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

Ehsan Mirsaeedi
Department of Computer Science and
Software Engineering
Concordia University, Montréal, Québec, Canada
s_irsaee@encs.concordia.ca

Peter C. Rigby
Department of Computer Science and
Software Engineering
Concordia University, Montréal, Québec, Canada
peter.rigby@concordia.ca

ABSTRACT

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

KEYWORDS

Turnover, Knowledge Distribution, Code Review, Recommenders, Tool Support

ACM Reference Format:
Ehsan Mirsaeedi and Peter C. Rigby. 2020. Mitigating Turnover with Code Review Recommendation: Balancing Expertise, Workload, and Knowledge Distribution. In *42nd International Conference on Software Engineering (ICSE '20), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3377811.3380335

1 INTRODUCTION

Turnover on software projects is frequent and inevitable and leads to the loss of knowledge when developers leave a project [3, 45]. Turnover incurs substantial economic cost in recruiting and training new employees [32, 36], it reduces the productivity of development teams [20, 32], it leads to the loss of critical tacit knowledge [19, 31, 32], and has been shown to increase the number of defects in a product and reduce overall product quality [12, 31, 32].

Recent works have tried to mitigate the adverse impact of turnover through increasing knowledge retention by predicting leavers [3, 9, 24], planning for succession [31, 35, 45, 49], documenting knowledge, and persisting knowledge on StackOverflow and other internal QA forums [35, 40]. 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 [40].

In this work, we show that code review can mitigate turnover risk because it naturally distributes knowledge by exposing developers to new code during reviews. Prior work interviewed developers and showed that code review is an opportunity for learning and it plays a vital role in distributing knowledge [1, 6, 17, 42, 47, 50]. Furthermore, studies have quantified the knowledge gained during code review [41, 47] and shown that developers review code in modules they have not modified [50]. 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 [2, 18, 21, 39, 52, 54–56] are solely focused on finding expert reviewers and disregard the role of code review in distributing knowledge among developers. These recommenders result in expertise concentration because the evaluation benchmark is how many of the actual developers who performed the review were recommended. Interviewed developers state that these recommenders suggest obvious candidates and do not provide additional value [23].

To evaluate recommenders from other perspectives, we introduce three outcome measures that interviews with developers indicated as important aspects of code review [1, 17]: *Expertise*, *CoreWorkload*, and *FaR*. The first outcome ensures that expertise remains high for finding defects during review. The second, ensures that the core developers are not unreasonably overworked due to always being the top recommendation. The third outcome measures the number of files that are at risk to turnover, *FaR*, to ensure that knowledge is adequately distributed during review. We run simulations on the historical reviews of five large projects to understand how recommenders affect each outcome. For completeness, we also calculate Mean Reciprocal Rank, MRR, to understand how well each recommender predicts the developers who actually performed the review.

Research Questions

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

Recent studies have quantified knowledge loss from turnover on the basis of the commits that each developer has authored [34, 45]. 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 [1, 42, 47, 50]. 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% of the total files are at risk to turnover. When we consider both authors and reviewers to be knowledgeable *FaR* drops to 32%. Code review naturally distributes knowledge.

RQ2, Ownership: Does recommending reviewers based on code ownership reduce the number of files at risk to turnover?

Studies show that teams tend to assign reviews to the owners of files under review [17, 47] and experts who have modified or reviewed the files in the past [2, 23]. We implement simple ownership recommenders that suggest reviewers based on the files that developers have modified or reviewed in the past.

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

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

We review the literature on review recommenders and find that most mine historical review information. Unfortunately, we did not find working implementations or replication packages for any of the existing recommenders. For comparison purposes, we re-implement cHRev which has been shown to outperform other recommenders [56]. When re-evaluate *cHRev* on our outcome measures, we find that like the ownership recommenders, cHRev increases the level of expertise by 11.11%, and has the added benefit of reducing *CoreWorkload* by -3.49%. Unfortunately, cHRev concentrates knowledge and increases the risk of knowledge loss through turnover by 4.15%.

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

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

RQ5, *Sofia*: Can we combine recommenders to balance *Expertise*, *CoreWorkload*, and *FaR*? Each of the previous recommenders has a focus and cannot simultaneously balance the outcomes. Our final recommender, *Sofia*, assigns either experts or learners based on the files under review. It uses cHRev when the files under review are not at risk and uses *TurnoverRec* when few developers know about the files under review. This multi focus strategy improves all outcomes simultaneously. *Sofia* increases the level of expertise during review by 6.27%, while having a minor impact of 0.09% on *CoreWorkload* and reduces turnover risk with a ΔFaR of -28.27%.

We integrated *Sofia* to make recommendations for GitHub pull requests and to recommend both expert and "learning" developers. The *Sofia* source code [30] is publicly available along with the data in a replication package [29]

This paper is organized as follows. In Section 2, we provide the study background as well as defining our measures, review recommender, scoring functions, and simulation methodology. In Section 3, we describe the projects under study. In Section 4, we present results for each of our research questions. In Section 5, we describe the *Sofia* bot which integrates into GitHub pull request. In Section 6, we discuss threats to validity. In Sections 7 and 8, we discuss our findings in the context of the existing literature and conclude the paper.

2 BACKGROUND AND DEFINITIONS

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

2.1 The Ownership Recommenders

The influence of code ownership on code quality has been extensively investigated in the literature[5, 12, 38, 50]. Ownership is a human factor that helps with finding knowledgeable developers that can be accountable for a particular part of code or task [33]. *Developer Recommenders* use ownership to automatically assign tasks to experts [22]. Researchers have used a wide range of granularity, from lines [13, 14, 38] to modules [5], to estimate ownership of developers. Studies on code review find that code owners are usually selected to review changes [1, 17, 47]. In this work we develop two simple scoring functions for review recommendation based on ownership.

AuthorshipRec. Bird *et al.* [5] defines the code ownership for a developer in a module as the ratio of commits the developer has made relative to the total commits made to that component. Our *AuthorshipRec* scores a developer, *D*, as a candidate reviewer based on the number of commits he or she has made to the files under review, R, divided by the total number of commits made to these files.

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

RevOwnRec. Thongtanunam *et al.* [50] 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.

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

2.2 The *cHRev* Recommender

There is a large literature on review recommendation [2, 18, 39, 52, 54–56]. We note that we did not find a replication package or recommender implementation for any of these works [25]. We only re-implement cHRev [56] because it includes a wide range of factors in its recommendation and has a higher accuracy than the other review recommenders such as RevFinder [52].

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.

2.3 The Turnover Mitigating Recommenders

The focus of existing recommenders on experts disregards the other benefits of code review such as knowledge sharing. Rigby and Bird [42] report that code review increases the number of files developers see by between 100% and 150%. In this work, we speculate that code review can be effective in mitigating the turnover-induced knowledge loss. Based upon this idea, we design reviewer recommenders that either distribute or retain knowledge.

2.3.1 *Distributing Knowledge.*

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

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

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

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

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

2.3.2 *Developer Retention.*

Developers who have made substantial recent contributions to a project have demonstrated a high degree of commitment to the project[9, 48]. In contrast, assigning a review to a developer who is transient and will likely leave the project is antithetical to the goal of retaining project knowledge. We define commitment and contribution consistency measures to recommend reviewers with a high potential of remaining on the project, *i.e.* high retention potential. In contrast to the previous measures which are at the pull request or review level, the retention is done at a project-wide level.

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

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

ConsistencyRatio. It is common for developers to make substantial contributions to a feature and leave the project after the feature is complete. To avoid assigning reviews to transient developers, we define the 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 who are unlikely to leave the project. The scoring function for a candidate review, D is

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

2.3.3 *Distribution and Retention Combined.*

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

$$TurnoverRec(D,R) = LearnRec(D,R) * RetentionRec(D) \quad (9)$$

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

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

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

2.4 Simulation and Evaluation

To evaluate reviewer recommenders, prior works made recommendations for each exiting review and compared their result against the actual reviewers who performed the review [2, 18, 39, 52, 54–56]. To compare with the actual reviewers, we use the Mean Reciprocal Rank (MRR) and evaluate each recommender. MRR is the average of the inverse rank of the highest ranked correct recommendation. For example, if a correct recommendation is on average the third recommendation, the score would be 1/3.

A criticism of prior works can be found in Kovalenco *et al.*'s [23] interviews with developers who state that the recommenders rarely provide additional value because they suggest obvious expert candidate reviewers. This problem is inherent in the outcome measure, which assumes that the actual reviewers were the best, *i.e.* "correct" reviewers. Kovalenco *et al.* [23] 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 formulas defined in Sections 2.1, 2.2, and 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 disrupting the peer review process and because Kovalenco *et al.* [23] showed that developers usually already know at least one expert review candidate.

To evaluate how each recommender changes the project, we measure three outcomes: the degree of reviewer *Expertise*, reviewer *CoreWorkload*, and the number of files at risk to turnover, *FaR*. These measures incorporate the reasons interviewed developers conduct review [1, 17]. We measure the change in the outcomes over the standard quarterly period [34, 45]. Each measure is calculated as a percentage change relative to the actual reviewers who performed the review. For example, if a recommender replaces an expert reviewer with a non-expert "learner," we would expect the measures to report a percentage decrease in expertise and a percentage increase in the knowledge distribution of the development team. We define each outcome measure below.

Expertise. Having high expertise ensures having high quality code review [1, 7, 11]. We measure the *Expertise* for a review as the proportion of files under review that the selected reviewers have modified or reviewed in the past, *i.e.* the union of the files that the reviewers know about. We sum the expertise across the reviews per quarter, Q.

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

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

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

FaR. We need to quantify the project's exposure to turnover from knowledge loss. Building on Rigby *et al.*'s [45] definition of knowledge loss we define the quarterly Files at Risk, *FaR*, as the number of files that are known by zero or one active developers. Given the function ActiveDevs(Q,f) that returns the set of developers who have modified or reviewed the file, f, and have not left the project at the end of the quarter, Q, we define *FaR*(Q):

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

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

Since percentage change is a trivial formula, we illustrate it only for ΔCoreWorkload:

$$\Delta\text{CoreWorkload}(Q) = \left(\frac{\text{Simulated}CoreWorkload(Q)}{\text{Actual}CoreWorkload(Q)} - 1\right) * 100 \quad (14)$$

The simulation results for an *ideal reviewer recommender* increases *Expertise* during review with a positive percentage change in ΔExpertise, reduces *CoreWorkload* with a negative percentage

Table 1: Size of projects under study. We explicitly select for large, long-lived projects.

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

change in ΔCoreWorkload, and reduces the number of files at risk, *FaR*, with a negative percentage change in ΔFaR.

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 reviews is small and the developers are often aware of the entire team. To select projects, we first query the GitHub torrent dataset to find projects with more than 10K pull requests [15, 29]. We then apply the following manual selection criteria:

(1) We need existing reviews, so 25% or more of the commits must be reviewed.
(2) We need to simulate across time, so the project must be 4 or more years old.
(3) We need diverse knowledge and modules, so we ensure there are at least 10K files.

Five projects met our selection criteria. Of these projects, CoreFX, CoreCLR, and Roslyn are led by industry but are available under an open source license and are developed in the open on GitHub. Rust and Kubernetes are community driven OSS projects. Table 1, provides the summary statistics including the number of files, pull requests, and commits. Our replication package contains a link to the project data [29].

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 [16]. 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 [29].

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 [4] and Canfora *et al.*'s [8].

Leavers. Robillard *et al.* [46] shows that using the last commit as an indicator for departure of developers draws some risks. Based on this finding, at the end of each quarter, we consider the knowledge of a developer to be inaccessible if he or she has no contribution in the subsequent four quarters. We exclude the last quarter of projects from analysis to ensure that we do not mistakenly label a developer as a leaver if they have gone on vacation for a month more.

Excluding mega commits. Rigby *et al.* [45] argue that commits with hundreds of file changes are too large to be fully comprehended by the author. In manual analysis of mega commits and review requests, we find that they tend to be superficial changes including renaming a folder, renaming a function throughout the source code, changing commented trademarks of files, or importing a large chunk of code from a different source control system to git. We do not associate any knowledge to the author or reviewer of changes with 100 or more files.

In this work, we limit our study of knowledge to code files, including *.cs*, *.java*, and *.scala*. Our replication package contains the full list of file types [29]. 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.

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

4.1 RQ1: Review and Turnover

What is the reduction in files at risk to turnover when both authors and reviewers are considered knowledgeable? Recent studies have quantified knowledge loss from turnover on the basis of the commits that each developer has made [34, 45]. The assumption in these works, is that knowledge is only attained through writing code. However, the knowledge transfer that occurs during code review is widely documented with prior work showing that review promotes team awareness, transparency, and shared code ownership [1, 42, 47, 50]. Rigby and Bird [42] quantified the additional knowledge attained during review and reported that code review exposes developers to between 100% and 150% more files than they edit. Thongtanunam *et al.* [50] added that developers who have not made any changes to a module contributed by reviewing 21% to 39% of the code changes in the module. In this section, we consider both authors of code as well as reviewers to be knowledgeable and calculate the number of files that are at risk when turnover occurs.

To assess the extent that the project is at risk to knowledge loss from turnover, we measure *FaR*, see Equation 13, which measures the number of files that have zero or one active developers at the end

Table 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%

of each quarter. To mirror prior works, we calculate the FaR_{author} which only considers authors to be knowledgeable [34, 45]. We then calculate *FaR*, which considers both authors and reviewers as knowledgeable.

Table 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 [34, 45].

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.

4.2 RQ2 Ownership

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

Studies show that teams tend to assign reviews to the owners of files under review [17, 47] and experts who have modified or reviewed the files in the past [2, 23]. 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 [22, 26, 33] that use historical authorship data to recommend reviewers [18, 56]. We partially reproduce these authorship recommendations by using the scoring function defined in Equation 1. We use the simulation method described in Section 2.4 and evaluate the impact of *AuthorshipRec* on MRR, ΔExpertise, ΔCoreWorkload, and ΔFaR. The average values are shown in Table 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 *CoreWorkload* increases for Rust by 7.50%, while it is reduced by -11.30%, -4.74%, -6.91%, and -2.95% for the other projects, with an average of -3.68%. Although *Expertise* is high for each review, *FaR* has risen across all projects by 28.05%, 12.00%, 36.23%, 33.51%, and 14.48%, with an average of 25.25%.

Developers who have authored the files under review are clearly experts. However, suggesting past authors as reviewers concentrates the knowledge of these files and puts the project at greater risk to turnover as non-authors are not suggested as reviewers.

RevOwnRec. The majority of review recommenders have used historical review data, *i.e.* who has reviewed which files or modules in the past, to recommend reviewers [2, 21, 52, 54, 55]. We partially reproduce these review ownership results by using the scoring function defined in Equation 2. We use the simulation methodology and outcome measures as described above.

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

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

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

4.3 RQ3 *cHRev* Recommender

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

cHRev builds upon prior work that leverages information in past reviews [22], 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 for further details). *cHRev* has been show to outperform the other review-based recommenders, including RevFinder [50]. 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 2.4 and evaluate the impact of *cHRev* on MRR, ΔExpertise, ΔCoreWorkload, and ΔFaR.

In the original cHRev paper, the authors report an average MRR of .67 across four projects [56]. On our projects, cHRev has an MRR of 0.64, 0.59, 0.49, 0.50, and 0.42, for CoreFX, CoreCLR, Roslyn, Rust, and Kubernetes, respectively. The average is 0.52. This implies that on average the actual reviewer is ranked 1.92. Although the MRR is lower in our reproduction than in the original study, we note that for MRR cHRev outperforms all of the other recommenders we consider.

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

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

4.4 RQ4: Learning and Retention

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

The previous research questions have demonstrated that existing review recommenders concentrate knowledge on experts increasing the risk of knowledge loss from turnover. Furthermore, in two large industrial settings, Kovalenco *et al.* [23] interviewed developers and found that suggesting prior review experts tends to recommend reviewers that are obvious to the author of the change. They state that making obvious recommendations leads to a lack of use of recommenders. They envision a new research path for next generation of recommenders that go beyond suggesting experts. In this research question, we investigate how we can mitigate turnover-induced loss and disseminate knowledge using learning and retention measures.

Table 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 ΔCoreWorkload), and reduces files at risk to turnover (negative ΔFaR).

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

LearnRec. Without review recommenders, development teams naturally distribute knowledge during review by assigning reviewers who would benefit by learning about the files under review [1, 6, 47]. Building on this idea, in Section 2.3.1, we defined a scoring function that determines how many files a candidate reviewer will learn about. We ensure that the candidate knows at least one of the files that is under review. In this way, we spread knowledge, but ensure that the reviewer has some relevant knowledge. We use the simulation method described in Section 2.4 and evaluate the impact of *LearnRec* on MRR, ΔExpertise, ΔCoreWorkload, and ΔFaR with the average outcomes shown in Table 3.

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

From the simulations, we see a substantial decrease in *Expertise*: -34.91%, -32.76%, -24.35%, -50.34%, and -33.33%, respectively, with an average of -35.13% across all projects. The *CoreWorkload* is drastically reduced as fewer expert reviewers are assigned reviews: -38.07%, -38.53%, -35.68%, -49.86%, and -35.45%, with an average of -39.51%. The goal of this measure is to distribute knowledge and reduce turnover. Counter-intuitively we see an increase in the files at risk with ΔFaR values of 16.26%, 22.31%, 119.32%, 108.72%, 48.61% with an average of 63.04%. By selecting non-experts, *LearnRec* recommends transient developers who have less commitment to the project.

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

RetentionRec. Assigning reviews to transient developers may distribute knowledge, but does not reduce turnover. In Section 2.3.2, we define a measure that captures how frequently developers contribute to the project and the number of months in the last year that they are active. We ensure that the candidate knows at least one of the files that is under review. Our goal is to assign reviews to committed developers. We use the same simulation methodology and outcome measures.

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

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

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

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

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

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

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

4.5 RQ5 *Sofia*

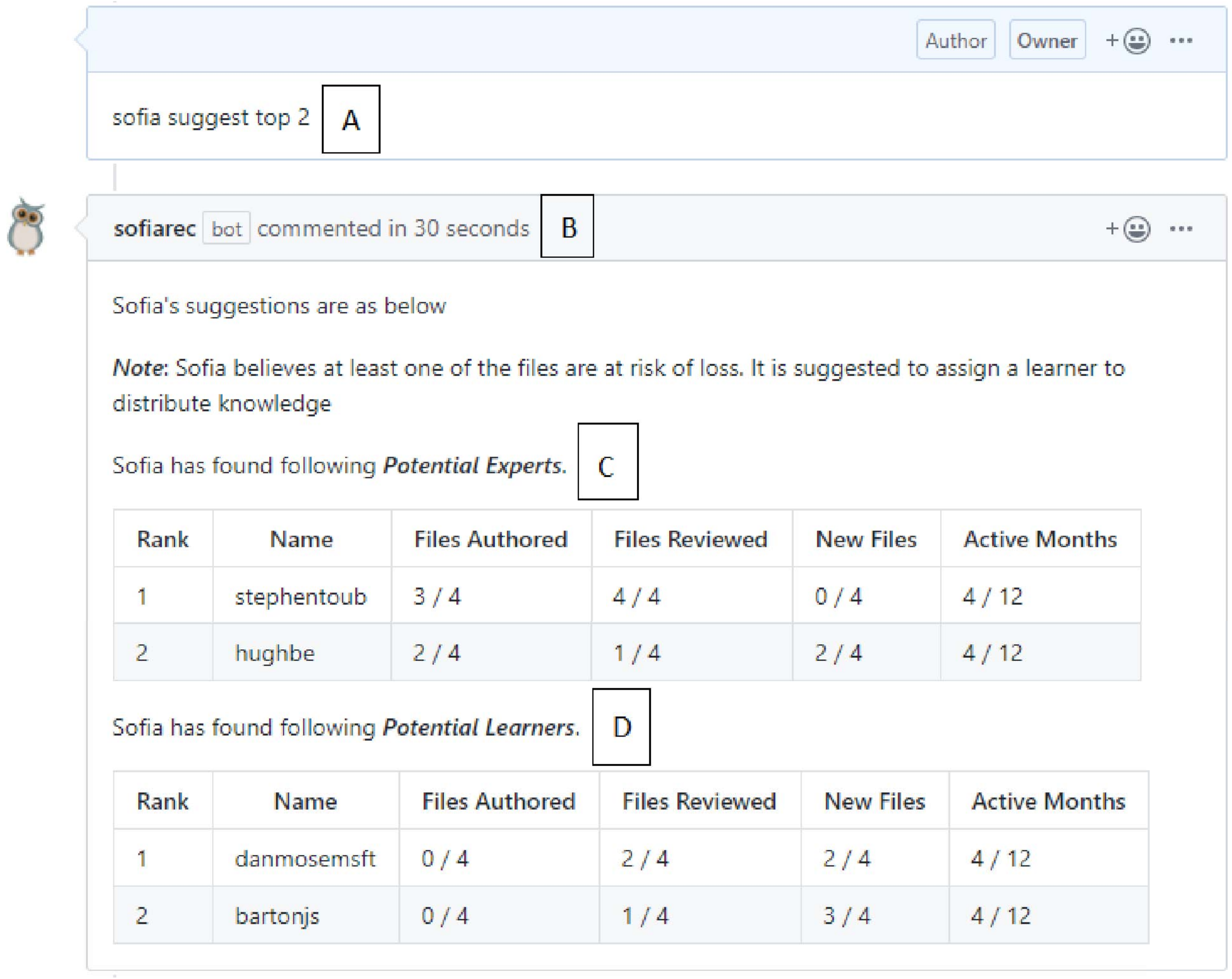
Can we combine recommenders to balance *Expertise*, *CoreWorkload*, and *FaR*?

Not all reviews contain files that are at risk of abandonment. As a result, we do not need to distribute knowledge on these files because there is already a sufficient number of developers to mitigate knowledge loss from developer turnover. In Equation 10, we define *Sofia* that distributes knowledge during review using *TurnoverRec* when there are files at risk of abandonment. In contrast, when all the files have active developers, *Sofia* uses the *cHRev* scoring function to suggest recent experts. Of the 13,690, 10,256, 10,388, 17,810, and 32,260 reviewed pull request on CoreFX, CoreCLR, Roslyn, Rust, and Kubernetes around 1/4, 25.18%, 26.13%, 29.82%, 29.41%, and 17.17%, contain files at risk. The remaining pull requests use *cHRev* recommendations to ensure concentrated expertise. We use the simulation method described in Section 2.4 and evaluate the impact of *Sofia* on MRR, ΔExpertise, ΔCoreWorkload, and ΔFaR with average outcomes shown in Table 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 *CoreWorkload*, we see a reduction of -0.27% and -5.89% in CoreFX and CoreCLR, an increase in Roslyn of 5.09% and a slight increase of 0.43% and 1.12% for Rust and Kubernetes. The average of ΔCoreWorkload is minor at 0.09%. *Sofia* distributes knowledge to developers who have a high retention potential and reduces the risk of turnover as measured by a decrease in ΔFaR of -34.46%, 12.42%, -41.56%, -19.92%, and -33.02%, with an average of -28.27%.

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



sofia suggest top 2

sofiarec bot commented in 30 seconds

Sofia's suggestions are as below

Note: Sofia believes at least one of the files are at risk of loss. It is suggested to assign a learner to distribute knowledge

Sofia has found following ***Potential Experts***.

Rank	Name	Files Authored	Files Reviewed	New Files	Active Months
1	stephentoub	3 / 4	4 / 4	0 / 4	4 / 12
2	hughbe	2 / 4	1 / 4	2 / 4	4 / 12

Sofia has found following ***Potential Learners***.

Rank	Name	Files Authored	Files Reviewed	New Files	Active Months
1	danmosemsft	0 / 4	2 / 4	2 / 4	4 / 12
2	bartonjs	0 / 4	1 / 4	3 / 4	4 / 12

Figure 1: An example of *Sofia* recommending both learners and experts for the CoreFX project.

5 THE *SOFIA* BOT ON GITHUB

Code review is known to have multiple purpose and outcomes from finding defects to distributing knowledge [1, 6, 17, 42, 47]. Our tool design allows developers to make an informed selection balancing the need for experts and learners. We created a GitHub application [30] that will recommend reviewers based on the combination of *cHRev* with *TurnoverRec* as the *Sofia* bot. Feedback from developers showed that the rationale behind a review recommendation is required [23]. For the *Sofia* bot we display simplified measures to complement a developer's intuition and domain expertise on who should review the pull request.

Implementation. The *Sofia* repository with the source code and the straightforward installation instructions are publicly available [28]. Once installed *Sofia* processes the entire history of the project to be able to recommend reviewers. *Sofia* uses GitHub webhooks to scan submitted commits and reviewed pull requests to keep recommendations up-to-date. *Sofia* can operate in two modes: fully automated or list candidates. In the fully automated mode, for each pull request, *Sofia* assigns the top scoring candidate to perform the review.

In Figure 1 *Sofia* displays a list of candidates when the pull request is created or when the `Sofia suggest` command is issued (Box A in figure). The *Sofia* bot displays the ranked list of potential reviewers (Box B). In Box C in the figure, the author can select the person with the highest expertise. Or if learning is more important they can select the developer who would learn about the most files (Box D). The author can also issue the `Sofia suggest learners` or `Sofia suggest experts` if he or she is only interested in a particular type of candidate.

To help with tool adoption, the displayed measures are designed to be quick and easy to interpret by pull request authors and are major simplifications of the scoring functions defined in Section 2.3. The ownership dimension maps to the "Files Authored" and the "Files Reviewed" fields which are simplified to show the proportion of files under review that the candidate has authored or reviewed in the past, respectively. Learning maps to the "New Files" field which is simplified to the number of files that the candidate would learn about, *i.e.* they have not modified or reviewed. Retention potential maps to the “Active Months" field which is simplified to the proportion of months that the developer has been active in the previous year.

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

6 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 [52, 56], ownership [13, 17, 42], and turnover [34, 45], we use the source code file as the unit of knowledge. Knowledge is contained in other documents and at other unit levels. We leave these investigations to future work. We have also provided formulas for each of our measures and scoring functions to facilitate replication.

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

Randomly 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 2.4). Table 1 showed that we examine over 80k reviews across 5 projects, making it unlikely that this random selection will lead to systematic bias. As a further check, over a period of four months, we re-ran our top two techniques, *cHRev* and *Sofia*, a minimum of 215 times for each project. For *cHRev*, we see a change of -0.04, -0.75, -0.52, and -0.86 percentage points for MRR, ΔExpertise, ΔCoreWorkload, and ΔFaR, respectively. The corresponding values for *Sofia*, are 0.00, -0.10, 2.24, and 1.21 percentage points, respectively. The results remain consistent with *Sofia* increasing *Expertise* with a minor increase in *CoreWorkload*, while drastically decreasing *FaR*.

***CoreWorkload* Threshold** We define *CoreWorkload* to measure the reviews performed by the top 10 reviewers on a project (see Equation 12). While future work could use proportion based core teams, we used this value because it simplified our functions and represented a reasonably consistent percentage of reviews across the studied projects: 58%, 53%, 61%, 52%, and 37% for CoreFX, CoreCLR, Roslyn, Rust, and Kubernetes, respectively.

Replication and Reproducability. Existing recommenders including ReviewBot [2], RevFinder [50], and cHRev [56] do not provide a replication package or source code for their recommenders. As a result, we re-implemented cHRev for comparison because it outperform other state-of-the-art recommenders. We also implemented simple authorship and ownership recommenders. Comparing each recommender with existing baseline recommenders reduces the threat of internal validity. We make all of our code, data, and GitHub *Sofia* bot available for future researchers as well as for use on software projects [29].

7 DISCUSSION AND LITERATURE

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

7.1 Understanding Code Review Practice

Fagan [11] introduced software inspections in 1976 with a detailed experiment that conclusively showed that inspection found defects earlier in the design process and that unreviewed design artifacts lead to defects that slipped through to latter stages increased overall effort. In the subsequent 40 years, code review has been extensively studied. Early works focused on examining the process [10, 11]. However, Porter *et al.* [37] demonstrated that process was much less of a factor than ensuring expertise during review. Current code review practice favors a lightweight process that focuses on expert discussion of changes to the system [1, 6, 7, 42, 44] that still improves software quality [27, 43]. We show that *RetentionRec* has the highest ΔExpertise among all expert recommenders with an average of 16.59%. *RevOwnRec* and *AuthorshipRec* that focus on ownership have an average of 15.17% and 11.29%, respectively. We also found that focusing on learners will reduce *Expertise* by up to -26.55%.

Recent works that interview reviewers, find that experts tend to be overloaded with their review workloads [17, 47] and that it is often difficult to find an available expert reviewer [17, 44, 52]. Moreover, it has shown that high overall workload could lead to poor review participation[51] and requesting feedback from experts can lead to delays from lack of availability and also fewer opportunities for knowledge dissemination [17]. We show that the relationship between *Expertise* and *CoreWorkload* is not straightforward. For instance, *cHRev* and *AuthorshipRec* improve the *Expertise* while at the same time reduce the *CoreWorkload* by -3.49% and -3.68% on average, respectively. On the other hand, *TurnoverRec* drastically reduces *Expertise* by -26.55% while increases the *CoreWorkload* by

1.07% and *Sofia* improves the *Expertise* with a negligible change of 0.09% in *CoreWorkload*.

7.2 Turnover-Induced Knowledge Loss and Mitigation

Turnover deprives the project of the leaver's experience and knowledge [19, 53] and has been shown to increase the number of defects [32]. 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 [34, 45]. 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[1, 6, 17, 42, 47]. Two separate studies quantified the knowledge gained during code review and showed that at both Google [47] and Microsoft [42] code review doubles the number of files that developers know. Furthermore, Thongtanunam *et al.* [50] showed that reviewers of modules are often not authors of the module [50]. In Section 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 attained during code review.

In this work, we design recommenders that explicitly distribute knowledge by suggesting reviewers who would learn about the files under review. We show that by distributing knowledge among developers who have a higher retention potential, there is a *FaR* reduction of -29.54% and -28.27% for *TurnoverRec* and *Sofia*, which outperforms cHRev which increases *FaR* by 4.15%. The advantage of using code review in mitigating knowledge loss is that it adds little additional effort because code review is already a common practice on software teams. In contrast, prior works on turnover mitigation suggest increasing documentation with blogs, formalizing the process of documenting bugs in issue trackers, and participating in StackOverflow and internal QA forums [35, 40]. Each strategy requires additional developer effort especially for developers who are expert enough to answer questions and write documentation.

7.3 Recommenders

Identifying the right reviewers for a given change is a challenging and critical step in the code review process [1, 2, 11, 17, 52, 56]. Inappropriate selection of reviewers can slow down the review process [52] or lower the quality of inspection [1, 7]. 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 [2, 18, 21, 52, 54–56].

Advanced recommenders have been proposed which are built upon machine learning [21], text mining [54], and social relation graphs [55]. 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 [29].

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[23]. However, it is reported that the focus on accuracy rarely provides additional value for developers because the recommendations are obvious[23]. Furthermore, in teams with strong code ownership, finding relevant experts is not problematic [47]. For replication completeness we calculated MRR. Our results confirm Kovalenco *et al.*'s findings that a broader perspective is needed when evaluating recommenders. We showed that recommenders with similar MRR values may have entirely different impact on *Expertise*, *CoreWorkload*, and *FaR*. For instance, *RevOwnRec* and *RetentionRec* have a difference of 0.06 in MRR while the difference between their ΔFaR is 81.10%. *LearnRec* and *TurnoverRec* have a difference of 0.07 in MRR while the difference between their ΔCoreWorkload and ΔFaR is 40.58% and 92.58%.

8 CONCLUDING REMARKS

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

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

We release *Sofia* bot as an open source software that fully integrates with GitHub pull requests and provides reviewer recommendations. The recommendations complement a developer's intuition and experience by providing simple rationale for each review candidate, such as showing how active a candidate has been, how many files he or she would learn about if they performed the review, and how many of the files under review they have modified or reviewed in the past. To the best of our knowledge, existing reviewer recommenders including Microsoft's CodeFlow [42] and Google's Gerrit [47] do not explicitly recommend reviewers based on distributing knowledge to reduce turnover. Future work is necessary to fully evaluate *Sofia* and to understand the costs and benefits of recommending "learner" reviewers in practice.

REFERENCES

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

[2] Vipin Balachandran. 2013. 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*. IEEE Press, 931–940.

[3] Lingfeng Bao, Zhenchang Xing, Xin Xia, David Lo, and Shanping Li. 2017. 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)*. IEEE, 170–181.

[4] Christian Bird, Alex Gourley, Prem Devanbu, Michael Gertz, and Anand Swaminathan. 2006. Mining email social networks. In *Proceedings of the 2006 international workshop on Mining software repositories*. ACM, 137–143.

[5] Christian Bird, Nachiappan Nagappan, Brendan Murphy, Harald Gall, and Premkumar Devanbu. 2011. 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*. ACM, 4–14.

[6] Amiangshu Bosu, Jeffrey C Carver, Christian Bird, Jonathan Orbeck, and Christopher Chockley. 2016. 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 (2016), 56–75.

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

[8] Gerardo Canfora, Massimiliano Di Penta, Rocco Oliveto, and Sebastiano Panichella. 2012. 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*. ACM, 44.

[9] Eleni Constantinou and Tom Mens. 2017. An empirical comparison of developer retention in the RubyGems and npm software ecosystems. *Innovations in Systems and Software Engineering* 13, 2-3 (2017), 101–115.

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

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

[12] Matthieu Foucault, Marc Palyart, Xavier Blanc, Gail C Murphy, and Jean-Rémy Falleri. 2015. Impact of developer turnover on quality in open-source software. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 829–841.

[13] Thomas Fritz, Gail C Murphy, and Emily Hill. 2007. 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*. ACM, 341–350.

[14] Tudor Girba, Adrian Kuhn, Mauricio Seeberger, and Stéphane Ducasse. 2005. How developers drive software evolution. In *Eighth International Workshop on Principles of Software Evolution (IWPSE'05)*. IEEE, 113–122.

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

[16] Georgios Gousios, Martin Pinzger, and Arie van Deursen. 2014. An exploratory study of the pull-based software development model. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, 345–355.

[17] Michaela Greiler, Christian Bird, Margaret-Anne Storey, Laura MacLeod, and Jacek Czerwonka. 2016. Code Reviewing in the Trenches: Understanding Challenges, Best Practices and Tool Needs. (2016).

[18] Christoph Hannebauer, Michael Patalas, Sebastian Stünkel, and Volker Gruhn. 2016. Automatically recommending code reviewers based on their expertise: An empirical comparison. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 99–110.

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

[20] Daniel Izquierdo-Cortazar, Gregorio Robles, Felipe Ortega, and Jesus M Gonzalez-Barahona. 2009. Using software archaeology to measure knowledge loss in software projects due to developer turnover. In *2009 42nd Hawaii International Conference on System Sciences*. IEEE, 1–10.

[21] Gaeul Jeong, Sunghun Kim, Thomas Zimmermann, and Kwangkeun Yi. 2009. 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)* (2009), 1–18.

[22] Huzefa Kagdi, Maen Hammad, and Jonathan I Maletic. 2008. Who can help me with this source code change?. In *2008 IEEE International Conference on Software Maintenance*. IEEE, 157–166.

[23] Vladimir Kovalenko, Nava Tintarev, Evgeny Pasynkov, Christian Bird, and Alberto Bacchelli. 2018. Does reviewer recommendation help developers? *IEEE Transactions on Software Engineering* (2018).

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

[25] Jakub Lipcak and Bruno Rossi. 2018. A Large-Scale Study on Source Code Reviewer Recommendation. In *2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE, 378–387.

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

[27] Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E. Hassan. 2016. An Empirical Study of the Impact of Modern Code Review Practices on Software Quality. *Empirical Software Engineering* 21, 5 (2016), 2146–2189.

[28] Ehsan Mirsaeedi and Peter C. Rigby. 2020. GitHub App: Sofia Bot. https://github.com/apps/sofiarec. (2020).

[29] Ehsan Mirsaeedi and Peter C. Rigby. 2020. Replication Package and RelationalGit. https://github.com/cesel/relationalgit. (2020).

[30] Ehsan Mirsaeedi and Peter C. Rigby. 2020. Sofia Bot Source code. https://github.com/cesel/Sofia. (2020).

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

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

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

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

[35] Loo Geok Pee, Atreyi Kankanhalli, Gek Woo Tan, and GZ Tham. 2014. Mitigating the impact of member turnover in information systems development projects. *IEEE Transactions on Engineering Management* 61, 4 (2014), 702–716.

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

[37] Adam Porter, Harvey Siy, Audris Mockus, and Lawrence Votta. 1998. Understanding the sources of variation in software inspections. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 7, 1 (1998), 41–79.

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

[39] Mohammad Masudur Rahman, Chanchal K Roy, and Jason A Collins. 2016. 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)*. IEEE, 222–231.

[40] Mehvish Rashid, Paul M Clarke, and Rory V OâĂŹConnor. 2017. Exploring knowledge loss in open source software (OSS) projects. In *International conference on software process improvement and capability determination*. Springer, 481–495.

[41] Peter Rigby, Brendan Cleary, Frederic Painchaud, Margaret-Anne Storey, and Daniel German. 2012. Contemporary peer review in action: Lessons from open source development. *IEEE software* 29, 6 (2012), 56–61.

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

[43] Peter C Rigby, Daniel M German, Laura Cowen, and Margaret-Anne Storey. 2014. Peer review on open-source software projects: Parameters, statistical models, and theory. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 23, 4 (2014), 35.

[44] Peter C Rigby and Margaret-Anne Storey. 2011. Understanding broadcast based peer review on open source software projects. In *2011 33rd International Conference on Software Engineering (ICSE)*. IEEE, 541–550.

[45] P. C. Rigby, Y. C. Zhu, S. M. Donadelli, and A. Mockus. 2016. 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)*. 1006–1016. https://doi.org/10.1145/2884781.2884851

[46] Martin P Robillard, Mathieu Nassif, and Shane McIntosh. 2018. Threats of Aggregating Software Repository Data. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 508–518.

[47] Caitlin Sadowski, Emma Söderberg, Luke Church, Michal Sipko, and Alberto Bacchelli. 2018. Modern code review: a case study at google. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*. ACM, 181–190.

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

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

[50] Patanamon Thongtanunam, Shane McIntosh, Ahmed E Hassan, and Hajimu Iida. 2016. 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. ACM, 1039–1050.

[51] Patanamon Thongtanunam, Shane McIntosh, Ahmed E. Hassan, and Hajimu Iida. 2017. Review Participation in Modern Code Review: An Empirical Study of the Android, Qt, and OpenStack Projects. *Empirical Software Engineering* 22, 2 (2017), 768–817.

[52] Patanamon Thongtanunam, Chakkrit Tantithamthavorn, Raula Gaikovina Kula, Norihiro Yoshida, Hajimu Iida, and Ken-ichi Matsumoto. 2015. 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)*. IEEE, 141–150.

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

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

[55] Yue Yu, Huaimin Wang, Gang Yin, and Tao Wang. 2016. Reviewer recommendation for pull-requests in GitHub: What can we learn from code review and bug assignment? *Information and Software Technology* 74 (2016), 204–218.

[56] Motahareh Bahrami Zanjani, Huzefa Kagdi, and Christian Bird. 2016. Automatically Recommending Peer Reviewers in Modern Code Review. *IEEE Trans. Softw. Eng.* 42, 6 (June 2016), 530–543. https://doi.org/10.1109/TSE.2015.2500238