f) \frac{1}{t_{\text{change}}(D, f)} + \\ C2 \cdot \sum_{p \in P} n_{\text{change}}(D, p) \frac{1}{t_{\text{change}}(D, p)} + \\ C3 \cdot \sum_{f \in F} n_{\text{review}}(D, f) \frac{1}{t_{\text{review}}(D, f)} + \\ C4 \cdot \sum_{p \in P} n_{\text{review}}(D, p) \quad \frac{1}{t_{\text{review}}(D, p)} \end{aligned} \tag{11}$$

Where D is the candidate reviewer, f is the files in the change, p is the set of last-level parent directories that are changed. The number of commits and reviews that D has done to the file f is $n_{\text{change}}(D, f)$ and $n_{\text{review}}(D, f)$, respectively. $n_{\text{change}}(D, p)$ is the number of times reviewer D has committed changes within directory p and $n_{\text{review}}(D, p)$ is the number of reviews D has performed on files in directory p.

$t_{\text{change}}(D, f)$ and $t_{\text{change}}(D, p)$ are the number of days since developer D changed a file and

parent directory respectively. $t_{\text{review}}(D, f)$ and $t_{\text{review}}(D, p)$ are the number of days since a reviewer D reviewed the file or directory, respectively. Developers who have recently modified or reviewed the files in the directory in current change will have a higher WhoDo score.

$C1$, $C2$, $C3$, and $C4$ are constant coefficients and can weight authorship and reviewership. We follow the creators of WhoDo and set them to one to equally consider authorship and reviewership.

Workload balancing. WhoDo balances the workload of reviewers by weighting the score by the number of active, open reviews that are assigned a candidate reviewer.

$$\text{Load(r)} = e^{\theta.TotalOpenReviews} \tag{12}$$

θ is a parameter between 0 and 1 to control the amount of load balancing and we follow the authors of WhoDo set it to 0.5.

WhoDo's final scoring function is

$$\textit{WhoDo}(r) = \frac{\text{Score}(r)}{\text{Load}(r)} \tag{13}$$

***SofiaWL*: *TurnoverRec* and *WhoDo* combined.** To make *Sofia* aware of workload, we replace cHRev with WhoDo. Specifically, when we have hoarded or abandoned files in a review we use *TurnoverRec* to distribute knowledge, otherwise, we use the *WhoDo* recommender to find an expert developer with a low active review workload. The equation shows the *SofiaWorkLoad* scoring function:

$\textit{SofiaWL}(D, R) =$

$$\begin{cases} \textit{TurnoverRec}(D, R) & \text{if } |\text{Knowledgeable}(f)| \leq k, \text{ any } f \mid f \in R \\ \textit{WhoDo}(D, R) & \text{otherwise} \end{cases} \tag{14}$$

Following the same rationale fully discussed for Equation 10, we set $k = 2$ because this is the smallest number where there is some knowledge distribution. We also conduct a sensitivity analysis varying k from 1 to 8 in the threats to validity section.

Our replication package contains the code implementing each scoring function, the raw and

simulated data, and other less promising combinations of recommenders [20].

2.2.4 Simulation and Evaluation Measures

To evaluate reviewer recommenders, prior works made recommendations for each existing review and compared their result against the actual reviewers who performed the review [60, 3, 56, 58, 59, 43, 21]. 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 Kovalenko *et al.*'s [27] interviews with developers who state that the recommenders rarely provide additional value because they suggest obvious expert candidate reviewers. This problem is also inherent in the outcome measure, which assumes that the actual reviewers were the best, *i.e.* "correct" reviewers. Kovalenko *et al.* [27] 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 review. The recommenders consider the files under review and according to the scoring functions defined in Sections 2.2.1 to 2.2.3, 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 drastically disrupting the peer review process and because Kovalenko *et al.* [27] 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*, the *GiniWork* of reviewers, and the number of files at risk to turnover, *FaR*. We measure the change in the outcomes over the standard quarterly period [49, 38]. 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 both expertise and core workload with a percentage increase in the knowledge distribution of the development team and fewer files at risk to turnover. We define each outcome measure below.

***Expertise* outcome measure.** Having high expertise ensures having high quality code review [13, 2, 8]. 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.

$$\text{Expertise}(Q) = \sum_{R}^{Reviews(Q)} \frac{\text{FilesReviewersKnow(R)}}{\text{FilesUnderReview(R)}} \tag{15}$$

***GiniWork* outcome measure.** Miraseedi and Rigby [33] only considered the top 10 core reviewers when measuring the change in workload. This effectively made adding learners outside of the core "free" because their change in workload was not measured. In contrast, Asthana *et al.* [1] measure the average number of open reviews per developer per day which is appropriate when when reviews are spread evenly across developers but is misleading when a core group of developers does a large number of reviews. In our empirical results section, we show that the top 20% of reviewers do between 75.85% and 84.08% of the work, depending on the project. Prior work divided developers into core and non-core developers, however, this dichotomy would require two measures and involves thresholds, *e.g.,* the change for the top 20% vs the bottom 80% of reviewers [36, 49].

To avoid threshold, we adapt the economic measure of the Gini coefficient and the Lorenz curve to understand the concentration of reviewer workload (rather than the concentration of money). The Lorenz curve is a percentage-percentage plot. Figure 2.1 is an inverted Lorenz curve to show the top percentile of reviewers rather than the bottom percentile of reviewers. The $x = y$ line shows an even normal distribution, where 20% of the reviews are done by 20% of the reviewers. The curve showing Actual Workload is highly skewed with 20% of the reviewers doing 82.02% of the reviews (see Section 2.4.5 for results and discussion). To convert the Lorenz curve to a single workload concentration value, we use the Gini coefficient of workload, *GiniWork*. *GiniWork* is defined as the

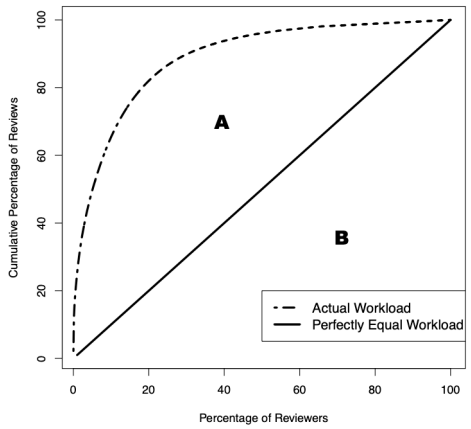

Figure 2.1: An example illustrating the *GiniWork* measure using data from Kubernetes. We see that Actual Workload is highly skewed with the top 20% of reviewers doing over 80% of the work. The x = y line represents a Perfectly Equal Workload. The Gini coefficient of workload, *GiniWork*, is the area of A divided by the area of B.

area between the Lorenz curve and the perfect equality line divided by the total area under the line of perfect equality. Given the areas A and B shown in the figure, the definition is as follows:

$$\textit{GiniWork}(Q) = A/B \tag{16}$$

Files at Risk, *FaR*, outcome measure. We need to quantify the project's exposure to turnover from knowledge loss. Building on Rigby *et al.*'s [49] 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, Files at Risk is defined as

$$\textit{FaR}(Q) = \{\, f \mid f \in \text{Files}\,, |\text{ActiveDevs}(Q, f)| \leq 1\} \tag{17}$$

Files with no active developers are abandoned. Files with only one active developer are at risk to turnover because the knowledge is hoarded by one person. We consider files with two or more active developers to lower risk risk to turnover.

Percentage change in the measures relative to actual values. 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 18,19, and 20 to report the percentage change of *Expertise*, *GiniWork*, and *FaR*, respectively.

$$\Delta\text{Expertise}(Q) = (\frac{\text{Simulated}\mathit{Expertise}(Q)}{\text{Actual}\mathit{Expertise}(Q)} - 1) * 100 \tag{18}$$

$$\Delta\text{GiniWork}(Q) = (\frac{\text{Simulated}\mathit{GiniWork}(Q)}{\text{Actual}\mathit{GiniWork}(Q)} - 1) * 100 \tag{19}$$

$$\Delta\text{FaR}(Q) = (\frac{\text{Simulated}\mathit{FaR}(Q)}{\text{Actual}\mathit{FaR}(Q)} - 1) * 100 \tag{20}$$

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

2.3 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 reviewers 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 [17, 20]. 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.

Table 2.1: Size of projects under study. We explicitly select for large, long-lived projects.

Project	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

Rust and Kubernetes are community driven OSS projects. Table 2.1 provides summary statistics, including the number of files, pull requests, and commits. Our replication package contains the project data [20].

2.3.1 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 [18]. 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 [20].

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 *et al.*'s [5] and Canfora *et al.*'s [9].

Leavers. Robillard *et al.* [50] 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 and bots. Rigby *et al.* [49] 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 the license and copyright of files, or importing a large chunk of code from another version control system. 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*. 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. The list of excluded bot users and commits are in our replication package [20].

2.4 Results

In this section, we discuss the results for our research questions relating to (1) an empirical study of knowledge distribution during review, (2) recommendations based on ownership, including *cHRev*, (3) learning and retention aware recommenders, including *Sofia*, (4) an empirical quantification of the workload of reviewers, (5) recommendations based on workload, including *WhoDo*, and a recommender that combines ownership, learning and retention, and workload, *i.e. SofiaWL*. We make three notes. First, we note that RQ1 and RQ4 do not involve simulation and are empirical results based on the actual reviews and commits. Second, we note that the MRR outcomes do not involve simulation and instead report how accurately the recommender predicts the actual reviewers. Third, simulations are run for each recommender and we note the changes in ΔExpertise, ΔGiniWork, and ΔFaR as a percentage difference relative to the actual values for each project. Table 2.3 shows the average for each outcome across all projects, the values for each project are in the body of the paper.

2.4.1 Empirical, 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 [49, 38]. The assumption in these works, is that knowledge is

Table 2.2: 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	Authors	Authors + Reviewers
CoreFX	89.46%	24.74%
CoreCLR	86.65%	45.56%
Roslyn	68.00%	22.14%
Rust	78.10%	44.51%
Kubernetes	76.78%	26.04%
Average	79.79%	32.59%

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, *e.g.,* [51].

To assess the extent that the project is at risk to knowledge loss from turnover, we measure *FaR*, see Equation 17, 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 [49, 38]. We then calculate *FaR*, which considers both authors and reviewers as knowledgeable.

Table 2.2 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 7,648, 3,704, 5,602, 2,932, and 5,448 files for CoreFX, CoreCLR, Roslyn, Rust, and Kubernetes, respectively. As a percentage of the codebase, between 68% and 89% 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 1,988, 2,000, 1,918, 1,958, 1,877, respectively. As a percentage of the codebase, between 22% and 45% 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 74.00%, 46.00%, 65.76%, 33.21%, and 65.54% 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 [49, 38].

Table 2.3: 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 ΔGiniWork), and reduces files at risk to turnover (negative ΔFaR).

Recommender	MRR	ΔExpertise	ΔGiniWork	ΔFaR	Recommender Scoring Functions
AuthorshipRec	0.49	11.29%	-1.78%	25.25%	Author Commit Ownership
RevOwnRec	0.45	15.17%	7.19%	65.19%	Review Ownership
cHRev	0.52	11.11%	-1.71%	4.15%	Recent Review Ownership
LearnRec	0.12	-35.13%	-19.53%	63.04%	Learner: spread knowledge
RetentionRec	0.39	16.59%	8.64%	-15.91%	Retention: consistent contributions
TurnoverRec	0.19	-25.30%	-0.28%	-29.54%	Learner and Retention
Sofia	0.43	6.27%	0.59%	-27.28%	(Recent Review Ownership) or (Learner and Retention)
WhoDo	0.22	4.35%	-19.16%	40.97%	Recent Ownership and Workload
SofiaWL	0.22	3.20%	-12.00%	-23.92%	(Recent Ownership and Workload) or (Learner and Retention)

When only authors are considered knowledgeable an average of 79.79% of files are at risk to turnover. When reviewers are also considered knowledgeable, the *FaR* average is 32.59%. There is substantial knowledge distribution during code review.

2.4.2 Simulation, RQ2: Ownership Aware

Does recommending reviewers based on code and review file 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 [51, 19] and experts who have modified or reviewed the files in the past [3, 27]. 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 [25, 31, 37] that use historical authorship data to recommend reviewers [60, 21]. We partially reproduce these authorship recommendations by using the scoring function defined in Equation 1. We use the simulation method described in Section 2.2.4 and evaluate the impact of *AuthorshipRec* on MRR, ΔExpertise, ΔGiniWork, and ΔFaR. The average values are shown in Table 2.3.

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 *GiniWork* increases for Rust by 4.22%, while it is reduced by -5.83%, -2.37%,-3.28% and -1.64% for CoreFx, CoreCLR, Roslyn and Kubernetes, with an average of -1.78%. Although ΔExpertise is high for each review, ΔFaR to turnover 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 [3, 24, 56, 58, 59]. 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 *GiniWork* of 2.75%, 10.58%, 4.63%, 9.77%, 8.24% with an average of 7.19%. By concentrating recommendations on past reviewers of a file, 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%.

cHRev [60]. This recommender builds upon prior work that leverages information in past reviews [25], but also accounts for the number of days a candidate reviewer has worked on a file,

and the recency of this work (See Section 2.2.1 for further details). *cHRev* has been show to outperform the other review history based recommenders, including RevFinder [54]. 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 2.2.4 and evaluate the impact of *cHRev* on MRR, ΔExpertise, ΔGiniWork, and ΔFaR.

In the original cHRev paper, the authors report an average MRR of .67 across four projects [60]. 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 other ownership recommenders, *cHRev* increases the ΔExpertise in reviews by 9.84%, 7.27%, 16.45%, 8.22%, and 13.81%, respectively, with an average of 11.11% across projects. ΔGiniWork decreases by -2.79% -1.51% -0.80% -1.20% -2.26% with the average of -1.71%. *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%.

> Ownership aware recommenders concentrate knowledge on a expert authors and reviewers. *cHRev* is the most accurate ownership recommender suggesting the actual reviewers with an MRR of 0.52. It increases the degree of *Expertise* during review by 11.11%, with a slight decrease in *GiniWork* of 1.71%. However, the files at risk to turnover increase with an average ΔFaR of 4.15% as knowledge is concentrated on expert reviewers.

2.4.3 Simulation, RQ3: Turnover Aware

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. 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 [51, 7, 2]. Building on this idea, in Section 2.2.2, 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 2.2.4 and evaluate the impact of *LearnRec* on MRR, ΔExpertise, ΔGiniWork, and ΔFaR with the average outcomes shown in Table 2.3.

As expected, *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 from the review 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 *GiniWork* is drastically reduced as fewer expert reviewers are assigned reviews: -16.53%, -21.41%, -21.60%, -23.94% and -14.18% with the average of -19.53%. This is the largest decrease of any recommender. 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 are less committed to the project.

RetentionRec. Assigning reviews to transient developers may distribute knowledge, but does not reduce turnover. In Section 2.2.2, 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 is similar 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 *GiniWork* of 9.76%, 16.38%, 12.13%, 11.81% and a decrease of -6.84% for Kubernetes with an average of 8.64%. 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%. *RetentionRec* selects committed developers who are unlikely to leave the project.

TurnoverRec. We showed that distributing knowledge through *LearnRec* does not alleviate knowledge loss and *RetentionRec* increases the *GiniWork*. 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*, 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 *GiniWork* there is only a slight increase of 1.24%, 0.42% and 1.38% for CoreFX, CoreCLR and Kuberenest and slight decrease of -1.19% and -3.29% for Roslyn and Rust with an average of -0.28%. 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 *GiniWork*, 1.07. The low MRR value of 0.19 indicates that developers naturally focus on reviewers with greater expertise than *TurnoverRec*.

2.4.4 Simulation, *Sofia*: Ownership and Turnover Aware

We have seen that the ownership recommenders, *e.g.,* cHRev, can increase expertise, and the combination of learning and retention aware recommender, *TurnoverRec*, can reduce the files at risk to turnover. We combine these recommenders in Equation 10. *Sofia* 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 a quarter contain files at risk: 25.18%, 26.13%, 29.82%, 29.41%, and 17.17%, respectively. The remaining pull requests use *cHRev* recommendations to ensure sufficient expertise. We use the simulation method described in Section 2.2.4 and evaluate the impact of *Sofia* on MRR, ΔExpertise, ΔGiniWork, and ΔFaR with average outcomes shown in Table 2.3.

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 *GiniWork*, we see a reduction of -0.55% and -0.92% in CoreFX and Kubernetes and a slight increase of 1.96%, 1.27% and 1.23% for CoreCLR, Roslyn and Rust. The average of ΔGiniWork is a minor increase of 0.59%. *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 *GiniWork*, 0.59%, 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.

2.4.5 Empirical, RQ4: Review Workload

How is the review workload distributed across developers?

With the exception of *LearnRec* that distributed knowledge to learners at the great expense of reducing *Expertise* and increasing *FaR*, none of recommenders we evaluated were able to have a substantial reduction in *GiniWork*. Before we describe recommenders designed to be aware of a candidate reviewer's workload, we want to understand the current distribution of review effort and to replicate the workload measures and results of three prior works [47, 26, 1].

Rigby *et al.* [47] found that on six major open source projects including Apache and Linux, the median reviewer participates in only 2 to 3 reviews per month. In contrast, the top reviewers, *i.e.* those at the 95th percentile, participated in between 13 and 36 reviews, depending on the project. Replicating these *monthly* results on CoreFX, CoreCLR, Roslyn, Rust, and Kubernetes, we find that the median reviewer participates in 2, 2, 4, 2, and 3 reviews per month, respectively. Top reviewers at the 95th percentile are involved 28, 25, 51, 34, and 37 reviews per month. Our results are consistent with Rigby *et al.*'s [47] and show the same skew in effort of the median reviewer vs the top reviewers.

Kononenko *et al.* [26] interviewed Mozilla developers and found a similar pattern at the weekly level with Mozilla's top reviewers having between 11 and 20 reviews per week. We replicate these *weekly* results and find that the median reviewer participates in 2, 1, 2, 2, and 2 reviews per week, with the 95th percentile top reviewers participate in 12, 11, 24, 16, and 20. Although Kononenko *et al.* interviewed developers, our measured 95th percentile range is 11 to 24, which clearly confirms their interviewed values and validates them on different open source projects.

At Microsoft, Asthana *et al.* [1] find that the *daily* per developer average number of open pull request reviews is between 1.31 and 2.68 on small projects and 8.4 on a large project. We find the average review daily workload is 1.84, 2.57, 2.48, 2.18, and 2.91. However, the distribution is skewed and find with the that median daily values of 1, 1, 2, 1, and 1. At the 95th percentile, the daily number of reviews is 7, 5, 7, 8, and 8. While the average daily values on our studied projects are comparable with the Microsoft values, unfortunately, Asthana *et al.* did not publish the median or skew of their data. Future work is necessary to determine if review effort is also skewed in a commercial dataset.

Our replication results show that review effort is not evenly distributed across the open source development teams. We examine the distribution of review effort to further understand the skew. Prior works of developer effort examined the commit workload of the top 20% of committers, *e.g.,* [36, 49]. Figure 2.2 plots the cumulative distribution as the percentage of reviews vs percentage of reviewers. We see that the distributions are highly skewed and that the top 20% of reviews account for 82.30%, 75.85%, 76.69%, 84.08%, and 82.02% depending on the project with an average of 80.19%. Since the review workload on these open source projects shows a strong Pareto "80-20 rule", the average workload is misleading.

As we discussed in Section 2.2.4, using a percentile cut-off also introduces an arbitrary threshold, and instead we calculate the Gini coefficient, *GiniWork*. In the figure, we see that the area under the curve is similar for the actual workload of each project. In our simulations, we calculate the change in area between the simulated and actual workloads, see Equation 16. As we can see in Figure 2.3, the workload area varies dramatically depending on the recommender. Understanding the distribution of review effort, we are now in a position to distribute work more evenly, while ensuring that *Expertise* remains high and that *FaR* is reduced.

The distribution of review effort is highly skewed on open source projects. We find that the median daily load is between 1 and 2 reviews, while at the 95th percentile the load is 5 to 8 reviews per day. We find that 20% of the reviewers are responsible for 80.19% of the reviews. By comparing the Lorenz curve and *GiniWork* coefficient, we measure the change in area under the curve to acknowledge the skewed distribution of review effort.

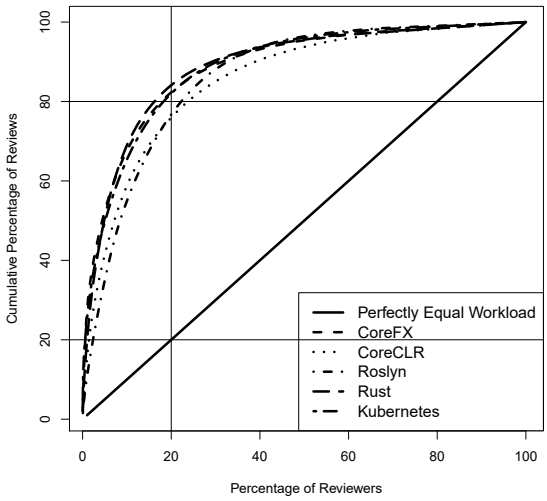

Figure 2.2: Inverted Lorenz curves of the review workload per quarter. No project has a normal or even distribution where 20% of the work is done by 20% of the development team. We see that review effort is highly concentrated with 20% of the developers performing around 80% of the reviews.

2.4.6 Simulation, RQ5: Workload Aware

WhoDo is designed to be workload aware, but can it also balance Expertise, GiniWork, and FaR?

In prior work, review recommenders focus on finding experts and do not measure the impact of recommendation on reviewer workload [3, 56, 60, 58, 24, 21]. The review recommender recently introduced at Microsoft, WhoDo [1], recommends developer based on the files and directories they have reviewed or committed to in the past. They then weigh the recommender's score by the daily review workload of each candidate, see Section 2.2.3. We re-implement WhoDo and evaluate it in the context of open source projects using the simulation method described in Section 2.2.4 and MRR, ΔExpertise, ΔGiniWork, and ΔFaR as outcome measures.

WhoDo balances the load of experts resulting in a diverse set of reviewers and low MRR of 0.30, 0.29, 0.22, 0.16, 0.17 for CoreFX, CoreCLR, Roslyn, Rust, and Kubernetes, respectively. The average MRR across projects is 0.22, which means the actual reviewer rank is averaged to 4.54.

From the simulations, we see that WhoDo has a nominal decrease in ΔExpertise for Rust, -0.16%, but substantial increases on the other projects: 4.14%, 2.01%, 10.16%, and 5.63% with an

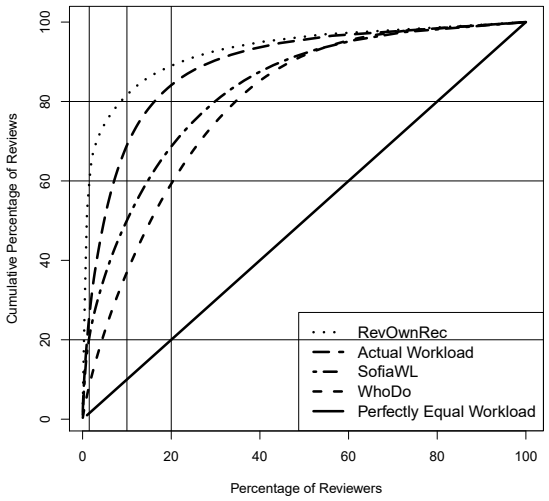

Figure 2.3: On the Rust project, 20% of the reviewers do 84.08% of the reviews (Actual Workload). We see that by suggesting past reviewers *RetentionRec* concentrates workload, and 1.5% of the reviewers do 59.19% of the reviews. In contrast, WhoDo distributes workload more evenly, with 20% of the developers doing 59.23% of the reviews. For *SofiaWL*, which also balances expertise and turnover risk, workload at 20% is 68.50%.

average of 4.35%. The design of WhoDo is workload aware and we see a correspondingly large decrease in ΔGiniWork of -17.73%, -12.41%, -21.15%, -24.65% and -19.86% with an average across projects of -19.16%. The number of files at risk increases dramatically for WhoDo: 36.66%, 9.53%, 64.85%, 33.77%, 60.07%, with an average of 40.97%. The side-effect of WhoDo recommending developers with a lower daily workload, is that suggested developers have fewer contributions, and are less commitment to the project, which increases turnover and *FaR*.

WhoDo recommends substantially different reviewers with a low MRR of 0.22. However, it is designed to find experts that have a low workload and it increases ΔExpertise, 4.35%, while drastically reducing ΔGiniWork, -19.16%. Unfortunately, by recommending reviewers with low workload it suggests transient developers and drastically increases ΔFaR by 40.97%.

2.4.7 Simulation, RQ6: Ownership, Turnover, and Workload Aware

Can we combine the recommenders to balance Expertise, GiniWork, and FaR?

The ownership based recommenders are aware of the files that are recently authored or reviewed by developers (*e.g.,* cHRev [60]), WhoDo [1] is aware of developer workload, and *Sofia* [33] is aware of files that are potentially at risk to turnover. We combine these recommenders to create a novel recommender that is aware of the ownership, workload, and knowledge distribution. When there are risky files in the pull requests we use *TurnoverRec* to distribute knowledge, but when no files are at risk to turnover, we use *WhoDo* to find an expert developer with a low workload (see Equation 14). We evaluate *SofiaWL* using the simulation method described in Section 2.2.4 and the outcome measures MRR, ΔExpertise, ΔGiniWork, and ΔFaR.

The MRR for *SofiaWL* on CoreFX, CoreCLR, Roslyn, Rust, and Kubernetes the MRR is 0.32%, 0.27%, 0.19%, 0.17%, and 0.17, respectively. The average MRR across projects is 0.22, which means that the actual reviewer is ranked 4.54. Both *WhoDo* and *SofiaWL* have higher MRR than recommenders focussed on learning, *TurnoverRec*, but lower than the pure ownership recommenders, *e.g., cHRev*.

SofiaWL inherits the increase in expertise from the ownership aware recommenders with per project increases of 1.79%, 0.23%, 6.02%, 2.06%, and 5.9%, and an average across projects of 3.2%. While the increase is still reasonable, it 1.15 percentage points lower than *WhoDo* and is lowest of the recommenders, including *Sofia*, that are aware of ownership.

SofiaWL's awareness of *GiniWork* leads to a per project decreases of -9.00%, -5.73%, -14.55%, -15.55%, and -15.20%. The across project average is -12.00 is substantial but less impressive than *WhoDo*'s decrease in *GiniWork* of -19.16%.

This workload decrease is coupled with an across projects decrease of -23.92% in *FaR*, with respective project decreases of -28.5%, -5.28%, -35.94%, -19.61%, and -30.28. While *TurnoverRec* has a decrease of -29.54% in *FaR*, *SofiaWL* maintains a strong decrease without the drastic reduction in *Expertise*.

With *SofiaWL*, *Expertise* is increased by 3.20%, *GiniWork* is reduced by -12.00%, *FaR* is reduced by -23.92%. While there are recommenders that can outperform *SofiaWL* on individual attributes, only *SofiaWL* can balance all three simultaneously.

2.5 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 [60, 56], ownership [19, 15, 46], and turnover [49, 38], 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.

We have considered all files to be of equal importance in our simulation. Future work could use measures of file churn, complexity, and centrality to factor the importance of a file into the knowledge loss calculation.

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.

Random replacement of a reviewer. In our simulation, we randomly select one of the reviewers in each review to be replaced with the top recommended reviewer for each recommender (see Section 2.2.4). Table 2.1 showed that we examine over 80k reviews across 5 projects, making it unlikely that this random selection will lead to systematic bias.

Replication and Reproducability. Existing recommenders including ReviewBot [3], RevFinder [54], cHRev [60], and WhoDo [1] 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 and WhoDo because it is aware of each reviewer's workload. 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 and

Table 2.4: We vary the minimum number of knowledge developers per file: $k = 1, \ldots, 8$. In the paper, we use $k = 2$ because this is the first time more than 1 developer knows about a file, and it also provides the best balance in outcomes.

k	ΔExpertise	ΔGiniWork	ΔFaR
1	6.79%	-3.18	-4.45%
2	3.20%	-12.00	-23.92%
3	2.77%	-2.29	-22.02 %
4	0.39%	0.11%	-23.42%
5	-3.78%	0.40%	-21.69%
6	-4.17%	-3.75%	-25.88%
7	-7.01%	0.68%	-24.68%
8	-7.48%	3.14%	-23.35%

data available for future researchers [20].

Sensitivity analysis for *SofiaWL*. *SofiaWL* is a bi-modal recommender that suggests experts with low workload when no files are at risk to turnover, with *WhoDo*, and it spreads knowledge with *TurnoverRec* when there are files at risk. We set $k = 2$ as the threshold for using *TurnoverRec* because this is the largest number below which files are hoarded or abandoned. We conduct a sensitivity analysis with $k = 1, \ldots, 8$. The average trend across projects for each k is show in Table 2.5. We see *Expertise* decreases with increase in k because higher k leads to more *TurnoverRec* recommendations that spreading knowledge. We see that $k = 2$ is the best point for the *GiniWork* because there are more *WhoDo* recommendations that distributes workload. At $k = 1$, knowledge is concentrated on experts with recommendations from *WhoDo*, afterwards, ΔFaR remains consistently below -20%, indicating that additional knowledge spreading above two knowledge people has little impact on the files at risk to abandonment.

Each of our recommenders is designed with a particular focus. Table 2.3 that contains the simulation results for each recommender demonstrates the tradeoffs that developers and managers can make for their projects. Managers can tune the scoring functions described in Section 2.2 based on their projects specific needs.

2.6 Discussion and Literature

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

2.6.1 Understanding Code Review Practice

Fagan [13] 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 the latter stages increased overall effort. In the subsequent 40 years, code review has been extensively studied. Early works focused on examining the process [13, 12]. However, Porter *et al.* [41] demonstrated that process was much less important than ensuring expertise during review. Current code review practice favors a lightweight process that focuses on expert discussion of changes to the system [7, 48, 2, 46, 8] that still improves software quality [47, 32, 28]. We show that *RetentionRec* has the highest increase in *Expertise* 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%.

2.6.2 Review Workload

Recent works that interview reviewers, find that experts tend to be overloaded with their review workloads [51, 19] and that it is often difficult to find an available expert reviewer [19, 48, 56]. Moreover, high overall workload can lead to poor review participation[55], requesting feedback from experts can lead to delays from lack of availability, and can result in fewer opportunities for knowledge dissemination [19]. In Section 2.4.5 and Figure 2.2, we show that the distribution of review workload is highly skewed with the top 20% of reviewers performing 80.19% of the reviews. While prior works have measured the number of reviews that developers perform over varying time periods [47, 26, 1], we are the first to demonstrate this strong Pareto principle with review workload.

Our recommenders show that the workload distribution can be changed without sacrificing *Expertise*. For space reasons we cannot plot the 45 lines that would represent all 9 recommenders and 5 projects (Please see the figures in our replication package [20]). In this discussion and Figure 2.3 we focus on the Rust project. On the Rust project, 20% of the developers do 84.08% of the reviews. Prior works that recommend reviews solely on the basis of the files that have been reviewed in the past [60, 54], further concentrate the review effort. *RevOwnRec*, which represents this category of recommenders has the top 20% of reviewers doing 88.97% of the reviews. However, the figure clearly shows that the concentration of effort is more extreme with the top 10% of reviewers doing 81.62% and 1.5% of reviewers doing 59.19% of the effort. This concentration of effort is extreme when we consider that 1.5% of rust developers actually do only 25.39% of the work.

In contrast, WhoDo does an excellent job of reducing workload. In the figure we see that the top 20% of reviewers do 59.23% of the reviews. While WhoDo balances expertise, it drastically increases turnover risk. For *SofiaWL* the top 20% do 68.50% of the reviews while maintaining expertise and reducing turnover risk. Clearly there is no direct tradeoff between the outcomes, and development teams can experiment with scoring functions to reduce the review workload on experts and spread knowledge.

2.6.3 Turnover-Induced Knowledge Loss and Mitigation

Turnover deprives the project of the leaver's experience and knowledge [22, 57] and has been shown to increase the number of defects [35]. 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 [49, 38]. 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[46, 51, 2, 19, 7]. Two separate studies quantified the knowledge gained during code review and showed that at both Google [51] and Microsoft [46] code review doubles the number of files that developers know. Furthermore, Thongtanunam *et al.* [54] showed that reviewers of modules are often not authors of the module [54].

In Section 2.4.1, our empirical results show that review naturally reduces turnover risk. We show that when only authors are considered knowledgeable an average of 79% of the total files are at risk. When both authors and reviewers are considered knowledgeable the average *FaR* is 32%. This reduction in *FaR* shows that substantial knowledge is spread 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 -23.92% for *TurnoverRec* and *SofiaWL*, which outperforms both *cHRev* and *WhoDo* which increase *FaR* by 4.15% and 40.97%.

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 [44, 39]. Each strategy requires additional developer effort especially for developers who are expert enough to answer questions and write documentation. In contrast, another 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.

2.6.4 Recommenders

Identifying the right reviewers for a given change is a challenging and critical step in the code review process [13, 2, 19, 3, 56, 60]. Inappropriate selection of reviewers can slow down the review process [56] or lower the quality of inspection [8, 2]. 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 [3, 56, 60, 24, 58, 21, 59, 10].

Advanced recommenders have been proposed which are built upon machine learning [24], text mining [58], social relation graphs [59]. 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 [20].

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 [27]. However, it is reported that the focus on accuracy rarely provides additional value for developers because the recommendations are obvious [27]. Furthermore, in teams with strong code ownership, finding relevant experts is not problematic [51]. For replication completeness we calculated MRR. Our results confirm Kovalenko *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*, *GiniWork*, 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 ΔGiniWork and ΔFaR is 40.58% and 92.58%.

2.7 Conclusion

In this study, we provide empirical results on the concentration of code review workload and the distribution of knowledge across files, *FaR*. We partially replicate existing work on code review recommendation, fully reproduce two state-of-the-art code review recommenders, *cHRev* and *WhoDo* on five open source projects, and propose novel recommenders that are of knowledge distribution. We evaluate the mean reciprocal rank (MRR) for replication purposes, but propose a novel evaluation framework for reviewer recommenders based their impact on *Expertise*, *GiniWork*, and Files at Risk to turnover (*FaR*). The main contributions from our empirical results and simulations are as follows.

Ownership Aware Recommenders concentrate knowledge on a small group of experts. Our findings triangulate Kovalenko *et al.*'s [27] interview finding that traditional review recommenders suggest experts who already known by author of the change. We see this in the increase *Expertise* and high MRR, *e.g., cHRev* 11.11% and 0.52, respectively. We further show that this knowledge concentration increases the risk of turnover with *cHRev* increasing *FaR* by 4.15% and *RetentionRec*, that suggests past reviewers, ΔFaR 65.19%.

We show that code review naturally spreads knowledge and develop recommenders to enhance

this learning effect. We develop three novel **Turnover Aware Recommenders.** The first, *LearnRec*, suggests the develop who knows only one file that is under review. This recommender reduces *GiniWork* but has drastic decreases in *Expertise* and *FaR* by recommending transient developers. *RetentionRec* attempts to solve this problem by recommending developers with high retention potential. While effective for both *Expertise* and *FaR*, it drastically increases core developer *GiniWork*. *TurnoverRec* combines these recommenders, and has the greatest reduction in *FaR* at -29.54% but reduces *Expertise* by -25.30%.

Prior works counted the number of reviews done by developers [47, 26, 1]. However, we are the first to plot the distribution of reviews across developers and to show a strong Pareto distribution, with the top 20% of reviewers accounting for 80.19% of code reviews. Given the skewed data, prior measures such as average workload [1] or thresholds [49, 33] are inappropriate. A major contribution of this work is the use of *GiniWork* to compare the concentration of review workload under the Lorenz curve.

We re-implement and re-evaluate the state-of-the-art **Workload Aware Recommender**, *WhoDo* [1], on open source projects. *WhoDo* is able to balance *Expertise* and reduce *GiniWork* by -19.16%. Unfortunately, it drastically increases *FaR* by 40.97%.

The final outcome of this work is *SofiaWL* that combines the state-of-the-art expert and workload aware recommender, *WhoDo*, with the learning and retention recommender, *TurnoverRec*. This bifunctional 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 with low workload. Through simulation we show that *SofiaWL* is the only recommender that balances the three outcomes simultaneously. This strategy allows us to increase the level of *Expertise* during review by 3.20%, reduces the *GiniWork* by -12.00, and reduces the number of files at risk with a ΔFaR of -23.92%. Recommenders that are aware of only one or two attributes can outperform *SofiaWL* on individual outcome measures, but only *SofiaWL* is aware of and can balance all three simultaneously. Developers and managers can tune the scoring functions to optimize the review recommendations based on their goals.

Chapter 3

Adding a reviewer to spread knowledge

Mirsaeedi and Rigby [33] simulate a recommender by randomly replacing one of the actual reviewers with the suggested reviewer. In this chapter, rather than replacing someone, we mitigate turnover by adding a new reviewer as a learner to the actual reviewers when we have an *Risky File* in the pull request. Since every review will require another reviewer, we consider different recommenders for choosing the additional reviewer. We also try to share knowledge by adding a reviewer when we know that someone will leave the project, *e.g.,* when the leaver gives two-weeks notice. In addition, we change the random replacement strategy to *FaR Replacement* by replacing one reviewer when we have *Risky File* and finally combine these strategies.

3.1 Research Questions

RQ1, Recommenders^{++}: Which recommender suggests the best additional reviewer?

Spreading knowledge during the code review is documented in many studies. They show that during code review, awareness and transparency between a team promote[54, 46, 51, 2]. Mirsaeedi and Rigby [33] randomly replace one of the actual reviewers with the suggested reviewer of the recommender to see the effect of different recommenders on turnover. In this research question, we want to see how well the existing recommenders perform when we add another reviewer when we have *Risky File* in the pull request.

We add a reviewer in 13% of pull requests which are *Risky Pull Request*. The results for different

recommenders are shown in Table 3.1. In addition, Table 3.2 shows the results of FaR^{++} for the *TurnoverRec* recommender over the five projects. Adding a new learner to the actual reviewers causes a substantial decrease in developers' turnover loss. *TurnoverRec* recommender, which tries to find experts who will remain in the project, has the highest decrease in *FaR* in comparison to the other recommenders with -81.17% decrease.

RQ2, Two-weeks' Notice: What is the impact of *FaR* if we know in advance that someone will leave?

In this research question, we want to investigate whether if we know a developer would leave the project, can we reduce the effect of knowledge loss caused by leaving the developer. We assume that if a developer has no contribution after two next weeks, he or she leaves the project with a two-week notice. We consider two weeks' notice because this period is more common between software projects, and we will be close to reality. Hence, we add a new reviewer to the actual reviewers in the pull requests that have a file which has more than two knowledgeable but one of the knowledgeable will leave the projects with two weeks notice and after leaving one developer the file would be *Risky File*. The results which are shown in Table 3.5 shows the reduction of three measures are negligible.

RQ3 *FaR Replacement*: How well does *TurnoverRec* perform when we only replace reviewers on *Risky Pull Requests*?

Previous research [33] randomly replaces one of the actual reviewers with the suggested reviewer. This confounded the normal case of recommendation with the case where there are risky files in a pull request. In this reproduction, we create a baseline for *TurnoverRec* on where we only make recommendations for pull requests with at least one *Risky File*, not all the pull requests. For the *FaR Replacement* recommender, we do not change the real reviewers when we have no *Risky File* in a pull request. In this approach, *FaR* reduction is -25.14% over five projects, which is near the *FaR* of *TurnoverRec* (-29.54%). *Expertise* improves substantially in compression to *TurnoverRec* recommender. The reason for better *Expertise* but less *FaR* reduction is that we replace fewer learners and replace learners in the *Risky Pull Request*. Table 3.3 shows the result over the projects.

RQ4 *AwareFaR*: How well does the combination of *FaR*$^{++}$ when there are abandoned files, and *FaR Replacement* when there are only hoarded files work?

FaR^{++} decreases *FaR* substantially by adding a reviewer in all the *Risky Pull Requests* but it causes 13% more pull requests to review for the additional reviewers. In this research question, we combine *FaR*$^{++}$ and *FaR Replacement* to decrease *FaR* with the minimum additional workload. In this approach to reduce the *FaR* we pay more attention to the abandoned files that nobody knows about them. We divide *Risky Files* into abandoned and hoarded files that one developer knows about them. We add a learner when we have an abandoned file in the pull request and when there is a hoarded file in a pull request replace one of the actual reviewers with a learner who is suggested by *TurnoverRec*. The results show that added pull requests to review decrease in comparison to *FaR*$^{++}$. *FaR* is decreased less than *FaR Replacement* but improves *FaR Replacement* and *TurnoverRec* recommender.

3.2 Background and Definitions

We introduce the background on the way of simulations, and the background about review recommenders and evaluation measures are described in section 2.2. In this section, we want to define a definition that is related to adding reviewers to the pull requests.

Risky File. Based on [33], we consider files that have no knowledgeable developers or that are hoarded by a single developer as a risky file.

$$\textit{Risky Files}(F) = \{\, f \mid \text{Knowledgeable}(f)| \leq 1\} \tag{21}$$

Risky Pull Request. If we have a risky file in a pull request, it is a *Risky Pull Request*. Below is the equation for *Risky Pull Request*.

$$\textit{Risky Pull Request}(R) = \{\, f \mid f \in \text{R}\ , |\text{Knowledgeable}(f)| \leq 1\} \tag{22}$$

***FaR*$^{++}$.** In this recommender we add new learner by *TurnoverRec* when we have *Risky Pull*

Requests to share knowledge among developers and in the other cases we do not do anything.

$$\begin{cases} \textit{TurnoverRec++}(D,R) & \text{if } |Knowledgeable(f)| \leq 1\text{,any } f \mid f \in R \\ \text{Actual Reviewers} & \text{otherwise} \end{cases} \tag{23}$$

GiveNotice. We assume that developers give notice in advance of quitting their job. We consider these developers if their last contribution which includes commits and review is done less than the notice time. If we consider two weeks' notice, it means that developer does not have any contribution after two weeks.

LeaversRiskyFiles. These are files that are not risky now, but if we subtract the NoticedLeaver from the knowledgeable developers, they will be Risky files.

FaR Replacement. In this simulation approach, instead of randomly replacement of actual reviewers in all pull requests, we replace when we have a *Risky File* in pull request to share the knowledge to reduce turnover risk and in the other pull requests that there is not a *Risky File*, we do not change the actual reviewers.

$$\begin{cases} \textit{TurnoverRec}(D,R) & \text{if } |Knowledgeable(f)| \leq 1\text{,any } f \mid f \in R \\ \text{actual reviewers} & \text{otherwise} \end{cases} \tag{24}$$

AwareFaR. In this recommender, we want to balance the *FaR* and *GiniWork*. To gain this goal, we add new learner by *TurnoverRec* just when we have an Abandoned File in a pull request. In pull requests that there is a hoarder file, we replace randomly an actual reviewer by a suggested reviewer of *TurnoverRec*.

$$\begin{cases} \textit{TurnoverRec++}(D,R) & \text{if } |Knowledgeable(f)| = 0\text{,any } f \mid f \in R \\ \textit{TurnoverRec} \text{ (D,R)} & \text{if } |Knowledgeable(f)| = 1\text{,any } f \mid f \in R \\ \text{actual reviewers} & \text{otherwise} \end{cases} \tag{25}$$

3.3 Project Selection and Data

The projects and the criteria over these projects that we simulate recommenders on them is described in section 2.3

3.4 Results

In this section, we discuss the results for our research questions relating to 1) Reproduction: Which recommender suggests the best, an additional *FaR*$^{++}$? 2)Add a new learner when we have two weeks' notice of developer leave, how can affect the evaluation measure? 3)Replace a learner in *Risky Pull Request* how can reduce *FaR*? 4)Combination of *FaR*$^{++}$ and *FaR Replacement* how can affect *FaR* and workload?

3.4.1 RQ1, Recommenders^{++}: Which recommender suggest the best additional reviewer?

In prior work [33] for each pull request, one of the actual reviewers is replaced with the top-recommended developer by the recommender. But we add a new reviewer to all the *Risky Pull Request*. We try different recommenders to know which of them perform better in all measures. Table 3.1 shows the results for different recommenders over five projects. *FaR* reduction is less in *LearnRec* and *AuthorshipRec* among the recommenders by -59.37% and -56.93% respectively. The reason is that they do not find developers who will stay in the project, so when we add a learner, in the future learner leave the project, and we lost the knowledge again. *RevOwnRec* and *cHRev* have the highest *Expertise* but *FaR* decrease is -67.36% and -69.87% which are not good as *TurnoverRec* and *RetentionRec*. *TurnoverRec* has the highest decrease in *FaR* because it considers that developer suitability likelihood. If we compare *RetentionRec* and *TurnoverRec* base on their definitions in *TurnoverRec* we use *LearnRec* to find developers who know less about the pull, so the *Expertise* in *RetentionRec* is 3.46% which is higher than *TurnoverRec* which is 2.88%. Added pulls are the percentage of pulls that we add a new learner to them in comparison to all pull requests. This measure is approximately the same in all the recommenders, and it is about 13%.

Table 3.1: The average change in *Expertise*, *FaR*, Gini and AddedPull Compared to the reviewers who actually performed the review in *FaR*$^{++}$ approach which adds reviewer when there is a *Risky File* in the pull request. AddedPull is a percentage of pull requests which are risky, and we add a reviewer.

Recommender	ΔExpertise	ΔFaR	ΔGini	AddedPull
AuthorshipRec^{++}	2.95%	-56.93%	-0.08%	13.27%
RevOwnRec^{++}	3.71%	-67.36%	1.40%	13.30%
cHRev^{++}	3.29%	-69.87%	2.61%	13.30%
LearnRec^{++}	0.93%	-59.37 %	-0.69%	13.81%
RetentionRec^{++}	3.46%	-79.22%	2.80%	13.25%
TurnoverRec^{++}	2.88%	-81.17%	2.09%	13.24%
WhoDo^{++}	3.00%	-73.21%	-0.23%	13.28%

Table 3.2: Outcome measures in different projects for *FaR*$^{++}$ by using *TurnoverRec* recommender to add a learner in the *Risky Pull Requests*.

Project	ΔExpertise	ΔFaR	Δ Gini	AddedPulls
CoreFX	4.72%	-83.59%	2.26%	14.94%
CoreCLR	2.37 %	-80.05%	2.72%	13.66%
Roslyn	2.53%	-83.70%	0.80%	13.80%
Rust	2.80%	-72.22%	3.26%	14.50%
Kubernetes	1.99%	-86.33%	1.42%	13.95%
Average	2.88%	-81.18%	2.09%	14.17%

When there are risky files, *TurnoverRec* has the highest decrease in *FaR* when we add an additional reviewer.

3.4.2 RQ2, Two-weeks' Notice: What is the impact on *FaR* if we know in advance that someone will leave?

We want to investigate the result of adding a new reviewer when we have a leaver that causes a *Risky File* in a pull request. We assume that leavers give notice before leaving the project and if we have a *GiveNotice* in a pull request add a reviewer to the actual reviewers. The results show that *FaR* decreases not substantially. It is -3.89%, -0.79%, -7.40%, -1.68%, -1.58% for CoreFX, CoreCLR, Roslyn, Rust, Kubernetes respectively with the average of -3.07%. The *Expertise* is near 0 in all projects because we add a new reviewer in a few pull requests(0.89%). The values of *Expertise* are 0.08%, 0.00%, 0.01%, 0.00% and 0.03% respectively. The average over projects is 0.024%. ΔGiniWork is 0.13%, 0.27%, 0.05%, 0.07% and 0.04% respectively with the average of

0.11%. The files that leavers are changed before two weeks of leaving is not substantially and with adding a new reviewer we can not decrease *FaR* substantially. We do not add a new reviewer to the *LeaversRiskyFiles* for the next research questions because of few impacts.

When we consider a two-weeks' notice for leavers, we do not find a decrease in the number of files at risk.

3.4.3 RQ3 *FaR Replacement*: How well does *TurnoverRec* perform when we only replace reviewers on *Risky Pull Requests*?

In the previous work [33] randomly replace one reviewer in each pull request. They change reality in all pull requests to simulate a recommender. We replace an actual reviewer with the suggested reviewer when we have a *Risky Pull Request*, not all pull requests because *Risky Files* cause *FaR* and when we do not have any *Risky File* it is not necessary to change the actual reviewers. In this approach *FaR* decreases in CoreFx, CoreCLR, Roslyn, Rust, Kubernetes by -30.83%, -10.43%, -36.52%, -20.19% and -27.77%. respectively. *TurnoverRec* recommender which randomly replaces one of the actual reviewers with a suggested reviewer in all pull requests. *FaR* results in *TurnoverRec* are -34.95%, -14.20%, -41.70%, -24.32% and -32.53% respectively. The average of *FaR* in *FaR Replacement* and *TurnoverRec* are -25.14% and -29.54% respectively. *TurnoverRec* recommender suggests reviewers who are not experts, so, when we replace less suggested reviewer, the *Expertise* decrease would be less. In *FaR Replacement Expertise* is substantially improves with 0.78%, 1.75%, -2.44%, -0.30% and -1.13% respectively with the average of -1.28% because we replace less learner developers. *Expertise* in *TurnoverRec* decreases -27.41%, -24.91%, -14.05%,-34.22% and -25.93% respectively by the average of -26.55%.

There is not a substantial difference when we only make recommendation on risky pull requests using *TurnoverRec*. This means there is no need to do recommendation when there are no files at risk.

Table 3.3: *FaR Replacement* approach outcome measures over five projects with *TurnoverRec* recommender. Replacement of actual reviewers with a suggested reviewer when there is a *Risky File* in a pull request.

Project	ΔExpertise	ΔFaR	Δ Gini
CoreFX	0.78%	-30.83%	1.79%
CoreCLR	1.75 %	-10.43%	2.87%
Roslyn	-2.44%	-36.52%	7.35%
Rust	-0.30%	-20.19%	2.42%
Kubernetes	-1.13%	-27.77%	1.06%
Average	-1.28%	-25.14%	3.09%

Table 3.4: The outcome measure for *TurnoverRec* recommender with replacement of suggested learner in all the pull request to share the knowledge and there is a negative *Expertise* and decrease of *FaR* because of distribution of knowledge.

Project	ΔExpertise	ΔFaR	Δ Gini
CoreFX	-27.41%	-34.95%	-3.1%
CoreCLR	-24.91 %	-14.20%	-1.67%
Roslyn	-14.05%	-41.70%	-0.2%
Rust	-34.22%	-24.32%	6.75%
Kubernetes	-25.93%	-32.53%	5.34%
Average	-26.55%	-29.54%	1.43%

Table 3.5: The outcome measure in comparison to the reality when we add new reviewer when we have two-week notice leaver which cause a *Risky File* in a pull request.

Project	ΔExpertise	ΔFaR	Δ Gini	AdddedPulls
CoreFX	0.08%	-3.89%	0.13%	1.06%
CoreCLR	0.00 %	-0.79%	0.27%	1.20%
Roslyn	0.01%	-7.40%	0.05%	0.88%
Rust	0.00%	-1.68%	0.77%	0.74%
Kubernetes	0.03%	-1.58%	0.04%	0.61%
Average	0.024%	-3.07%	0.23%	0.89%

3.4.4 RQ4 *AwareFaR*: How well does the combination of *FaR*$^{++}$ when there are abandonded files, and *FaR Replacement* when there are only hoarded files work?

In this approach, we want to balance *FaR* and workload by combining *FaR*$^{++}$ and *FaR Replacement* and when we have an abandoned file we add learner and when there is a hoarder file, we just replace one of the reviewers with the suggested reviewer by the *TurnoverRec*. In this way

we can reduce added pulls to review substantially from 13.24% to 1.86% over five projects and reduce *FaR* for the Corefx, CoreCLR, Roslyn, Rust and Kubernetes by -47.02%, -23.11%, -37.33%, -25.19% and -54.92% respectively. The average of *FaR* over the five projects is -37.51% which is less than *FaR Replacement* (-25.14%) and higher than *FaR*$^{++}$ (-81.59%). *Expertise* is 0.47%, -1.45%, -1.34%, -0.08% and -0.94% respectively and an average of -0.66% over the projects which improves the *Expertise* of *FaR Replacement* because we add new learner not replace when we have abandon file in a pull request.

> Adding reviews only when there is are abandoned files dramatically reduces the number of extra reviews, but does not spread knowledge as well as *FaR*$^{++}$.

Table 3.6: *AwareFaR* outcome measures over five projects. Add learner when we have an abandoned file and replace a reviewer when we have a hoarder file in a pull request.

Project	ΔExpertise	ΔFaR	Δ Gini	AddedPulls
CoreFX	0.47%	-47.02%	1.85%	2.45%
CoreCLR	-1.45 %	-23.11%	0.09%	2.35%
Roslyn	-1.34%	-37.33%	0.80%	1.95%
Rust	-0.08%	-25.19%	2.59%	1.55%
Kubernetes	-0.94%	-54.92%	1.16%	1.03%
Average	-0.66%	-37.51%	1.29%	1.86%

3.5 Threats to Validity

Threats to validity are the same as 2.5 except the random replacement. We remove randomly replacement in the *FaR*$^{++}$ by adding a new reviewer and try to reduce it in the *FaR Replacement* and *AwareFaR*.

Assumption that leavers give notice. In many cases that developers leave projects, they do not give notice before leaving, they remain in the project until the last day. In addition, we consider two weeks' notice, which can be different in companies' policies.

Additional reviewers We assume that we can have additional reviewer when we have a *Risky File* in pull request. It is possible in small projects, we do not have enough developers to add one more reviewer to the pull requests.

3.6 Discussion and Concluding Remarks

We position our findings in the chapter. We discuss how we advance our understanding of code review practice, mitigation of turnover risk through *FaR* by adding a reviewer to the *Risky Pull Requests* and how it can reduce the amount of added workload by combining additional and replacement reviewer.

FaR^{++}. In this approach, we add a new reviewer to the pull requests that have files with less than two knowledgeable we choose these pull requests to add one reviewer to the actual reviewers to share the knowledge of the *Risky File* among the team and reduce the number of files at risk. We use different recommenders for selecting the additional reviewer and *TurnoverRec* has better performance because we want to share the knowledge with developers who will remain in the project but do not know about the files. With this approach *FaR* decreases -81.71% which is the best result among recommenders. However, the *Expertise* is not as good as recommenders that choose experts. We added reviewers in 13.24% of pull requests, and it is the workload over the team that we should consider when we want to use this approach.

Two weeks' notice. We try to reduce *FaR* by adding a reviewer when we have a leaver developer, with the assumption that give a notice two weeks in advance. We consider two weeks because if many projects there is a rule that leavers must give two weeks' notice. In this approach, we consider files that are not *Risky File* in the pull requests but after leaving one of the knowledgeable it would be *Risky File* and in these cases we add new reviewer by *TurnoverRec*. The results show that there would be a few reductions in the *FaR* because we just add new reviewer in the 0.89% of pull requests. Files that are changed by leavers in comparison to the *Risky Files* which we consider in the *FaR* is not substantial. So the reduction of *FaR* is -3.07% over the projects.

FaR Replacement. We change the Mirsaeedi and Rigby approaches. They randomly replace one of the actual reviewers with one of suggested developers by recommenders. We want to change the reality less that *TurnoverRec* simulation recommender by replacing a suggested reviewer in *Risky Pull Requests* because when we do not have *Risky File* in the pull request it does not necessary to change the reality. With this approach we could increase *Expertise* because we replace less learner. However, *FaR* reduction is -25.14% in comparison to the *TurnoverRec* which is -29.54% is

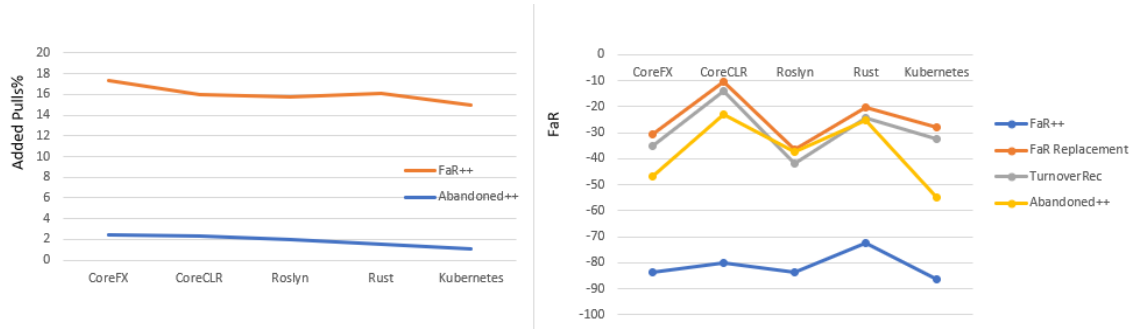

Figure 3.1: Left graph shows *FaR* in different approaches over five projects and the right one shows the amount of added work

acceptable because we change 13.24% of pull requests.

AwareFaR. In this approach, we divided *Risky Files* to abandoned files that do not have any knowledgeable and hoarded file with one knowledgeable and combine *FaR Replacement* and FaR^{++} approaches. We consider that abandoned files are more risky and we have to add a reviewer when we have these files in the pull requests. If we have hoarder files that one developer known about them, we can replace one of the actual developers with the suggested developer. In this approach, we could balance workload and *FaR*. We improve *TurnoverRec* recommender *FaR* by adding new reviewer in 1.86% of pull requests. Figure 2.2 shows that the *FaR* reduction in FaR^{++} is substantially higher than other approaches, while it shows that the added pull requests are high. Hence, reduction of *FaR* and added workload is a trade-off, and it depends on the goal of projects that which of them is more important for us.

Chapter 4

Conclusion

In the first part of this thesis:

(1) We introduce the ΔGiniWork measure which is the area under the Lorenz curve that shows the distribution of workload of reviewers and re-evaluate existing recommenders by the *FaR*, *Expertise*, ΔGiniWork measures.

(2) We replicate *WhoDo* recommender which is used in Microsoft and evaluate by it by our three outcome measures.

(3) We introduce a novel recommender, *SofiaWL*, which combines workload and expert aware recommenders to balance the *FaR*, *Expertise* and *GiniWork*. This recommender improves the *Expertise* by 3.20%, reduces the *GiniWork* by -12.00% and decreases files at risk by -23.92%. *SofiaWL* can balance all the measures, while other recommenders have good results on a specific measure.

In the second part of this thesis:

(1) We change the simulation approach and introduce *FaR*$^{++}$, which adds a new reviewer as a learner when there is a *Risky Files* in the pull request. In this approach, *FaR* decreases substantially. With *TurnoverRec*, we share knowledge through additional reviewers and remove *Risky Files*. By adding a suggested reviewer by *TurnoverRec* recommender to the *Risky Pull Requests*, *FaR* decrease by -81.17%. This approach increases overall workload because we

need one more reviewer for *Risky Pull Requests* but ΔGiniWork does not change substantially because the distribution of new reviews is the same.

(2) We introduce another measure, AddedPull, which shows the percentage of pull requests that we add additional reviewers to. In the *FaR*$^{++}$ approach we add reviewers on 13% of the pull requests.

(3) We also randomly replace one of the actual reviewers with a suggested reviewer by *Turnover-Rec* recommender when we have *Risky Pull Requests* instead of all pull requests. The reason is that when we do not have *Risky File* in the pull request it is not necessary to spread knowledge. In this simulation approach, which is named *FaR Replacement*, we change the reality less than Mirsaeedi and Rigby's approach. *FaR Replacement* decrease *FaR*-25.14% which is acceptable in comparison to the *TurnoverRec* recommender (-29.54%). This less reduction of *FaR* is because we replace less learner in comparison to *TurnoverRec* recommender.

(4) We combine these approaches. It means that we divide *Risky Files* into abandoned and hoarded files. Abandoned files are files that no one knows about them. They are very important to consider, so we add a new reviewer to the pull requests that have abandoned files. However, when we have a hoarded file, it is better to replace one of the actual reviewers with the expert people in other areas who will probably remain in the project. In this approach, we combine *FaR*$^{++}$ and, *FaR Replacement* which is named *AwareFaR* and reduces *FaR*-37.51% which is better than *TurnoverRec* but it is less than *FaR*$^{++}$. The point of this approach is that we add a reviewer to 1.86% of the pull request, and it is a substantial reduction in comparison to *FaR*$^{++}$ which adds a reviewer in 13% of pull requests.

Bibliography

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

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

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

[11] 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.

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

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

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] F. Hajari, E. Mirsaeedi, and P. C. Rigby. Replication package and relationalgit. https://github.com/fahimeh1368/RelationalGit/tree/master/ReplicationPackage, 2021.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] O. Kononenko, O. Baysal, and M. W. Godfrey. Code review quality: how developers see it. In *Proceedings of the 38th International Conference on Software Engineering*, pages 1028–1038, 2016.

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

[28] A. Krutauz, T. Dey, P. C. Rigby, and A. Mockus. Do code review measures explain the incidence of post-release defects? *Empirical Software Engineering*, 25(5):3323–3356, 2020.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] E. Mirsaeedi and P. C. Rigby. Mitigating turnover with code review recommendation: Balancing expertise, workload, and knowledge distribution. In *Proceedings of the ACM/IEEE 42nd*

International Conference on Software Engineering, ICSE '20, page 1183–1195, New York, NY, USA, 2020. Association for Computing Machinery.

[34] 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.

[35] 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.

[36] A. Mockus, R. T. Fielding, and J. Herbsleb. A case study of open source software development: The apache server. In *Proceedings of the 22nd International Conference on Software Engineering*, ICSE '00, page 263–272, New York, NY, USA, 2000. Association for Computing Machinery.

[37] 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.

[38] 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.

[39] 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.

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

[41] 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.

[42] 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.

[43] 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.

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

[45] 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.

[46] 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.

[47] 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.

[48] 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.

[49] 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.

[50] 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.

[51] 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.

[52] 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.

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

[54] 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.

[55] 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.

[56] 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.

[57] 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.

[58] 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.

[59] 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.

[60] 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.

[61] M. Zhou and A. Mockus. Developer fluency: Achieving true mastery in software projects. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, page 137–146, New York, NY, USA, 2010. Association for Computing Machinery.