Exploring the Notion of Risk in Code Reviewer Recommendation

Farshad Kazemi*, Maxime Lamothe†, Shane McIntosh*
*University of Waterloo, Canada, †Polytechnique Montréal, Canada
E-mail: {given_name}.{family_name}@{*uwaterloo.ca||†polymtl.ca}

***Abstract*—Reviewing code changes allows stakeholders to improve the premise, content, and structure of changes prior to or after integration. However, assigning reviewing tasks to team members is challenging, particularly in large projects. Code reviewer recommendation has been proposed to assist with this challenge. Traditionally, the performance of reviewer recommenders has been derived based on historical data, where better solutions are those that recommend exactly which reviewers actually performed tasks in the past. More recent work expands the goals of recommenders to include mitigating turnover-based knowledge loss and avoiding overburdening the core development team. In this paper, we set out to explore how reviewer recommendation can incorporate the risk of defect proneness. To this end, we propose the Changeset Safety Ratio (CSR) – an evaluation measurement designed to capture the risk of defect proneness. Through an empirical study of three open source projects, we observe that: (1) existing approaches tend to improve one or two quantities of interest, such as core developers workload while degrading others (especially the CSR); (2) Risk Aware Recommender (RAR) – our proposed enhancement to multi-objective reviewer recommendation – achieves a 12.48% increase in expertise of review assignees and a 80% increase in CSR with respect to historical assignees, all while reducing the files at risk of knowledge loss by 19.39% and imposing a negligible 0.93% increase in workload for the core team; and (3) our dynamic method outperforms static and normalization-based tuning methods in adapting RAR to suit risk-averse and balanced risk usage scenarios to a significant degree (Conover's test, $\alpha < 0.05$; small to large Kendall's W).**

***Index Terms*—Code Review Recommendation, Mining Software Repositories, Software Quality**

I. Introduction

Code review is the process by which developers assess each other's code changes [1]. This process can help to prevent bugs in their early stages before they are merged into the code base [2]. The tool-based code review process is known to provide stakeholders with technical [3–5] and non-technical [6] benefits. A popular form of code review, which often involves a tool-based procedure, is called modern code review [7].

Finding reviewers with the time to review a code change and familiarity with the modified subsystems has been a challenge in organizations who adopt code review [8, 9]. This is especially the case for large organizations with hundreds of developers. In such organizations, the author of a contribution may not yet have a professional relationship with the team responsible for overseeing the development of all of the components that they have changed. Code Reviewer Recommendation (CRR) aims to help stakeholders to find suitable reviewers [10].

While early reviewer recommendation studies were evaluated against historical records, i.e., who performed each task in the past [11], more recent work explores how recommendation approaches can be used to balance quantities of interest [12, 13]. These approaches consider previous interactions of the candidates with the modified files, the workload of the candidates at the time of the code review, and previous interactions between the developers in the project. Candidate reviewers are then ranked based on these metrics, and top-ranked candidates are suggested to decision-makers.

The results from previous studies suggest reviewers who share properties with those who performed similar reviews in the past and improve evaluation metrics such as files at risk. While the measures that have been proposed by previous studies align with important dimensions, the risk of defect proneness has not been explored. The risk of defect proneness of a code change indicates how probable it is for the change to induce fixes in the future. As an intervention, changes with a high risk of inducing future fixes may be assigned to subject matter experts for review. Prior work suggests that subject matter experts may be more adept at identifying problems during the review process [14, 15]. However, this intervention is likely to impose a greater burden on key team members.

In this paper, we take the position that an ideal recommendation approach should balance the trade-off between the burden on expert reviewers and the risk of defect proneness. Therefore, we set out to incorporate defect proneness in the reviewer recommendation process. More specifically, we set out to address the following research questions:

RQ1 How do existing code reviewer recommenders perform with respect to the risk of inducing future fixes?
Motivation: Every code change induces some degree of risk. The degree of risk varies based on the change and its domain [16]. A key goal of the code review process is assessing and mitigating the risk of introducing defects during or shortly after the code integration process [17]. It is crucial to involve subject matter experts in the review process to achieve that goal. Otherwise, if non-experts review high-risk tasks, defects may slip through the integration process. Thus, we first set out to understand how well existing reviewing assignments and CRR-based reassignments perform in terms of risk mitigation.
Results: We observe an inherent trade-off between our studied quantities of interest. For instance, the RetentionRec recommender – a reviewer recommendation

approach proposed to minimize the risk of developer turnover-based knowledge loss while ignoring other quantities of interest – reduces files at risk by up to 23.89% with respect to the reviewers who have already performed the review. On the other hand, RetentionRec underperforms in terms of the Changeset Safety Ratio (CSR) – a measure that we propose to indicate the performance of a recommendation approach concerning the safety of the code change process – by 4.56% to 37.07%.

RQ2 How can the risk of fix-inducing code changes be effectively balanced with other quantities of interest?
Motivation: Optimizing for other quantities of interests, such as files at risk of turnover, without considering defect proneness is unlikely to perform well due to the inherent trade-offs discovered in RQ1. Therefore, an approach is needed to incorporate defect proneness in recommendation decisions without overly disrupting other quantities of interest. To that end, we propose RAR – a reviewer recommendation approach that aims to incorporate defect risk into recommendations – and set out to evaluate how well it performs.
Results: Our experiments indicate that RAR increases the expertise of reviewers assigned to reviews by 12.48% and the CSR by 80.00% while reducing files at risk of turnover by -19.39% and only increasing the core development team workload by 0.93%. Moreover, we find that project or team-specific tolerance of risk can be incorporated by adjusting the threshold P_D, which is the threshold of the likelihood of fix-inducing PRs at which changes are deemed risky enough to require intervention. The effective P_D interval is defined as the change interval for which the performance of the RAR is impacted. For instance, in Roslyn, the effective interval of P_D is 0 - 1; however, the effective interval of P_D is 0 - 0.3 and 0 - 0.1 for the Kubernetes and Rust projects, respectively. Thus, P_D must be calibrated to its effective range for RAR to achieve optimal results.

RQ3 How can we identify an effective fix-inducing likelihood threshold (P_D) interval for a given project?
Motivation: The performance of RAR depends on the P_D setting. P_D itself is dependent on a project's past defect proneness. Moreover, different projects may assign different weights to the importance of defect proneness. Therefore, we set out to propose approaches to support stakeholders in tuning P_D to an appropriate value for their development context.
Results: We propose static, normalization, and dynamic approaches to tune the value of P_D. Results that explore P_D settings in risk-averse, risk-tolerant, and balanced contexts indicate that the proposed methods affect the performance of RAR significantly. Moreover, the dynamic method outperforms the others in risk-averse and balanced contexts to a statistically significant (Conover's Test, $\alpha < 0.05$) and practically significant degree (Kendall's W = 0.0727 - 0.543, small - large).

II. Related Work

In this section, we describe related studies on defect proneness prediction and reviewer recommendation approaches.

Reviewer Recommendation. The main task of a reviewer recommender is to suggest suitable reviewers for reviewing tasks. Reviewer recommenders often leverage historical data to make recommendations [18, 19]. Other approaches aim to optimize other characteristics, such as workload balance [12, 20] or distributing knowledge [13]. Regardless of the optimization method, when a new Pull Request (PR) is created, the recommender ranks potential candidates based on the score that has been calculated by its objective function.

Recently, however, early work has explored a change in perspective of the goal of the reviewer recommendation process. Kovalenko *et al.* [21] observed that developers are often aware of the top recommendations of CRR approaches, suggesting that other goals, such as workload balancing, might be more appropriate. Gauthier *et al.* [22] found that history-based evaluations of reviewer recommenders are often more pessimistic than optimistic since the proposed reviewers who did not perform a review (i.e., incorrect recommendations) often reported high comfort levels with those review tasks. Mirsaeedi and Rigby [13] proposed Sofia, a multi-objective recommendation system that tries to maximize reviewer expertise and minimize the risk of turnover-based knowledge loss, as well as the workload on the core development team. In this paper, we set out to complement the prior work by also incorporating estimates of the risk of inducing future fixes in the recommendation process. To this end, we evaluate three different projects using seven different approaches. We use cHRev [19] as a conventional recommender to suggest reviewers. In addition, we consider greedy recommendation strategies like LearnRec [13], which tries to maximize the learning from a PR. We also evaluated Sofia [13] as a state-of-the-art recommender.

Defect Prediction. Defect prediction models are used to help the stakeholders of a project focus their limited resources on bug-prone modules [23]. Practitioners have used defect prediction systems to find bugs in their early stages, reducing technical debt [24] and the effort required to fix them. These models can also help teams identify buggy changes before they are merged into the repository. These defect prediction models are often trained using historical data and then used to assess new code changes by estimating the likelihood that a given code change will induce a future fix (i.e., estimating the fix-inducing likelihood). There have been a plethora of contributions on defect prediction, but we focus below on two lines of work that are most relevant, i.e., (1) approaches to more accurately identify fix-inducing changes and (2) proposed indicators of fix-inducing commits. Just-In-Time (JIT) defect prediction models — like any prediction model — will only be as good as their training data. Since the true set of fix-inducing changes is not clearly labelled in historical software data, heuristic approaches are used to recover that signal.

The SZZ algorithm [25] first identifies bug-fixing commits by mining for keywords such as "fix" or "bug" in commit

messages. Next, potential fix-inducing commits are associated with these fixes by tracing lines that were removed or modified back to the commit(s) that introduced them. Finally, filters are applied to remove potential fix-inducing changes that are unlikely to have caused the bug (e.g., potentially fix-inducing commits that were recorded after the bug was created in the issue tracker). The SZZ algorithm has seen several revisions in the literature [26, 27]. Since improving SZZ is beyond the scope of this paper, we use the off-the-shelf implementation of SZZ available in the Commit Guru tool [28].

The set of indicators that are used to predict fix-inducing changes are derived from the change itself, historical tendencies of the modified areas of code, and characterization of the personnel involved with the change [23]. For example, Kamei *et al.* [29] used measures of the size, purpose, and diffusion of a change, as well as the historical tendencies of the modified modules and the experience of change authors to estimate the likelihood of a change to induce future fixes. Hoang *et al.* [30] and McIntosh *et al.* [31] expanded the set of measures to include review metrics such as iterations, number of reviewers, and comments. Pascarella *et al.* [32] added more detailed measures such as owner's contribution lines, and change code scattering. In this paper, we use the set of measures that have been provided by Commit Guru to calculate the 13 metrics similar to Kamei *et al.* 's set of measures for various Pull Requests (PR) based on PR's Commits.

Different variables are used as defect prediction model input, usually based on the change, source code metrics and historical data [23]. However, some studies have used code change chunks as well as metrics such as developer networks to determine the buggy commits [33]. In this study, we use the output of Commit Guru to identify commit features. The tool needs a GitHub repository address to perform the SZZ analysis on the repository. Given a specific Git branch, Commit Guru starts analyzing the desired Git branch and identifies the fix-inducing commits using the SZZ algorithm. The implementation details of the Commit Guru tool can be found in a study by Kamei *et al.* [29]. Commit Guru extracts thirteen metrics for each commit (Table 2 in supporting materials [34]) and a flag that indicates whether the commit is suspected to be fix-inducing. We use these metrics and the fix-inducing flag to predict the defect proneness of code changes.

III. Studied Datasets

In this section, we present the sources of data and the projects used to conduct our study and the rationale for their selection.

Data Source. To evaluate RAR, we seek to ground our analysis in a comparison to previous multi-objective reviewer recommenders [13]. Therefore, to obtain a fair comparison, we begin with the same subject systems that Mirsaeedi and Rigby studied [13]. However, two of these projects, CoreFx and CoreCLR have been since merged with the *.Net Runtime* project. Due to this migration, Commit Guru was unable to obtain the necessary information for the prediction model and rendered us unable to process the master branch for possible fix-inducing commits. As a result, we omit CoreFx and CoreCLR, focusing our analysis on Rust, Kubernetes, and Roslyn. Rust and Kubernetes are community-driven projects, and Roslyn is an industry project developed openly on GitHub. These projects are well-established (more than four years old) with more than 10K PRs. Kubernetes has had a significant impact on cloud computing platforms with more than 3.1K contributors. Roslyn, with 524 contributors, is an open source .NET compiler platform for languages such as C# and VB. Finally, Rust, with 3.8K contributors, is a multi-paradigm, general-purpose programming language. Further details of these projects are listed in Table 1 of our supporting materials [34].

Data Collection. We begin our data collection process by downloading the relevant details from the replication package provided by Mirsaeedi and Rigby [35]. The shared data includes commits, files that have been modified in each commit, developers involved in a PR, a list of developers and reviewers of each PR, and developers' interaction with the PR. To perform defect analysis, our approach requires a list of the commits that comprise each of the PRs. Moreover, we need to compute the measures listed in Table 2 in supporting materials [34] to train our defect prediction model. We use the GitHub API to gather the additional data for each PR in the data set. We did not use the commits of a PR to calculate additions and deletions since they might have cancelled each other out (e.g., one line added in one commit could be removed in the next commit of the same PR). Instead, we calculate the net number of additions and deletions extracted directly for each of the PRs.

IV. Study Design

This study is comprised of two parts: (1) identifying fix-inducing PRs and (2) evaluating reviewer recommendation approaches. This section describes each part of our study and explains the rationale behind our design decisions.

A. Identifying and Predicting Fix-Inducing PRs

Because our approach aims to incorporate the notion of risk in the recommendation process, identifying fix-inducing PRs with which to evaluate our approach is an important part of the study. In this study, we operationalize risk by mining the repositories of the studied projects for defect-fixing, and fix-inducing commits using Commit Guru [28]. Figure 1 provides an overview of our discovery process for risky PRs.

Step1: Extract defect prediction data: We first apply Commit Guru [28] to the studied repositories in order to produce data sets of fix-inducing commits, as well as a popular set of measures for their prediction. Commit Guru clones each repository, computes commit-level measures that share a relationship with risk (e.g. patch size, diffusion), and applies the SZZ algorithm [25] to identify which historical commits have induced future fixes. Finally, a logistic regression model is fit to estimate the riskiness of code changes. Table 2 in supporting materials [34] shows the set of used risk measures.

Although studies by Quach *et al.* showed some of the limitations of SZZ ([36, 37]), its output is still an indicator of bug-inducing probability. Moreover, we decided not to use

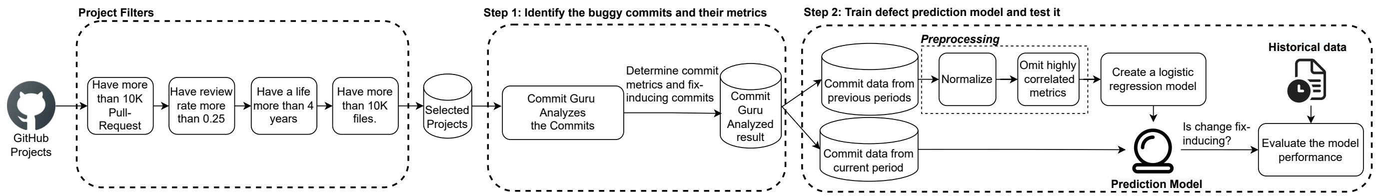


Figure 1: The simplified overall architecture of the project selection filters and the defect prediction process.

manually verified bug datasets such as the one by Rodriguez-Perez *et al.* [38] as we wanted to view the effects of the recommendation approaches in their natural habitat, which would normally be automated and include tools such as SZZ.

Step2: Train and test PR-level risk model: We use the risk measures extracted by Commit Guru to train defect prediction models. A logistic regression method is used to train the model for each quarter (three months). The three-month time interval is based on similar studies, like Mirsaeedi and Rigby [13], and retraining this period length setting allows us to extend reviewer recommendation approaches to incorporate risk and more directly compare results. Moreover, updating the prediction model in short (three months) intervals has been recommended to counteract concept drift [39]. This step is decomposed into the following tasks:

1) ***Data preprocessing.*** Before training the models, data must be preprocessed to counteract biases. First, we standardize the risk measures since their magnitudes vary broadly. We use Scikit StandardScaler[1] to transpose all risk measures' values to have zero mean and unit variance. Then, we identify highly correlated measures, as they affect the model's performance. To this end, we calculate pairwise Pearson correlation (ρ) between each pair of risk measures. As suggested by Tay [40], any pair of risk measures with $|\rho| > 0.6$ is considered to have too much similarity to include in the same model fit. In such cases, we remove all the measures but one (Based on their order of appearance as listed in Table 2 in supporting online materials [34]).
2) ***Fit defect prediction model.*** Once the data has been preprocessed, we use the data to fit a logistic regression model for every quarter using the previous quarters' data. The model then estimates the likelihood that each code change will be fix-inducing in the following quarters.
3) ***Aggregate risk estimates to the level of PRs.*** Using the trained models, we estimate the riskiness of each PR by aggregating the risk measures across all of the PR changes. We use the PR's commits risk measures to calculate the risk measures for a PR. Table 2 in the supporting online materials [34] has a brief explanation of how each of these risk measures is calculated from the set of commits belonging to a PR. Using the PR risk measures, the model estimates the PR's likelihood of inducing a future fix. We use the balanced accuracy performance measure to evaluate the performance of our models since our datasets are inherently imbalanced, i.e. there are more non-fix-inducing PRs than fix-inducing PRs. The median balanced accuracy over different periods for Roslyn, Rust, and Kubernetes projects are 75.9%, 50%, and 97.5%, respectively.

[1]https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html

B. Ranking potential reviewers of a PR

As the next step, we use the fix-inducing likelihood of the PR and its risk measures to suggest reviewers for each PR. We evaluate seven baseline approaches (RQ1) as well as our proposed method, RAR (RQ2). We describe the baseline approaches below, and describe RAR in Section VI.

1) AuthorshipRec: Suggested by Mockus and Herbsleb [41], the authorship of a file is an important factor when assigning software experts to (reviewing) tasks. Bird *et al.* [42] formulated the AuthorshipRec in their paper based on the proportion of the files that a developer modified prior to the PR.

2) RevOwnRec: Thongtanunam *et al.* [4] suggested a new reviewer recommender based on the developers' previous review history. The rationale was that the project code reviewers for each project subsystem are constant most of the time. Similarly to AuthorshipRec, RevOwnRec considers the proportion of a developer's reviews or modifications relative to all of the reviews and modifications in a PR.

3) cHRev Recommender: The cHRev recommender [19] is a popular conventional reviewer recommender. When ranking developers as potential candidates of a code change, cHRev considers the developer's expertise from previous reviews as well as the recency of the contributions. To rate the fit of a developer D for reviewing a file F, the *xFactor* was used:

$$xFactor(D, F) = \frac{C_f}{C'_f} + \frac{W_f}{W'_f} + \frac{1}{|T_f - T'_f| + 1} \qquad (1)$$

Where C_f, W_f, and T_f represent the number of review comments, the number of workdays that D commented on the file's reviews, and the most recent day that D worked on F, respectively. The prime versions of the variables in the denominator represent the total number used to normalize the output. Then, the fit for each developer is estimated using the summation of the *xFactor* for all the files in the code change.

4) LearnRec: The LearnRec recommender is designed to distribute knowledge among team members. *LearnRec* suggests developers who are poised to learn the most from reviewing a PR. *ReviewerKnows* has been suggested as a way to measure how knowledgeable a potential reviewer is about a review request [13]. The *ReviewerKnows* estimates how familiar a

developer would be with the modified files of a review request. It is usually favourable to distribute the knowledge among developers in repositories to mitigate any loss of knowledge if any developer leaves the project. To this end, *LearnRec* is formulated by subtracting *ReviewerKnows* from one, which estimates how much a developer can learn by reviewing a PR. This metric can be used to create a reviewer recommender that distributes the knowledge among the project developers by assigning the review to the developer with the largest *LearnRec*.

5) RetentionRec: Although LearnRec seems like a reasonable choice to prevent knowledge loss, in reality, many developers do not contribute to a project over a long time [43]. Those who stand to learn the most may leave the project before that knowledge can be put to use. To mitigate this issue, *contribution ratio* and *consistency ratio* have been proposed. The *contribution ratio* for a developer is the proportion of contributions during the previous particular period of time (e.g., one year) for which the contributor is responsible. The *consistency ratio* is the proportion of sub-periods (e.g., months) that the developer was actively contributing to the project throughout a study period (e.g., year). As developers become more consistent or more (proportionally) active, the *RetentionRec* increases, suggesting that it is less likely that they will leave the project.

6) TurnoverRec: Mirsaeedi and Rigby [13] multiplied *RetentionRec* and *LearnRec* and created *TurnoverRec*. This recommender helps with distributing knowledge among the more active members of the development team. Recommending reviewers based on this measure minimizes the risk of turnover-induced knowledge loss caused by developers leaving the company by distributing knowledge among active members.

7) Sofia: Sofia [13] is a combination of *TurnoverRec* and *cHRev* whose objective is to distribute knowledge among the more active team members whenever files with a large risk of knowledge loss are present in a PR. The scoring function used for the developer (D) and the code change R is:

$$\begin{cases} \mathit{cHRev}(D,R), & \mathit{if}\ |\mathit{knowledgeable}(f)| \leq d, \mathit{any} f | f \in R \\ \mathit{TurnoverRec}(D,R), & \mathit{otherwise} \end{cases} \quad (2)$$

We consider d=2 in this equation, similar to the original work by Mirsaeedi and Rigby [13], to prevent any knowledge loss by leaving one developer from the team.

C. Recommendation Component

We apply these reviewer recommender to our datasets and calculate the recommenders' scores for all the candidate reviewers. We then rank potential candidates based on the scores. Configurable parameters include the number of reviewers per PR and the maximum number of files per PR for the reviewer's knowledge. For the purposes of this paper, we choose only the top suggested candidate per PR and randomly replace it with one of the actual reviewers (to match prior work [13]). We only consider PRs with less than 100 files and do not associate the PR with developers' knowledge otherwise regarding maximum files per PR. It is because one developer cannot perceive large code changes as argued by Bird *et al.* [44].

V. Evaluation Setup

In this section, we describe the evaluation metrics used to assess the performance of reviewer recommenders and our rationale for selecting those metrics. As explained in Section II, conventional recommendation approaches aim to recommend the reviewers who performed the task [19, 45–47]. However, Kovalenko *et al.* [21], suggest that recommending the reviewer who reviewed a PR provides little value to the project. Furthermore, there exist many qualified developers who may not have reviewed PR but would have been comfortable doing so [22]. Conventional evaluation methods consider these recommendations incorrect and penalize the recommenders for making such suggestions.

To assess the effect of a recommendation approach on the mitigation of the risk of fix-inducing PRs, we leverage the simulation approach presented by Mirsaeedi and Rigby [13]. These measures quantify previously discussed aspects of the reviewer recommendation process and estimate the performance of a reviewer recommender through history-based simulation. We run simulations for the selected projects and compare the outcome of the recommenders with one another with respect to the evaluation measures. We expand the set of evaluation measures proposed by Mirsaeedi and Rigby [13] to incorporate the CSR — a cumulative measure of the risk of fix-inducing changes in a given period of time. These measures originated from the challenges and expectations of the researchers who studied the code review process and recommendation approaches prior to this study [1].

In the remainder of this section, we explain each of the recommendation evaluation measures we employ in this study.

Expertise. *Expertise* of the reviewers assesses the recommended reviewers by the expertise that they have in the PRs they have been tasked to review. It is the primary evaluation criterion used in past studies [48, 49]. Past work has indicated the important role that involving subject matter experts has on the review process [6, 14]. To quantify this measure, Mirsaeedi and Rigby [13] proposed the following measure:

$$\mathit{Expertise}(Q) = \sum_{R}^{\mathit{Reviews}(Q)} \frac{\mathit{FileReviewersKnow}(R)}{\mathit{FileUnderReview}(R)} \quad (3)$$

Where Q is the quarter in which this metric is calculated. A developer is assumed to know a file if they have modified or reviewed the file prior to the PR reviewing task.

CoreWorkload. Having all PRs reviewed by experts is ideal, but there is an inherent trade-off between the time that experts invest in reviewing PRs and the amount of time they have for other development tasks [14]. The problem amplifies as projects grow if the core developer teams do not grow as well. Mirsaeedi and Rigby [13] proposed a static core team size of the top 10 reviewers and using the following equation, estimate the reviewing workload that the core team is coping with:

$$\mathit{CoreWorkload}(Q) = \sum_{D}^{\mathit{Top10Reviewers}(Q)} \mathit{NumReviews}(D) \quad (4)$$

Files at Risk of turnover (FaR). The loss of knowledge caused by knowledgeable developers leaving a project may consume resources and even stall its progress. The *File at Risk of turnover (FaR)* measures the number of files known by zero or one developer in a period of one quarter. The formula [44] to calculate this measure is:

$$FaR(Q) = \{f | f \in Files, |ActiveDevs(Q, F)| \leq 1\} \quad (5)$$

Where *ActiveDevs* represent the developers who are familiar with the set of files F and are still actively contributing to the project by the end of quarter Q.

Changeset Safety Ratio (CSR). The replacement of reviewers does not affect the incidences of bugs in our simulation. Instead, to assess the impact of replacing reviewers on risk, we assume that having an expert, preferably one who has recently interacted with files in the code change, will reduce the likelihood of merging fix-inducing code changes [42]. To this end, we formulate the Changeset Safety Ratio(CSR) as a measure of how well the review assignments have mitigated the fix-inducing likelihood of a set of PRs:

$$CSR(Q) = \sum_{R}^{Reviews(Q)} (1 - DefectProb(R)) \times MaxXFactor(R) \quad (6)$$

The *DefectProb* is the risk estimate of a PR being fix-inducing, and the *MaxXFactor* is the maximum score of the *xFactor* (equation 1) among all the suggested reviewers of a PR. The xFactor incorporates both the recency and quantity of contributions in assessing reviewer expertise and is at the core of the cHRev recommender [19]. If the risk of inducing a future fix that a PR presents is small, we may assign developers with less expertise to that code change without impacting the CSR disproportionately. Increases in CSR indicate that the code change is less likely to be fix-inducing or that the developer's maximum expertise has increased. In either case, increases to CSR suggest that the review process is performing well in terms of risk mitigation.

VI. Experimental Results

In this section, we describe our experiments, the results and the analysis of the results. We use the percentage of change to evaluate different reviewer recommenders' performance:

$$\Delta MeasureChange(Q) = (\frac{SimulatedMeasure(Q)}{ActualMeasure(Q)} - 1) \times 100 \quad (7)$$

The $ActualMeasure$ and $SimulatedMeasure$ refer to the calculated evaluation metric for the historical data and a simulation run, respectively.

RQ1: How do existing code reviewer recommenders perform with respect to the risk of inducing future fixes?

In this experiment, we seek to determine whether reviewer recommenders mitigate the risk of inducing future fixes by introducing an evaluation measure (*CSR*).

Approach. For each studied system, we analyze the historical data and fit one model per quarter to estimate the likelihood that a PR is fix-inducing. Then, starting from the second quarter, we use a model fit of the previous quarter to estimate the fix-inducing likelihood of each PR. We use PR metrics listed in table 2 in the supporting online materials [34] as the model's input. We then rank potential reviewers for each PR using the seven baseline recommendation approaches. For every PR in each studied system, we swap one of the actual reviewers with our top candidate and evaluate the performance of this change by calculating the *MeasureChange* according to Equation 7.

Table I: Recommender performance vs. reality. Up and down arrows indicate improvement and degradation, respectively.

CRR	Project	Expertise	Workload	FaR	CSR
AuthorshipRec	Roslyn	15.52% ↑	-7.045% ↑	34.91% ↓	17.50% ↑
	Rust	10.64% ↑	4.09% ↓	42.58% ↓	16.66% ↑
	Kubernetes	12.87% ↑	-2.07% ↑	18.60% ↓	18.36% ↑
RevOwnRec	Roslyn	21.82% ↑	1.83% ↓	17.5% ↓	2.76% ↑
	Rust	12.72% ↑	8.16% ↓	98.62% ↓	-9.57% ↓
	Kubernetes	18.56% ↑	3.89% ↓	-4.05% ↑	1.08% ↑
cHRev	Roslyn	12.35% ↑	-1.52% ↑	0% –	75.06% ↑
	Rust	7.72% ↑	-2.11% ↑	11.84% ↓	92.09% ↑
	Kubernetes	13.97% ↑	-3.06% ↑	-11.27% ↑	104.31% ↑
LearnRec	Roslyn	-23.85% ↓	-34.77% ↑	138.84% ↓	-36.20% ↓
	Rust	-50.27% ↓	-50.26% ↑	122.63% ↓	-61.44% ↓
	Kubernetes	-34.98% ↓	-34.55% ↑	49.1% ↓	-46.38% ↓
RetentionRec	Roslyn	22.92% ↑	20.36% ↓	-23.89% ↑	-27.22% ↓
	Rust	13.38% ↑	15.70% ↓	-16.86% ↑	-4.56% ↓
	Kubernetes	19.75% ↑	47.78% ↓	-20.94% ↑	-37.07% ↓
TurnoverRec	Roslyn	-14.66% ↓	0.67% ↓	-38.33% ↑	-33.51% ↓
	Rust	-34.21% ↓	-4.38% ↑	-23.66% ↑	-53.43% ↓
	Kubernetes	-25.72% ↓	-0.09% ↑	-30.32% ↑	-44.49% ↓
Sofia	Roslyn	7.38% ↑	4.03% ↓	-34.9% ↑	55.22% ↑
	Rust	4.97% ↑	0% -	-25.42% ↑	73.09% ↑
	Kubernetes	9.42% ↑	1.70% ↓	-28.67% ↑	96.74% ↑

Results. Table I presents the results of this experiment. The up and down arrows next to the numbers indicate performance improvement and degradation, respectively.

Analysis. For *AuthorshipRec*, code owners are predominantly assigned to reviews. Thus, increases to CSR are not unexpected, since coders owners are among the most knowledgeable contributors to whom reviewing tasks may be assigned. However, this assignment prevents others from learning about files they have not developed, which causes the *files at risk of turnover* measure to degrade. For *RevOwnRec*, each studied system has a trusted developer circle for the reviews; hence this recommender fails to optimally distribute knowledge and improve *files at risk of turnover*. Since these reviewers may not be the file owners, the *CSR* also tends to decrease or not to change considerably.

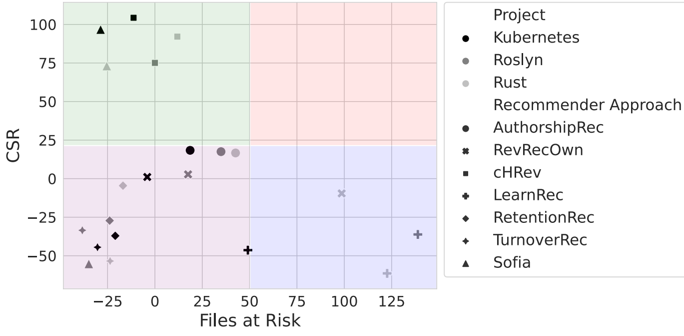


Figure 2: Relation analysis of *CSR* and *Files at Risk of turnover*.

For *cHRev*, the score function is based on *xFactor*. Hence, the *CSR* is consistently improved, notably at the cost of limiting the improvement of *workload* for the core development team in comparison to other recommendation approaches.

For *LearnRec*, there is no consideration for the retention of recommended candidates, so the *files at risk of turnover* measure tends to increase because many reviewers leave the project. The suggested reviewers by this recommendation system are not experts, but seek to learn by reviewing the PR, so the *CSR* measure tends to decrease.

For *RetentionRec*, the recommender suggests candidates with the most knowledge about the project, not a specific PR. As a result, undesirably, the *core developers' workload* increases because they are mostly permanent developers of a project. However, their knowledge causes *CSR* and *expertise* to improve.

For *TurnoverRec*, the recommender favours the most permanent candidates, regardless of the degree of knowledge that they have about the code being modified by a PR. This bias leads to knowledge retention, thus improving *files at risk of turnover*. However, since distributing knowledge among developers is an important risk mitigation measure, the choice of less knowledgeable candidates causes the *CSR* to decrease.

For *Sofia*, when none of the changed files are at risk of turnover, *cHRev* is used. This compensates for *expertise* and *CSR* measures that are lost due to knowledge distribution caused by *TurnoverRec*. However, most of the time, this is at the cost of increasing the *workload* for the core development team. *Sofia* uses *TurnoverRec* for changesets with files at risk of knowledge loss, which has a favourable effect and causes the *files at risk of turnover* measure to improve.

Figure 2 shows the relation between *CSR* and *files at risk of turnover* in our experiments. The bottom-left quadrant shows evidence of a trade-off between *CSR* and *Files at Risk* for approaches that optimize only one characteristic. Meanwhile, cHRev and Sofia mostly present results in the top-left quadrant. This indicates that they are generally robust to the trade-off between *CSR* and *files at risk of turnover*, and can broadly optimize both the risk of knowledge turnover and *CSR*. Finally, the bottom-right quadrant shows that optimizing for learning opportunities (e.g., using LearnRec) negatively impacts both *files at risk of turnover* and *CSR*.

These observations indicate that if there is no deliberate effort to distribute knowledge, as the *files at risk of turnover* improves, unless the necessary restrictions are put in place, such as a limitation on the most knowledgeable reviewers, the *CSR* degrade. This decrease, in turn, increases the chance of merging a fix-inducing PR into the project. This suggests that there is an inherent trade-off between the *files at risk of turnover* and *CSR* evaluation measures. This does not hold in all cases. For example, in *LearnRec*, both *files at risk of turnover* and *CSR* decreases which is likely because the recommended candidates leave the project as retention is not considered in the score function. Since leavers may leave a gap in the team understanding of an area of the codebase, the *files at risk of turnover* and *CSR* measures tend to degrade. For *Sofia*, the recommender's candidate scoring function maximizes the expertise of the reviewers unless there is a file with few knowledgeable developers in the changeset. In these cases *Sofia* tries to distribute knowledge which lessens the *core workload* and improves the *files at risk of turnover*. This active effort cancels out the native trade-off and improves both *files at risk of turnover* and *CSR*.Sofia works better in terms of fix-inducing code changes, but like other approaches, it does not have any parameter to control the sensitivity to these changes. The inflexibility may become a barrier to adoption for this recommender as it cannot be tuned to suit the needs of users.

The evaluation results indicate that unless active effort is put into knowledge distribution while keeping the expertise high, the CSR and files at risk of turnover have an innate trade-off. In cases where both CSR and files at risk of turnover are maximized, other measures such as core developer workload suffer. Hence, one cannot simultaneously optimize suggested reviewers with respect to the risks of knowledge loss and fix-inducing changes.

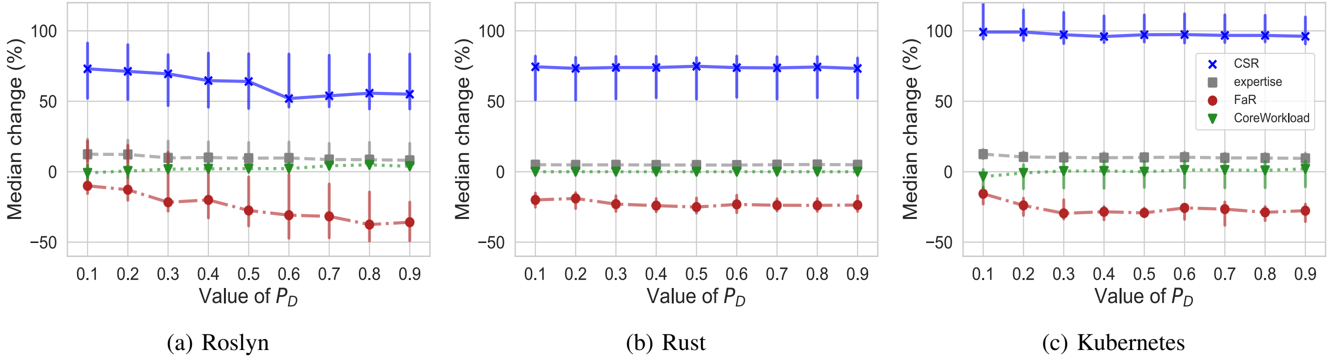
RQ2: How can the risk of fix-inducing code changes be effectively balanced with other quantities of interest?

To balance the innate trade-off between *files at risk of turnover* and *CSR,* we suggest using a hybrid reviewer recommendation approach to optimize the recommendation process based on the PR fix-inducing likelihood. We propose a recommender to improve the *CSR* when a PR has a high risk of being fix-inducing. The objective function for the proposed Risk Aware Recommender (RAR) is formulated as:

$$RAR(D,R) = \begin{cases} \textit{Sofia}(D,R), & \textit{DefectProb}(R) \leq P_D \\ \textit{cHRev}(D,R), & \textit{otherwise} \end{cases} \quad (8)$$

In this formula, the P_D represents the threshold for the likelihood of PRs to be fix-inducing. If the P_D threshold is exceeded, *cHRev* is used to suggest experts. Otherwise, *Sofia* will suggest reviewers for the PR. The *cHRev* ranks candidate reviewers based on their familiarity with the changed files while *Sofia* opportunistically distributes knowledge when the modified files are not at risk of turnover.

Approach. We study the performance of *RAR* in terms of the *coreWorkload*, *files at risk of turnover*, *expertise*, and *CSR* measures. We also study the impact that varying the P_D threshold from 0.1 to 0.9 has on RAR performance.



(a) Roslyn (b) Rust (c) Kubernetes

Figure 3: The effect of P_D on the performance of RAR for each evaluation metric, on different projects over different quarters.

Results. Figure 3 shows the evaluation measures as the P_D changes for the studied systems.

Analysis. Figure 3 shows that as the value of P_D increases, the tolerance of *RAR* for fix-inducing PR grows. As a result, we expect more knowledge distribution leading to a decrease in *CSR*. As fewer experts are assigned to the tasks, the overall *expertise* also diminishes, which is not an unexpected outcome.

However, there are project-specific trends that are worth noting. For example, Figure 3a shows that the evaluation measures for the Roslyn project are steadily declining as P_D increases, whereas Figure 3c shows that the majority of the impact of varying P_D in the Kubernetes project takes place between $P_D = 0.1$ and $P_D = 0.3$. Moreover, Figure 3b shows that for Rust, the impact of varying P_D is relatively small. Overall, the Risk Aware Recommender (RAR) yields an average change of 12.48%, 0.93%, -19.39% and 80.00% over different quarters for evaluation measures of Expertise, Core workload, *files at risk of turnover* and *CSR*, respectively.

A closer look at the model estimates of the likelihood of fix-inducing changes helps to explain these project-specific trends. Figure 4 shows the distributions of the estimated likelihood of changes being fix inducing stratified by project and quarter for four quarters (The complete distribution can be found in our supporting material's Figure 1 [34]) . We observe that, unsurprisingly, larger performance fluctuations in Figure 3 are associated with the P_D values where the majority of the estimated likelihoods lie in Figure 4. Moreover, despite an overall decreasing trend in terms of the likelihood of fix-inducing changes over time, the trend of each quarter is similar to its adjacent quarters. This local similarity may help stakeholders to effectively tune P_D values (see RQ3 for a more detailed analysis).

> *The RAR settings can be tuned to balance the risks of knowledge loss and fix-inducing changes. Indeed, as the threshold for indicating tolerance of the risk of fix-inducing changes increases, the risk of knowledge loss impacts fewer files. However, identifying the optimal threshold setting requires an awareness of project-specific trends in the model estimates of the likelihood of fix-inducing changes.*

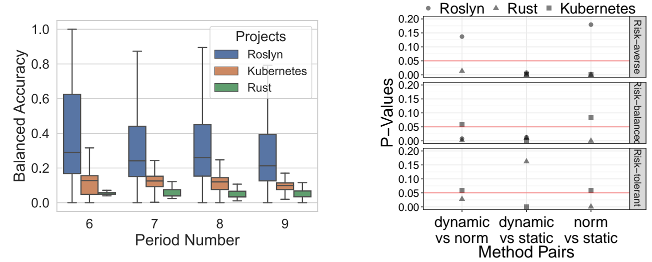


Figure 4: Distributions of predicted defect probabilities.

Figure 5: Conover Test results.

RQ3: How can we identify an effective fix-inducing likelihood threshold (P_D) interval for a given project?

Our analysis from RQ2 indicates that the performance of the *RAR* is sensitive to the P_D setting. The effective range of P_D is dependent on the past likelihood of fix-inducing changes. In this question, we seek to propose an approach to help project stakeholders in the selection of effective P_D settings based on their tolerance for the risk of fix-inducing changes.

Approach. We explore the following three approaches to identify effective periods:

- *Static method*: This baseline considers the effective period spans the entire range between 0 and 1.
- *Normalization method*: The effective range spans between upper and lower extremes of the distribution of likelihoods from the prior periods. To match common outlier definitions, we set out lower and upper extremes to $Q_1 - 1.5 \times IQR$ and $Q_3 + 1.5 \times IQR$, respectively, where Q_i is the ith Quartile, and IQR is the Interquartile range between Q_1 and Q_3. All examples within the range are normalized by the maximum value.
- *Dynamic method*: A selective variant of the normalization method. Instead of considering all previous periods, we only consider the last six months. This allows the model to focus on the current part of the project life cycle.

For each of these three methods, we simulate three different thresholds: 25% (risk-averse recommendation), 50% (balanced recommendation), and 75% (risk-tolerant recommendation) of the effective period for our three projects in the dataset.

Results. Figure 6 shows distributions of relative improvement in *CSR* that are achieved for different time periods (points) of the studied systems (plot columns) of our approaches (y-axes) in different configurations (plot rows).

Analysis. We use the Friedman test (two-tailed, paired, $\alpha = 0.05$) [50] and apply it to the *CSR* performance data (Table II). We observe significant differences between the investigated methods in all configurations except for Roslyn in the risk tolerant setting. Next, we use Kendall's W to determine the magnitude of this effect [51] (Table III). Large and small effects are observed in 55% and 22% of the cases, respectively.

We apply the Conover test to discern which pairs cause this significant difference [52]. Figure 5 shows p-values for different thresholds with red lines indicating the 0.05 confidence interval. The results imply that the dynamic method significantly affects risk-averse ($P_D = 25\%$) and risk-balanced ($P_D = 50\%$) recommendations in all studied systems. For the normalization method, the effect on the results is inconsistent. The dynamic method considers the pivot of the project in various periods, which affects the *CSR*. In contrast, the normalization method considers the entire history and may not be sensitive enough to react to risk fluctuations as projects age [53].

On the other hand, for risk-tolerant recommendations ($P_D = 75\%$), none of the methods have a consistent effect on the results due to the difference in the distribution of defect proneness for various periods. Roslyn has a high rate of fix-inducing PRs ($P_D > 0.5$) in all the periods, so the approach does not affect the results. However, Kubernetes, which has more fix-inducing PRs in the earlier periods than more recent ones, is affected mainly through a dynamic method.

> *For risk-averse ($P_D = 25\%$) and risk-balanced ($P_D = 50\%$) recommendations, the dynamic method tends to provide the most value by recommending an effective period while for risk-tolerant recommendation ($P_D = 75\%$), none of the methods outperform others significantly.*

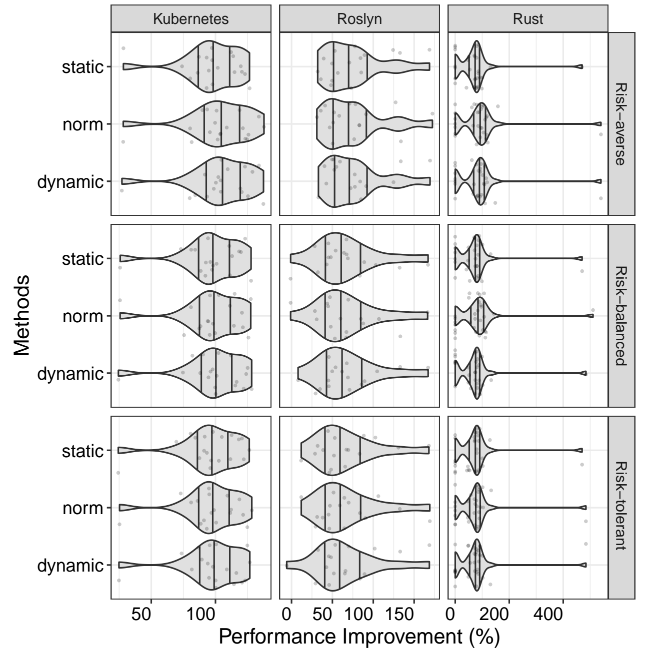


Figure 6: The distribution of performance improvement for CSR for different project over time.

VII. Practical Implications

Below, we summarize what we believe are the practical implications of greatest value for practitioners and researchers.

RQ1) Practitioners can use code review to balance files at risk of abandonment with the risk of fix-inducing changes. Our observations in RQ1 show that if the likelihood of a PR inducing a fix is not considered explicitly as a parameter in the recommenders' objective function, the recommended reviewers may lack the subject matter expertise to prevent future fixes, and in turn, increase the risk of merging fix-inducing PRs. The results also show an inherent trade-off between some of the evaluation measures, such as *files at risk of turnover*, and the risk of merging fix-inducing PRs. We propose *CSR* as a heuristic to assess the degree to which a (recommended) reviewer assignment mitigates the risk of fix-inducing changes.

RQ2,RQ3) *RAR* can be tuned according to the tolerance of the risk of fix-inducing changes without drastically impacting other properties of interest of the recommended reviewing assignment. Our observations in the first research question indicate that active effort should be made to mitigate the inherent trade-off between *CSR* and *files at risk of turnover*. To this end, *RAR* is proposed, which uses the P_D setting, as the threshold for the likelihood of a PR being fix inducing, to influence the suggested set of reviewers. The results of the second research question illustrate that *RAR* prevents other evaluation measures from being drastically impacted. The P_D setting can be tuned using a combination of our proposed dynamic method (see RQ3) and input from stakeholders about their tolerance of risk for fix-inducing changes. While project-specific characteristics (e.g., the incidence rate of fix-inducing changes) impact the sensitivity of the approach to the P_D setting, our dynamic approach can be scaled to apply well in different risk tolerance settings.

VIII. Threats to Validity

Below, we discuss the threats to the validity of our study.

Construct Validity. Our implementations may contain errors. To mitigate this risk, we augment an existing data set and vetted code from prior work [13] rather than producing our own from scratch. We share our implementation openly to enable the community to audit and build upon our code [54].

It is also possible that *CSR* does not truly reflect how well fix-inducing code changes are mitigated when assigning reviewers in reality. Because we cannot go back in time and change existing assignments to observe how well *CSR* truly performs, we evaluate its performance using historical data. We mitigate the chances of *CSR* being a poor refection of reality by basing it on proven measurements such as the fix-inducing likelihood and the xFactor [19]. Furthermore, the main idea behind *CSR*, that experts that have recently interacted with files in a code

Table II: The χ^2 and p-value results (two degrees of freedom) of the Friedman test applied to the RQ3 values.

Project \ Threshold	25%		50%		75%	
	Chi-Square	p-value	Chi-Square	p-value	Chi-Square	p-value
Roslyn	10.7	0.00473	12.8	0.00164	2.47	0.291
Rust	41.7	< 0.001	38.6	< 0.001	15.8	< 0.001
Kubernetes	23.1	< 0.001	16.5	< 0.001	20.6	< 0.001

Table III: Effect size and magnitude for Kendall's W (RQ3).

Project \ Threshold	25%		50%		75%	
	Effsize	magnitude	Effsize	magnitude	Effsize	magnitude
Roslyn	0.315	moderate	0.377	moderate	0.0727	small
Rust	0.695	large	0.643	large	0.264	small
Kubernetes	0.607	large	0.435	moderate	0.543	large

change reduce the likelihood of merging fix-inducing code changes, has been shown to reflect reality in prior studies [42].

To obtain data at a scale required for this study we must use automated tools. However, such approaches are not perfect and may induce errors in our results. To prevent any implementation errors, we use an existing tool (Commit Guru). We sampled the tool's output and manually verified the results. The resulting precision (i.e., 43.9% with confidence=95% and margin=±5%), aligns with prior works [36, 37]. While SZZ may introduce errors into our dataset, our results show that reviewer recommendations can still suggest the most relevant reviewer to reduce fix inducing changes, even when trained on noisy data. Future tools could be used to improve the performance of the approach.

Internal Validity. In this study, we consider the effect of assigning experts to review PRs that are potentially fix-inducing using measures, such as *CSR*. While assigning experts rather than novices to review PRs may change such measures, it does not guarantee that they will actually spot more defects. It is possible that other factors, that do not reflect a reduction in defects, are influencing the changes in *CSR*. However, prior studies have shown that experts increase the possibility of detecting fix-inducing PRs before merging, we therefore believe that similar outcomes should hold for our study. Further studies might help to clearly identify the impact of reviewers' experience and *CSR* on catching bugs during the PR process.

The defect prediction in Rust presents a low balanced accuracy. However, the other two projects yield similar results in different experiments, which we believe voids the possibility of the effect of this low accuracy in our experiments.

External Validity. While we apply eight different approaches to three systems, it is possible that our results might not generalize to other approaches or systems. We mitigate this threat by using a large number of approaches and systems with many files and a high volume of PRs. We target such systems because reviewer recommenders are most beneficial in big repositories with many developers. Through this selection we aim to make our findings applicable to the most pertinent systems.

IX. Conclusions

In this study, we set out to explore how using a code reviewer recommender to suggest reviewers can affect the risk of defect proneness. To this end, we introduce a new evaluation measure, *CSR*, and assess seven existing reviewer recommenders against this new measure. Three other measures previously used in the literature are also compared. The results show an inherent trade-off between *files at risk of turnover* and *CSR* – improvements to one measure often degrade the performance of the other. To balance this trade-off, an adjustable multi-objective code reviewer recommender, *RAR* is proposed. We analyze how *RAR* can be used to tune the recommendations with respect to the tolerance of the risks of fix-inducing PRs and files at risk of knowledge loss. Our findings suggest that:

- There is a trade-off between knowledge distribution and the likelihood of merged PRs being fix-inducing. However, this trade-off may be resolved by simultaneously optimizing recommendation strategies for both measures. This optimization, in turn, may lead to a decrease of other evaluation measures like core developers' workload.
- *RAR* can be tuned to balance the risk of knowledge loss and fix-inducing changes by tuning the P_D setting. However, identifying the optimal threshold setting requires an awareness of project-specific trends in the model estimates of the likelihood of fix-inducing changes. The results yield the average change of 12.48%, 0.93%, -19.39% and 80.00% over different quarters for evaluation measures *Expertise*, *Core workload*, *files at risk of turnover* and *CSR*, respectively. For the proposed measure, *CSR*, the average change is 73.80%-102.04% for various P_D settings.
- Project stakeholders can use *RAR* with a dynamic method for identifying effective range for the P_D setting. The dynamic method provides better performance for risk-averse and risk-balanced reviewer recommendation strategies while not hurting the risk-tolerant strategy's performance.

In future work, we plan to build an application on top of reviewer recommendation approaches such as *RAR* to study their effect of various in a live environment (e.g., Github). Moreover, we believe that investigating other measurements that estimate the risk of fix-inducing PRs could yield even more suitable candidates for *CSR*. Future work should also investigate the effects of using inaccurate bug detection methods on the results of reviewer recommenders such as *RAR*. To allow continued progress in this line of inquiry, we have made our code and dataset available online [54].

References

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

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

[3] M. di Biase, M. Bruntink, and A. Bacchelli, "A security perspective on code review: The case of chromium," in *16th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2016, pp. 21–30.

[4] 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*, 2016, pp. 1039–1050.

[5] H. Bodner, "10 reasons why code reviews make better code and better teams," May 2018. [Online]. Available: https://simpleprogrammer.com/why-code-reviews-make-better-code-teams/

[6] 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*, pp. 56–75, 2016.

[7] L. MacLeod, M. Greiler, M.-A. Storey, C. Bird, and J. Czerwonka, "Code reviewing in the trenches: Challenges and best practices," *IEEE Software*, pp. 34–42, 2017.

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

[9] P. Thongtanunam, R. G. Kula, A. E. C. Cruz, N. Yoshida, and H. Iida, "Improving code review effectiveness through reviewer recommendations," in *Proceedings of the 7th International Workshop on Cooperative and Human Aspects of Software Engineering*, 2014, pp. 119–122.

[10] M. Wessel, A. Serebrenik, I. Wiese, I. Steinmacher, and M. A. Gerosa, "Effects of adopting code review bots on pull requests to oss projects," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2020, pp. 1–11.

[11] K.-H. Yang, T.-L. Kuo, H.-M. Lee, and J.-M. Ho, "A reviewer recommendation system based on collaborative intelligence," in *2009 IEEE/WIC/ACM International Joint Conference on Web Intelligence and Intelligent Agent Technology*. IEEE, 2009, pp. 564–567.

[12] S. Rebai, A. Amich, S. Molaei, M. Kessentini, and R. Kazman, "Multi-objective code reviewer recommendations: balancing expertise, availability and collaborations," *Automated Software Engineering*, pp. 301–328, 2020.

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

[14] 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*, 2016, pp. 1028–1038.

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

[16] M. Torchiano, F. Ricca, and A. Marchetto, "Are web applications more defect-prone than desktop applications?" *International journal on software tools for technology transfer*, pp. 151–166, 2011.

[17] N. U. Eisty and J. C. Carver, "Developers perception of peer code review in research software development," *Empirical Software Engineering*, pp. 1–26, 2022.

[18] X. Ye, Y. Zheng, W. Aljedaani, and M. W. Mkaouer, "Recommending pull request reviewers based on code changes," *Soft Computing*, pp. 5619–5632, 2021.

[19] M. B. Zanjani, H. Kagdi, and C. Bird, "Automatically recommending peer reviewers in modern code review," *IEEE Transactions on Software Engineering*, pp. 530–543, 2015.

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

[21] V. Kovalenko, N. Tintarev, E. Pasynkov, C. Bird, and A. Bacchelli, "Does reviewer recommendation help developers?" *IEEE Transactions on Software Engineering*, pp. 710–731, 2018.

[22] I. X. Gauthier, M. Lamothe, G. Mussbacher, and S. McIntosh, "Is Historical Data an Appropriate Benchmark for Reviewer Recommendation Systems? A Case Study of the Gerrit Community," in *Proc. of the International Conference on Automated Software Engineering (ASE)*, 2021, p. To appear.

[23] J. Nam, "Survey on software defect prediction," *Department of Compter Science and Engineerning, The Hong Kong University of Science and Technology, Tech. Rep*, 2014.

[24] J. Xuan, Y. Hu, and H. Jiang, "Debt-prone bugs: technical debt in software maintenance," *arXiv preprint arXiv:1704.04766*, 2017.

[25] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" *ACM sigsoft software engineering notes*, pp. 1–5, 2005.

[26] E. C. Neto, D. A. Da Costa, and U. Kulesza, "The impact of refactoring changes on the szz algorithm: An empirical study," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2018, pp. 380–390.

[27] S. Davies, M. Roper, and M. Wood, “Comparing text-based and dependence-based approaches for determining the origins of bugs,” *Journal of Software: Evolution and Process*, pp. 107–139, 2014.

[28] C. Rosen, B. Grawi, and E. Shihab, “Commit guru: Analytics and risk prediction of software commits,” in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. ACM, 2015, pp. 966–969.

[29] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi, “A large-scale empirical study of just-in-time quality assurance,” *IEEE Trans. Softw. Eng.*, pp. 757–773, Jun. 2013.

[30] T. Hoang, H. K. Dam, Y. Kamei, D. Lo, and N. Ubayashi, “Deepjit: an end-to-end deep learning framework for just-in-time defect prediction,” in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 2019, pp. 34–45.

[31] S. McIntosh and Y. Kamei, “Are fix-inducing changes a moving target? a longitudinal case study of just-in-time defect prediction,” *IEEE Transactions on Software Engineering*, pp. 412–428, 2017.

[32] L. Pascarella, F. Palomba, and A. Bacchelli, “Fine-grained just-in-time defect prediction,” *Journal of Systems and Software*, pp. 22–36, 2019.

[33] M. K. Thota, F. H. Shajin, and P. Rajesh, “Survey on software defect prediction techniques,” *International Journal of Applied Science and Engineering*, pp. 331–344, 2020.

[34] anonymous, 2022, supporting online materials. [Online]. Available: https://zenodo.org/record/6727155#.YrYCtejMI2w

[35] E. Mirsaeedi and P. C. Peter, 2020, relationalGit. [Online]. Available: https://github.com/cesel/relationalgit

[36] S. Quach, M. Lamothe, Y. Kamei, and W. Shang, “An empirical study on the use of szz for identifying inducing changes of non-functional bugs,” *Empirical Software Engineering*, pp. 1–25, 2021.

[37] S. Quach, M. Lamothe, B. Adams, Y. Kamei, and W. Shang, “Evaluating the impact of falsely detected performance bug-inducing changes in jit models,” *Empirical Software Engineering*, pp. 1–32, 2021.

[38] G. Rodríguez-Pérez, G. Robles, A. Serebrenik, A. Zaidman, D. M. Germán, and J. M. Gonzalez-Barahona, “How bugs are born: a model to identify how bugs are introduced in software components,” *Empirical Software Engineering*, pp. 1294–1340, 2020.

[39] J. Ekanayake, J. Tappolet, H. C. Gall, and A. Bernstein, “Tracking concept drift of software projects using defect prediction quality,” in *International Working Conference on Mining Software Repositories*. IEEE, 2009, pp. 51–60.

[40] R. TAY, “Correlation, variance inflation and multicollinearity in regression model,” *Journal of the Eastern Asia Society for Transportation Studies*, pp. 2006–2015, 2017.

[41] 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*. IEEE, 2002, pp. 503–512.

[42] 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*, 2011, pp. 4–14.

[43] M. Zhou and A. Mockus, “What make long term contributors: Willingness and opportunity in oss community,” in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 518–528.

[44] 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)*. IEEE, 2016, pp. 1006–1016.

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

[46] 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*, 2016, pp. 99–110.

[47] M. M. Rahman, C. K. Roy, and J. A. Collins, “Correct: code reviewer recommendation in github based on cross-project and technology experience,” in *Proceedings of the 38th international conference on software engineering companion*, 2016, pp. 222–231.

[48] J. Kim and E. Lee, “Understanding review expertise of developers: A reviewer recommendation approach based on latent dirichlet allocation,” *Symmetry*, p. 114, 2018.

[49] M. Chouchen, A. Ouni, M. W. Mkaouer, R. G. Kula, and K. Inoue, “Recommending peer reviewers in modern code review: a multi-objective search-based approach,” in *Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion*, 2020, pp. 307–308.

[50] M. Friedman, “A comparison of alternative tests of significance for the problem of m rankings,” *The Annals of Mathematical Statistics*, pp. 86–92, 1940.

[51] M. G. Kendall *et al.*, “The advanced theory of statistics. vols. 1.” *The advanced theory of statistics. Vols. 1.*, 1948.

[52] W. Conover and R. L. Iman, “On some alternative procedures using ranks for the analysis of experimental designs,” *Communications in Statistics-Theory and Methods*, pp. 1349–1368, 1976.

[53] S. McIntosh and Y. Kamei, “Are Fix-Inducing Changes a Moving Target? A Longitudinal Case Study of Just-In-Time Defect Prediction,” *IEEE Transactions on Software Engineering*, p. 412–428, 2018.

[54] anonymous, 2022, replication package for this paper. [Online]. Available: https://github.com/software-rebels/RAR_Recommender