Characterizing the Prevalence, Distribution, and Duration of Stale Reviewer Recommendations

Farshad Kazemi, *Student Member, IEEE,* Maxime Lamothe, *Member, IEEE,*
and Shane McIntosh, *Member, IEEE*

Abstract—The appropriate assignment of reviewers is a key factor in determining the value that organizations can derive from code review. While inappropriate reviewer recommendations can hinder the benefits of the code review process, identifying these assignments is challenging. Stale reviewers, i.e., those who no longer contribute to the project, are one type of reviewer recommendation that is certainly inappropriate. Understanding and minimizing this type of recommendation can thus enhance the benefits of the code review process. While recent work demonstrates the existence of stale reviewers, to the best of our knowledge, attempts have yet to be made to characterize and mitigate them.

In this paper, we study the prevalence and potential effects. We then propose and assess a strategy to mitigate stale recommendations in existing code reviewer recommendation tools. By applying five code reviewer recommendation approaches (LearnRec, RetentionRec, cHRev, Sofia, and WLRRec) to three thriving open-source systems with 5,806 contributors, we observe that, on average, 12.59% of incorrect recommendations are stale due to developer turnover; however, fewer stale recommendations are made when the recency of contributions is considered by the recommendation objective function. We also investigate which reviewers appear in stale recommendations and observe that the top reviewers account for a considerable proportion of stale recommendations. For instance, in 15.31% of cases, the top-3 reviewers account for at least half of the stale recommendations. Finally, we study how long stale reviewers linger after the candidate leaves the project, observing that contributors who left the project 7.7 years ago are still suggested to review change sets. Based on our findings, we propose separating the reviewer contribution recency from the other factors that are used by the CRR objective function to filter out developers who have not contributed during a specified duration. By evaluating this strategy with different intervals, we assess the potential impact of this choice on the recommended reviewers. The proposed filter reduces the staleness of recommendations, i.e., the Staleness Reduction Ratio (SRR) improves between 21.44%–92.39%. Yet since the strategy may increase active reviewer workload, careful project-specific exploration of the impact of the cut-off setting is crucial.

Index Terms—Code Review, Code Reviewer Recommendation, Developer Turnover

1 Introduction

Code review is the practice of having peer developers inspect *change sets*, i.e., cohesive units of change to a codebase (e.g., fixes to defects, feature additions) [3]. Past work [11] has shown that reviewer feedback spans concerns about the impact that the change sets have on the *evolvability* of the codebase and the *functionality* of the system.

Although AI-based tools have been proposed to replace or augment human reviewers in code review [16, 24, 26], projects today still rely on humans to conduct reviews. The selection of reviewers with appropriate expertise directly impacts the quality [29, 43], duration [47], and outcome [10] of the code review process. This challenge is amplified in large organizations with swaths of developers and files, where it is difficult to identify who is knowledgeable about changed modules. For instance, Thongtanunam *et al.* [47] found that in several open-source communities, the code reviewer assignment problems delays the change set resolution by an average of 12 days.

- *Farshad Kazemi and Shane McIntosh are with the David R. Cheriton School of Computer Science, University of Waterloo, Canada. E-mail: {farshad.kazemi, shane.mcintosh}@uwaterloo.ca*
- *Maxime Lamothe is with the Department of Computer Engineering and Software Engineering, Polytechnique Montreal, Canada. E-mail: maxime.lamothe@polymtl.ca*

Manuscript received date; revised date.

Code Reviewer Recommendation (CRR) approaches have been developed to suggest suitable reviewers for a change set by ranking potential candidates. They evaluate contributors using their objective functions, taking into account factors such as experience [45], ownership [34], and developer interactions [38] and suggest the candidates with the highest scores to review the change set.

Traditionally, CRR approaches are evaluated by applying them to historical data, and comparing recommended reviewer lists to those who performed the review; however, recent studies call this practice into question. For example, Kovalenko *et al.* [22] found that the top recommendations are often known to developers. Thus, when recommendation approaches are *considered correct* (i.e., they recommend the reviewer who performed the review), their recommendations are often obvious choices and of limited value. Moreover, prior work [13] found that recommended reviewers who did not perform the review would often have been appropriate assignees. Thus, when CRR approaches are *considered incorrect*, the implications are often unclear. This raises a question: when can researchers and tool builders be certain that recommended reviewers are truly incorrect?

In this paper, we study stale recommendations—a class of recommended reviewers that are certainly incorrect. We define a stale recommendation as a recommended reviewer who has stopped contributing to the project under analysis. Since these contributors cannot perform the review, they

add no value to review recommendation lists. Indeed, the interviewees of Kovalenko *et al.* [22] point out that it is not uncommon for recommendation lists to include stale reviewers. Furthermore, Zhang *et al.* [51] found that 91.03% of the negative feedback they received from practitioners about the performance of a proprietary CRR system was about "irrelevant recommendations," where stale recommendations comprise 23.83% of this category. In contrast, other factors, such as a lack of prior participation in code review, only accounted for 8.97% of all the negative feedback. Aligned with prior work [22], the study further revealed that contributors frequently change their focus area or switch teams, potentially making them stale for their prior focus areas and development teams. These observations, coupled with the incontrovertible effect that stale recommendations have on the performance of CRR approaches (unlike other types of incorrect recommendations), highlight the considerable risks that stale recommendations pose to the quality of CRRs and provide an opportunity to better understand stale reviewer recommendations to mitigate the issue. Prior studies have explored the effect of stale reviewers through the lens of turnover-induced knowledge loss [9, 36, 39]; however, to the best of our knowledge, their characteristics and other potential effects on reviewer recommendation are yet to have been explored.

Using data from the Kubernetes, Rust, and Roslyn open-source projects, we study the prevalence of the stale recommendations that are produced by five reviewer recommendation approaches (LearnRec [32], RetentionRec [32], cHRev [50], Sofia [32], and WLRRec [1]). We find that on average, stale recommendations account for 12.02%, 8.33%, and 16.44% of incorrect recommendations that are produced by the cHRev [50], Sofia [32], and WLRRec [1] approaches per quarterly period, respectively, with medians of 10.28%, 6.63%, and 14.72%. Furthermore, when the number of recommendations to be produced is set to one, two, and three, CRR approaches produce at least one stale recommendation for up to 33.52%, 55.47%, and 69.18% of change sets, respectively, with medians of 6.34%, 15.01%, and 23.36%.

Since stale recommendations (1) represent a notable portion of incorrect recommendations, (2) can influence a considerable proportion of change sets, and (3) have clear implications (unlike other incorrect recommendations), we aim to characterize them and propose mitigation strategies. To do so, we address three Research Questions (RQs):

RQ1 **Are code reviewer recommendation approaches resilient to stale recommendations?**
Motivation: For a CRR approach, it is desirable to be resilient to developer turnover, while also suggesting suitable reviewers. However, in reality, CRR approaches are susceptible to changes in the set of active reviewers. This RQ aims to explore the extent to which stale reviewers are prevalent in the recommendations produced by the studied CRR approaches.
Results: CRR approaches that consider the recency of contributions tend to be more robust to stale recommendations. For instance, RetentionRec suggests no stale recommendations in the studied periods, while all LearnRec [32] recommendations are stale in 80.39% of all the studied quarters. We establish the worst and best performers as benchmarks and assess the effectiveness of each approach relative to these benchmarks. We introduce the *Recommender Adaptability Score* (RAS) to estimate an approach's capacity to handle the volatility of active contributors (larger RAS values indicate greater resilience). We observe that cHRev, Sofia, and WLRRec have median RAS values of 91.87%, 94.27%, and 84.66%, respectively, indicating that WLRRec is the least resilient, whereas Sofia is the most resilient.

RQ2 **Distribution: How is staleness distributed among recommendations?**
Motivation: If staleness is concentrated among a few reviewers, targeting these specific individuals would be more effective than strategies identifying many departed reviewers. Thus, to guide future work, we study how staleness is distributed across personnel.
Results: Staleness is highly concentrated for all of the studied recommendation approaches across the studied projects. Indeed, in 15.31% of periods over various evaluation settings, the top-3 reviewers account for at least half of the total observed staleness.

RQ3 **Duration: How long do stale recommendations linger on suggestion lists?**
Motivation: If most stale recommendations linger for a short period, recommendation approaches need to adapt to a dynamic list of reviewers who left the project. If, on the other hand, reviewers linger in stale recommendation lists for a long period, identifying a stable set of impactful candidates will have a larger effect. Thus, to guide future work, we study the extent to which stale recommendations linger.
Results: Stale recommendations can still be suggested up to 7.7 years after their departure, with a median time of 7 months and 21 days. While the lingering duration of top-3 reviewers increases over time, their proportion in stale recommendations reduces. Approaches that consider the recency of contributions (e.g., cHRev) reduce the number of stale recommendations, but are ineffective when other factors like experience dominate.

Our results suggest that a periodic pruning of the top stale reviewers may help existing approaches with staleness. In industrial settings, such a CRR system could integrate with personnel management systems to identify who left the company; however, team reorganization and internal movement of personnel may still present challenges [51]. To address those challenges, we propose a mitigation strategy to enhance CRR approaches with a separate time-bassed contribution recency filter to remove stale reviewers from the available developer pool, especially the top ones that have been lingering for a long time. This strategy diminishes stale recommendations by up to 92.16%, 92.39%, and 89.45% for cHRev, Sofia, and WLRRec, respectively. Future work could further improve this by improving the identification of the top reviewers and forecasting stale reviewers.

2 Related Work

Code Reviewer Recommendation (CRR). CRR approaches have been proposed to suggest reviewers for change sets [28], alleviating the burden of identifying a suitable reviewer [46], especially when potential reviewers are busy [8]. Delays in identifying appropriate reviewers may

cause change sets to be abandoned or delay their integration [42].

The fundamental behaviour of CRR approaches is similar. Once a new change set is submitted, candidate reviewers are ranked from most to least appropriate based on the approach objective function. Approaches vary in their focus, with some identifying reviewers who performed the task in the past [49] and others striving to improve knowledge distribution [32], reduce the reviewing workload for core team members [1, 15], and minimize the risk of defects [21].

For example, Jiang et al. [19] proposed CoreDevRec, which uses file paths as the input of their algorithm and recommends reviewers most familiar with the change set based on their previous reviews. Similarly, Thongtanunam et al. [47] proposed RevFinder, which leverages previous reviews and the similarity of file paths to recommend reviewers.

Approaches that recommend the familiar developers with the change set typically generate a high reviewing workload for these contributors, who are often the most senior team members. To combat this, recent work has treated reviewer recommendation as a multi-objective optimization problem. For example, Chouchen et al. [6] proposed WhoReview, which strives to balance the selection of expert reviewers with their workload.

The use of machine learning techniques have been explored in this domain. Strand et al. proposed Carrot—a context-aware CRR based on LightFM[1] algorithm [23]. Their industrial survey showed that over 50% found the tool helpful in assigning reviewers to change sets, but it did not affect the time interval between submission and first review.

Traditionally, CRR approaches have been evaluated based on their capacity to emulate historical data. However, recent studies argued that these assessments struggle to generate value in practical settings [13, 22] and are unlikely to provide a suitable ground truth against which CRRs should be benchmarked [8]. Thus, Mirsaeedi and Rigby [32] proposed a simulation-based evaluation strategy in which the impact of recommendations on objectives, such as the risk of turnover-induced knowledge loss and the overall expertise level of review task assignees, can be compared.

Several CRR approaches have attempted to enhance the quality of recommendations by incorporating contributor activity. cHRev [50], takes into account the recency of a person's contribution and the proportion of commits for files involved in a change set. Jiang et al. [20] also proposed a recommendation system for identifying suitable candidates to comment on a change set. Like cHRev, their approach incorporates a time-decaying factor for developers' past experiences, along with other metrics such as social interactions and prior involvement in similar change sets.

In this study, our objective is to assess the impact of stale recommendations on CRRs. To this end, we select five approaches for examination—one well-established traditional algorithm (cHRev [50]) and two state-of-the-art CRR algorithms that balance multiple objectives, such as the risk of turnover-induced knowledge loss with the reviewing burden imposed on the core team (Sofia [32]), and the expertise of reviewers with current task backlogs (WLRRec [1]). Additionally, to consider both optimal and suboptimal CRRs that focus on a single outcome, we include the LearnRec [32] and RetentionRec recommenders [32], which naïvely optimize for creating learning opportunities for reviewers or maximize the likelihood that the reviewer will continue to contribute to the project, respectively. Given their objective functions, we expect LearnRec and RetentionRec to generate high and low proportions of stale recommendations, respectively.

1. https://github.com/lyst/lightfm

Developer Turnover. Prior research on developer turnover [5, 12, 17] has investigated its impact on software projects. For example, Ton and Huckman [48] investigated the impact of knowledge-intensive employee turnover on operational performance. Mockus [33] studied the effect of developer turnover on software defects. They observed that the departure of developers was associated with an increase in software defect rate. Lin et al. [27] explored turnover from an individual-centric perspective, studying contributors of five projects and their correlation with activities and retention. Rigby et al. [40] used a Knowledge-at-Risk (KaR) measure to quantify how susceptible industrial and open-source systems are to turnover-induced knowledge loss. Nassif and Robillard [36] replicated and extended the concept to seven other projects and noticed a similar knowledge loss probability distribution among all projects.

In addition, research has explored the impact of developer turnover, proposing identification methods based on behavior. Avelino et al. [2] examined core developer turnover's effect on open-source projects, Miller et al. [31] studied reasons for open-source contributor disengagement, and Bao et al. [4] created a model to predict long-term GitHub contributors. Qui et al. [37] looked into projects' ability to attract new contributors, while Robillard [41] analyzed how employee leaves impact software companies, highlighting the significant disruptions caused by sudden and temporary departures.

Developer Recommendation Tasks. While this study explores the concept of staleness within the context of CRRs, it is a concern not only in CRRs, but also for software engineering tasks that involve developer recommendations, such as bug triaging [14] and task recommendation [25]. Similar to CRR, automated software engineering tasks usually involve applying information retrieval and (deep) learning techniques to incoming bug reports to recommend developers with relevant expertise [30, 35]. While providing accurate recommendations can considerably improve software maintenance, recommending inactive developers for such tasks may slow down the software development lifecycle—an issue that becomes particularly important for large open-source projects due to their decentralized nature [51].

3 Study Design

Figure 1 gives an overview of our experiment design for this study. This section outlines the dataset preparation (Section 3.1), the studied CRR approaches (Section 3.4), and the data processing procedures (Section 3.5).

3.1 Dataset Preparation

We conduct our research using a dataset derived from active open-source projects. The dataset is sourced from prior work [21]. While the dataset includes the information required for recommending reviewers, it lacks the reviewers'

TABLE 1
The details of the dataset used (built upon Kazemi et al. [21]) .

Name	Files	Reviewed Change Sets	Developers	Review Invitations
Roslyn	12,313	8,646	469	1,546
Rust	12,472	17,499	2,720	128
Kubernetes	12,792	32,400	2,617	26,164

Fig. 1. The simpli ed overall architecture of study data analysis.

invitations for the change sets, which we require for our study. Therefore, we augment the dataset by extracting the required reviewer data using the GitHub API. [2]

Studied Dataset: The dataset includes data from the Roslyn, Rust, and Kubernetes open-source projects. Rust is a popular high-level programming language with over 4.1K contributors. Roslyn provides tools for the analysis of C# and Visual Basic with 548 contributors and is backed by Microsoft.[3] Initially developed by Google, Kubernetes is now managed by over 3.5K contributors and aims to automate operational tasks for container management. The selection criteria for these project were to be active for over 4 years, have more than 10K change sets with review rate of more than 25 percent overall, and have more than 10K les. Further details on these projects can be found in Table 1.

3.2 Mining Contributors Lifecycle

This component is responsible for determining when a contributor joined and left a project. To this end, we query the contribution logs in the extracted dataset to identify each contributor's last contribution to the project, signifying when they ceased their involvement. To ensure accurate developer identi cation in the logs, after mining the data from GitHub we implement a cleaning and matching stage. This stage involves preprocessing the extracted user records, considering their names and emails, and removing special characters and diacritics. Moreover, we calculate the distance using the Damerau-Levenshtein algorithm [7] for string matching with a tolerance of 1 to accommodate minor user name and email variations. This helps with user identi- cation and creates comprehensive pro les for developers. We retain this information for our analysis (see Section 3.5).

3.3 Key Terms

To enhance the clarity of our methodology, we explicitly de ne the key terms employed throughout this report. These de nitions are used for identifying contributors and establishing the criteria for determining when a contributor is considered to have departed from a project.

Contribution: To identify potential reviewers from a project's developer pool, we consider those who had previous contributions to the project. Thus, we need to provide a crisp de nition of contribution in this paper. There are multiple ways in which individuals can contribute to the advancement of a project, including activities such as reviewing code and reporting bugs. Since end users can submit bug reports, we elect to concentrate on the contributions of code reviewers and change set authors in this study. Furthermore, since the recommendation approaches only consider developers for reviewing change sets, we only consider those who reviewed or authored a change set in the past.

Developer : Within the context of this study, a developer refers to individuals who have authored a portion of the code or participated in reviewing a Pull Request(PR). It is crucial to acknowledge that while they are quali ed to review subsequent changes to their own code, not all are actively involved in the review process. The term contributor is thus synonymous with developer as it aligns with the de nition of contribution established in this study.

Reviewer : This term refers to those reviewing a change set to ensure that new changes do not introduce bugs, ful ll the authors' intent, and adhere to the standards of the repository. Previous studies have shown that choosing the optimal reviewer impacts the quality of the review process [29, 43].

Stale Reviewer: The identi cation of contributors who have left a project has been explored in various studies using different thresholds from a contributor's latest contribution in periods of 30, 60, 180 days, and even a year [18, 27, 44]. A contributor is deemed stale at a given point in the project's history one day after they cease authoring or reviewing Pull Requests (PRs) and do not make any subsequent contributions in the project contribution history. We identify when recommendations become stale by rst compiling the contributions of developers from the project history and then determining their rst and last contribution. Similar to prior works [12, 27], we classify those who have contributed within the last six months of the project's available history as available developers to ensure all stale reviewers have been inactive for a minimum of six months. Subsequently, we compare the generated CRRs for the studied approaches against the activity lifecycle of developers to determine which recommendations are stale.

In summary, we consider all the developers who have made contributions before the change set submission as potential reviewers and then apply the CRR approach, using its objective function to prepare a set of reviewers and recommend that they review the change set. Among these recommended reviewers, some of them may not have been actively engaged in the project development, i.e., stale reviewers.

3.4 Generating Reviewer Recommendations

To generate CRRs that align with the state of the project at the time of proposing each change set, we conduct a simulation of the project's development over time. This simulation

2. https://docs.github.com/en/rest
3. https://devblogs.microsoft.com/visualstudio/introducing-the-microsoft-roslyn-ctp/

involved exclusively considering the data points available before the change set was proposed. For every change set, we identify the previous contributors and treat them as the pool of potential reviewers. Subsequently, we feed this data as the input of the CRR approach to reproduce the CRRs for the change set, which are then stored for further analysis.

Studied CRR Approaches: To investigate the quality of reviewer recommendations, we choose five CRR approaches with various recommendation styles. Below, we briefly describe each approach and its selection criteria. Since it is not feasible for one study to implement and evaluate all the available CRRs, we chose five approaches that cover different recommendation styles which are popular among the CRRs approaches [20, 45].

LearnRec [32] solely focuses on mitigating the risk of turnover-induced knowledge loss by promoting knowledge sharing among team members. It recommends contributors who are likely to learn the most from participating in the review of a change set by estimating the familiarity of the candidate with the modified files. LearnRec ranks candidates in ascending order based on the complement of a heuristic, which estimates how much the candidates know about the modified files (i.e., $1 - \text{ReviewerKnows}$). Even though LearnRec is singular in its optimization focus and would not reasonably be deployed in production, we include it as a benchmark to which other CRR approaches can be compared. Our hypothesis is that this approach will exhibit the lowest resilience to stale recommendations because individuals who are making a one-time contribution to a project are typically ranked as those who stand to learn the most from a review [21, 32].

RetentionRec [32] suggests only Long Term Contributors (LTC). While the former approach, LearnRec, is an extreme to mitigate the risk of turnover knowledge-loss, the developers who benefit the most from reviewing code may have little to offer in terms of feedback to benefit the authors of change sets. Moreover, they are highly likely to be one-time contributors [21, 32]. As an extreme countermeasure, the RetentionRec approach ranks candidates in descending order according to their frequency and consistency of contribution. The contribution ratio measures the proportion of contributions made by a developer during a period, while the consistency ratio measures the proportion of sub-periods in which the developer was actively contributing to the project. As developers become more consistent or active, the RetentionRec approach is more likely to suggest them as reviewers. Due to the characteristics of the objective function, we expect this approach to exhibit the highest resilience to stale recommendations; however, this tends to overburden the core team since they have the highest frequency and consistency of contributions.

cHRev [50] ranks potential reviewers for a change set based on their previous reviews and the recency of their contributions. To evaluate the suitability of developer D for reviewing file F, cHRev uses the xFactor measure, which is calculated as the sum of three terms: (1) the ratio of the number of review comments made by D on file F to the total number of review comments on F, (2) the ratio of the number of workdays that D commented on reviews of F to the total number of workdays for all reviewers of F, and (3) the inverse of the difference in days between the most recent day that D worked on F and the last date that F has changed, plus one. CHRev calculates the xFactor for files in the change set for potential reviewers and recommend those with the highest xFactor. In this paper, we consider cHRev as an example of a traditional CRR algorithm, which are approaches that seek to match review suggestions with historical review data [21].

Sofa [32] aims to balance multiple objectives, i.e., the knowledge distribution among active team members and the expertise of reviewers assigned to tasks. Suppose the number of knowledgeable developers in the project for any file in the change set R is N. When N is greater than a risk tolerance threshold ($N = 2$ in the original paper [32]), Sofa uses cHRev to rank recommendations by their expertise. Conversely, when N is below the risk tolerance threshold (i.e., there are fewer than N developers in the project that have knowledge of the file) Sofa uses the combination of RetentionRec and LearnRec to rank LTC candidates who can learn the most by reviewing R.

WLRRec (WorkLoad-aware Reviewer Recommendation) [1] takes into account the workload of potential reviewers when ranking candidates for a change set as well their social interactions. The idea is that if a reviewer is already very busy, they are less likely to agree to take on another task. When ranking candidates, WLRRec considers their past rate of accepted review invitations (review participation rate), the assigned reviews that candidates still have pending (remaining reviews), and the expertise and experience that candidates have with respect to the code under review (ownership and experience). We study WLRRec because it is a state-of-the-art approach that does not place importance on the recency of the candidate reviewer contributions. This attribute is desirable to help us comprehend a broad range of CRR characteristics in this study.

3.5 Data Processing

In this component, we address the research questions outlined in Section 1. Our investigation begins with a preliminary assessment of the prevalence of stale recommendations in CRR systems. If noticeable prevalence is observed, we will proceed to conduct a more in-depth analysis of the data collected in the previous stage.

We rely on historical data from Git repositories to produce CRRs. Further information on the history of each project can be found in Table 1. The historical data from each studied project is stratified into quarterly (three-month) intervals and CRR performance is evaluated for each interval. This approach aligns with previous studies on knowledge turnover [32, 36, 40], which also chose quarterly intervals. The rationale behind this choice is that it provides a balance: quarterly intervals are long enough to capture trends and patterns effectively, yet not so long that crucial details are obscured. Additionally, smaller time-frames have shown to be more susceptible to extreme events when compared to their respective means [36].

To confirm that code review was consistently carried out, we focus on contiguous periods where more than 80% of integrated change sets were reviewed. Figure 2 shows the quarterly review rates for each project.

Fig. 2. Quarterly review rates of Rust, Roslyn and Kubernetes projects.

Fig. 3. Share of stale recommendations over time for studied projects. Rows indicate variations for reviewer set sizes ranging from 1 to 3.

4 Preliminary Study

Prior studies have shown that developers complain about stale recommendations [22, 51]; however, the prevalence of and reasons for stale recommendations remain unexplored. Therefore, we conduct a preliminary investigation of stale recommendations in CRR approaches.

Approach. To gauge the potential impact of stale recommendations, we study the rate at which incorrect recommendations are stale. We apply the studied CRR approaches to produce reviewer recommendations at several points in time. Then, we identify incorrect recommendations, i.e., recommended reviewers who did not review the code. Next, we measure the prevalence of stale recommendations in incorrect ones, and the ratio of all change sets (i.e., PRs) that have at least one stale recommendation since they can potentially be impacted by stale recommendations.

Results. Stale recommendations frequently account for a considerable proportion of incorrect recommendations with an average of 12.59% of incorrect recommendations for non-naïve approaches [21], i.e., CRR approaches that optimize the recommendation for multiple objectives such as cHRev, Sofia, and WLRRec. Specifically, the average proportion of stale recommendations to the incorrect ones over reviewer set sizes of one to three for LearnRec, RetentionRec, cHRev, Sofia, and WLRRec is 97.13%, 0%, 12.03%, 8.33%, and 16.44%, respectively, with the median share of 100%, 0%, 10.28%, 6.63%, and 14.73% across all the studied periods. These proportions indicate that for all studied approaches, except for RetentionRec, stale recommendations account for a non-negligible proportion of incorrect recommendations. By exclusively suggesting contributors who exhibit a higher likelihood of remaining engaged in the project, RetentionRec surpasses other existing CRR methods in terms of mitigating the issue of stale recommendations. However, RetentionRec's superiority in this aspect comes at the cost of imposing a significant workload on core developers, rendering it impractical [21, 32].

The clarity of the impact of stale recommendations compared to other types of incorrect recommendations warrants a closer inspection. Prior research indicates that the reviewers of a change set are not necessarily the optimal choices [28], whereas those who are recommended but have not conducted the review often possess the necessary qualifications to conduct the review [13]. Hence, it is not straightforward to identify the truly incorrect recommendations among the produced reviewer recommendations. Stale recommendations, however, are unequivocally incorrect—they are unavailable to conduct the review. Furthermore, in specific periods (e.g. last periods of Kubernetes project), the proportion of stale recommendations becomes considerable, likely a factor that contributes to the negative feedback that was observed in previous studies [22, 51]. Therefore, mitigating stale recommendations directly enhances the performance of CRR approaches and improves the experience of teams that use them.

The size of the recommendation set has little influence on the proportion of stale recommendations, whereas the proportions vary substantially from one project to another. Figure 3 shows the proportion of incorrect recommendations that are stale over time for the three studied projects for reviewer set sizes from one to three. Each line shows the proportion of stale recommendations that are not influenced by the recommendation set size, whereas the project and CRR approach affect their proportion. While initial observations suggest reviewer set size does not influence the prevalence of stale reviewers, further analysis of potentially affected change sets indicates otherwise. Reviewer set size impacts the extent of affected change sets–a more important measure of stale reviewers' potential effect. Additionally, the figure indicates that project-specific factors, such as knowledge turnover rates, have a substantial impact on stale reviewer rates.

A considerable proportion of change sets has at least one stale recommendation. Our analysis reveals that up to 33.52%, 55.47%, and 69.18% of change sets include at least one stale recommendation when recommendation sets are of length 1, 2, and 3, respectively, with corresponding medians of 6.34%, 15.01%, and 23.36%. Figure 4 shows the proportion of change sets that include stale recommendations (Y-axis) over time (X-axis) across the studied projects (horizontal grid) with recommendation lists of length 3. This figure shows that the share of influenced change sets by stale recommendations tends to grow over time in all settings, except in the Rust project where WLRRec is applied. This difference is due to Rust's lack of review invitation records. While this might occur in real-world projects for various reasons, such as using alternative communication channels for review requests or allowing reviewers to self-select change

Fig. 4. Prevalence of potentially impacted change sets by stale recommendations for cHRev (left), So a (middle), and WLRRec (right) for each period (percentage).

sets to review, we consider it an interesting application of WLRRec to such projects rather than a threat to the validity of WLRRec results for Rust. Nevertheless, this lack of information causes a divergence between Rust and other projects when using WLRRec, which takes into account the accepted review invitation rate. Since the accepted invitation data is not available on Rust's GitHub, WLRRec does not perform as well. However, over time, other factors in the WLRRec algorithm, such as Reviewing Experience and Familiarity, compensate for this limitation. The results for reviewer set lengths of one to three can be found in our online appendix. [4]

Stale recommendations represent a considerable and persistent issue within CRR systems, as evidenced by the median percentage of change sets containing at least one stale recommendation, which ranges from 6.34%, 15.01%, to 23.36% for reviewer set lengths of one, two, and three, respectively. This trend not only underscores their prevalence but also suggests an increasing tendency over time, highlighting the shortcomings of widely-used CRR systems with this regard. A deeper investigation is warranted to better characterize their occurrences to guide the development of mitigation approaches.

5 RQ1: The Prevalence of Stale Reviewers in CRRs

In this section, we study the prevalence of stale recommendations that are produced by our studied approaches.

Approach. We evaluate the recommendations for studied approaches over the quarterly periods with recommendation lists of lengths one, two, and three. For each period, we compute the share of stale recommendations over all the incorrect recommendations. To assess the quality of studied approaches, we propose the Recommender Adaptability Score (RAS) measure, which gauges the approach's ability to respond to developer turnover and consider only active contributors:

$$RAS(CRR) = \frac{AUC(CRR_{LearnRec}) \quad AUC(CRR)}{AUC(CRR_{LearnRec}) \quad AUC(CRR_{RetentionRec})} \quad (1)$$

AUC refers to the Area Under the Curve that plots the proportion of stale recommendations against time (i.e., studied periods). We expect LearnRec and RetentionRec to produce the worst and best performance (i.e., the largest and smallest possible AUC), respectively. Therefore, the RAS calculates how much closer the CRR is to the optimal (best) performance than to the worst performance. The RAS ranges between 0–1; higher values indicate better performance.

4. https://zenodo.org/records/10971688

Results. Figure 5 shows the proportion of stale recommendations (Y axis) over time (X axis) across the studied projects (Horizontal grid) for reviewer set length of one. The results when the length of the recommendation list is set to two and three can be found in our online appendix. [4]

Stale recommendations account for up to 33.33% of all suggested reviewers with a median share of 8.3% of all of the recommendations. Figure 5 largely con rms previously reported developer complaints [22], i.e., that CRR approaches often suggest stale reviewers. CRR approaches, con guration settings, and periods have a considerable effect since Figure 5 also shows that the proportion of stale recommendations can drop to as low as 0.33%; however, even a minimal presence of stale recommendations—especially those involving stale reviewers who have long since departed from the project—can erode developers' trust in CRR systems, thereby affecting their usability. Additionally, reducing the incidences of stale recommendations, even if they constitute a small proportion of the recommendations at times, would certainly enhance the rate of correct recommendations, unlike other types of recommendations which may have ambiguous implications [8, 13].

Considering the recency of candidate contributions enhances the quality of CRRs.We nd approaches that consider the recency of contributions outperform WLRRec, a CRR approach that does not consider recency. Figure 5 indicates that RetentionRec is the best CRR approach with no stale recommendations, while LearnRec is the worst, providing stale recommendations in 80.39% of all the studied periods. The poor performance of LearnRec in recommending active contributors can be attributed to its prioritization of contributors with the greatest learning opportunity. This na ¨ve prioritization can lead to sizable knowledge loss [21, 32]. Meanwhile, RetentionRec prioritizes the most active contributors and is thus least prone to making stale recommendations; however, RetentionRec tends to overburden core developers (by design).

The performance of So a and cHRev—CRR approaches intended for actual deployment—falls in between these two extremes, with So a exhibiting a slight advantage over cHRev. We suspect that this is due to So a's use of RetentionRec to mitigate the risk of knowledge loss. Both So a and cHRev consider recent contributions of candidates. In contrast, WLRRec employs a combination of candidates' prior interactions, prior accepted review rate, experience, and workload to rank reviewer candidates without considering the recency of their contributions.

Table 2 shows the RAS scores of the studied approaches. LearnRec and RetentionRec are not included in the table, as they are the benchmarks used to compare other CRR

Fig. 5. The proportions of stale to all recommendations (y-axis). The period numbers are normalized, with zero representing the oldest period.

TABLE 2
Measured Recommender Adaptability Score (RAS) values for each setting. A higher RAS indicates better adaptability to developer turnover.

Project	Roslyn			Rust			Kubernetes		
Approach \ Reviewer set length	1	2	3	1	2	3	1	2	3
cHRev	0.9521	0.9448	0.9349	0.9356	0.9065	0.8902	0.9188	0.8945	0.8783
So a	0.9647	0.9595	0.9540	0.9632	0.9428	0.9313	0.9336	0.9135	0.8996
WLRRec	0.8280	0.8442	0.8466	0.8220	0.8041	0.8588	0.8757	0.8560	0.8676

Fig. 6. Developer expertise turnover rate for the studied periods over time. We consider the rst studied period to be zero in all projects.

approaches (worst and best performers, respectively). The median RAS scores of 0.9187, 0.9427, and 0.8466 for cHRev, So a, and WLRRec, respectively, suggest that they perform more similarly to RetentionRec (optimal) than LearnRec (worst). Moreover, the RAS obtained from Table 2 for cHRev, So a, and WLRRec exhibit a standard deviation of 0.0262, 0.0228, and 0.0181, respectively. This suggests that So a and cHRev are relatively more susceptible to variations in the reviewer set size and project, as compared to WLRRec.

Abrupt changes in developer expertise turnover led to a c layed impact on the staleness rate of CRRs, a trend observed ac all projects analyzed.To explore further, Figure 6 plots the expertise turnover rate of the studied projects over quarterly periods. The gure plots the ratio of previous contributions from developers who stopped contributing to the project during each period against the total prior contributions from all developers who were active by the end of the period. Period numbers are normalized to begin from zero to simplify comparisons across projects. For example, in the Roslyn project, there is a decline in the turnover rate between periods 2 and 3, followed by a steady increase until the end of the timeframe with small peaks at periods 4 and 6, as depicted in Figure 6. In this case, we observe a comparable trend in the proportion of stale recommendations, with a gentler slope for both segments in Figure 5. The gure also shows two small peaks at periods 5 and 8 with a delay from the expertise turnover peaks. The uctuation of the RAS scores for one CRR approach over different projects also con rms the resiliency of the CRR approaches against developer expertise turnover. For Kubernetes and Roslyn, the developer expertise turnover rate in Figure 6 shows an upward trend with a similar pattern of peaks occurring with 1 or 2 periods of delay in Figure 5. For the Rust project, however, while the upward slope for the share of stale recommendations is not as steep as the expertise turnover rate, we can still observe the impact of periods that have peak turnover ratio, such as periods 9 and 13, in Figure 5 with 1–2 periods of delay in periods 11–12 and 14–15. The WLRRec approach exhibits weaker adherence to the trend. We suspect this is because the Rust project does not keep any record of review invitations (i.e., developers invited to review change sets). This lack of data profoundly impacts the quality of recommendations generated by WLRRec since review invitations are part of its objective function.

Stale recommendations account for a considerable portion of the suggestions provided by CRR approaches, accounting for up to 33.33% of the recommendations with a median share of 8.3% of all of the recommendations that were produced. Although considering the recency of the candidate's contributions can partially mitigate the negative impact of stale recommendations, the performance of cHRev concerning stale recommendations suggests that solely considering this metric cannot eliminate this type of incorrect recommendation.

6 RQ2: The Distribution of Stale Recommendations Across Reviewers

In this section, we study the distribution of stale recommendations across the reviewers of the studied projects.
Approach. To study the distribution of stale recommendations, we calculate the proportion of stale recommendations accumulated by each reviewer quarterly. We aim to analyze and justify our observations to identify the most in uential factors contributing to stale recommendations.
Results. Figure 7 shows the proportion of stale recommendations accumulated by the top-3 most recommended reviewers to all stale recommendations for each quarterly

Fig. 7. The share of top-3 reviewers' recommendations of all stale recommendations for cHRev (leftmost bar), So a (middle bar), and WLRRec (right bar) for studied quarterly periods with reviewer set length of one.

Fig. 8. Change of top-N reviewers' share in stale recommendations with value of N when So a is applied to Roslyn (reviewer set lengths 1-3).

Fig. 9. The distribution of the duration of stale recommendations (in days) over quarterly periods for the studied projects. Only the rst nine periods are drawn.

period when the recommendation list length is set to one. We normalize study period numbers to begin from zero for easier comparisons. Results for recommendation list lengths two and three are in the online appendix. [4]

A small number of reviewers account for a substantial proportion of the stale recommendations. Figure 7 shows that in periods 2 and 3 of the Roslyn project, 100% of the stale recommendations are in reference to three reviewers. On the other hand, Figure 7 also shows that the proportion of stale recommendations accumulated by the top-3 reviewers can drop as low as 0.072 (i.e., in normalized period eight of the Kubernetes project). Moreover, in 15.31% of evaluated quarterly periods, over half of the stale recommendations are accumulated by the top-3 reviewers.

Figure 8 shows that a few reviewers constitute most of the stale recommendations in the periods of the Roslyn project when So a is applied. To enhance the quality of CRRs, these reviewers can be excluded from the candidate list. Similar trends were observed in other projects. [4]

The proportion of stale recommendations that the top reviewers accumulate tends to decrease as projects age. Figure 7 shows that the proportion of stale recommendations for the top-3 reviewers diminishes over time. For instance, when So a is applied, this proportion decreases from 75.00%, 64.29%, and 40.00% in period 0 to 55.26%, 38.89%, and 22.91% in the nal studied periods of Roslyn, Rust, and Kubernetes, respectively. Moreover, the periods in between show a decreasing trend.

CRR approaches frequently recommend a small number of reviewers who stopped contributing to the project based on their prior contributions. Although the proportion of such reviewers decreases over time as experienced contributors leave the project, removing them has the potential to considerably enhance the perceived quality of the CRRs.

7 RQ3: The Lingering effect of stale recommendations

In this section, we study how long contributors who left a project linger in recommendation lists.

Approach. To calculate the duration of a lingering stale recommendation, we measure the time between the last contribution and subsequent recommendations of the reviewer. We conduct experiments for studied projects, and assess the consistency and impact of each variable on the duration of lingering stale recommendations. Our ndings exclude the LearnRec and RetentionRec approaches since they are considered baseline approaches and are unlikely to be adopted in practice.

Results. There exist reviewers who persist in the recommendation list of CRR approaches for up to 7.7 years, with a median time of 7 months and 21 days. Figure 9 shows the distribution of the elapsed time (in days) between the departure of reviewers and their subsequent stale recommendations for all the recommendations over speci c quarterly periods. Among the studied projects, the upper bounds of the distributions tend to increase. This suggests that the evaluated CRR approaches do not effectively prune their candidate pool over time to eliminate stale recommendations, even though some consider the recency of their contributions. While some may argue that the responsibility of identifying

Fig. 10. The distribution of lingering duration for the top-3 reviewers over quarterly periods.

potential reviewers lies with those who are familiar with the current team, this task becomes increasingly challenging as teams grow, particularly in the context of open-source projects or when developers switch teams internally. For instance, we encountered cases in the Roslyn project where developers moved from Roslyn to Office 365 and other teams within Microsoft. These complexities and barriers indicate that CRR approaches could be used to effectively prune the pool of potential reviewers.

Indeed, contributors who have left a project may be recommended by CRR approaches long after they have left the project. For example, PR #65216 of the Kubernetes project, both cHRev and Soa recommend developer M when the reviewing set length is set to three. Developer M both joined and left the project in 2014. Nevertheless, upon submission of PR #65216 (more than 3.5 years later), M was recommended as a reviewer due to the extensive contributions to the file "pkg/util/iptables/iptables.go".

In another example (Roslyn PR#33501) cHRev recommends three reviewers, including developer H, who participated in Roslyn's code development from June 2014 to September 2018. In determining H's score, contributions and workdays each equally account for 30% of cHRev's score. The remaining 40% of the score weight is determined by the recency of the contribution. Meanwhile, Soa deems this change set at a high risk of turnover-induced knowledge loss and recommends candidates who are actively involved in the project's development. Thus, Soa does not make any stale recommendations for this change set. The WLRRec's recommendation for this PR is primarily influenced by previous interactions with the code's author and, as such, it only makes one stale recommendation when the length of the recommendation list is set to three.

The lingering attribute of stale recommendations differs among CRR approaches, influenced by their objective functions. CHRev shows the highest staleness (median staleness of 245 to 279 days) across the projects under scrutiny, with Soa performing slightly better (median staleness of 201 to 258 days). WLRRec presents the lowest median staleness in Roslyn and Kubernetes (160 and 203 days), but the highest in Rust (371 days). We suspect that these differences are explained by the focus of cHRev on maximizing expertise, which can overshadow the recency, leading to stale recommendations. Soa is similar to cHRev, but includes an adjustment for knowledge turnover risks, and uses RetentionRec to recommend active reviewers in high-risk changes. By considering social dynamics and reviewer request responsiveness, WLRRec provides a more balanced recommendation distribution in the Roslyn and Rust project. Its notably poorer performance in Rust (371 days median staleness) may be attributed to the lack of review request responsiveness records, which highlights the importance of this feature in its objective function as a countermeasure for lingering stale recommendations.

As projects age, CRR approaches tend to accrue a larger candidate list without pruning those who have left the project. This exacerbates the impact of top reviewers who left a project many periods ago. CRR approaches with high reliance on the expertise of the candidates (e.g., cHRev) are especially prone to this problem. As Figure 9 suggests, some of the reviewers may remain on the candidate list for a long time. These developers are most likely stuck in the candidate list due to their experience and prior contributions to important modules, which may degrade the performance of CRR approaches over time. Figure 10 shows this distribution for the top-3 reviewers over quarterly periods when the reviewer set length is set to one and confirms the increasing tendency of the longevity of the lingering duration over time. The results for reviewer set lengths two and three also follow the same trend.[4]

Sudden declines in the staleness of the top-3 reviewers are linked to increased release frequencies paired with a surge in new file additions, followed by high developer expertise turnover. This pattern is evident in periods 15 and 16 for Roslyn and 12 and 13 for Kubernetes, as shown in Figure 7. Our analysis indicates that these declines in the lingering days of the top-3 stale reviewers occur after periods marked by heightened release activities, new file additions, and developer turnover. New releases are often accompanied by a spike in bugs or feature requests that are related to newly added files, necessitating the involvement of developers who are familiar with those files. Suppose these developers have recently left the project. In that case, they are likely to be recommended, thereby reducing the lingering days of the top-3 stale reviewers by suggesting those who worked on those releases. For Roslyn, this effect can be observed with an increase in new files in PRs and a peak in release rates between periods 12 to 15. This is coupled with an increase in developer turnover that peaks during periods 14 and 16. Consequently, a noticeable drop for top-3 developer lingering days occurs between periods 15 to 16. In Kubernetes, the drop occurs between periods 12 and 13, triggered by a spike in developer turnover in period 11 and a high release rate compared to the median rate of all releases from periods 9 to 12, with an increase in new file additions in PRs from periods 8 to 11. The Rust project does not exhibit such concurrent events; hence, there is no similar decline in lingering days for its top-3 stale reviewers. Further details

and analyses are available in our online appendix. [4]

Comparing our latest findings with our previous research question shows that while the proportion of top-3 reviewers may decrease over time, their residual effect tends to increase. This highlights the need to remove the top reviewers who have left the project from candidate pools to address the lingering effect present in CRR approaches.

> CRR approaches make stale recommendations frequently, even years after contributors have left a project. While the percentage of the top reviewers of all the stale recommendations may decrease over time, their residual effect tends to increase. Regular pruning of the candidate pool could provide a reliable way to improve the perceived quality of reviewer recommendations.

8 Mitigation Plan

Based on our findings, we propose to incorporate the recency of developer contributions and filter out stale reviewers as a new stage for CRR approaches. This filter can ensure that the recency factor is not overshadowed by other variables, and can integrate with existing CRR approaches.

Approach. We propose a time-based filtering stage that excludes developers from the recommendation list if they have not contributed within a specified timeframe ($P_{ContributionGap}$). We assess the effect of different $P_{ContributionGap}$ durations—one year, six months, three months, and one month—on the performance of each approach to reveal the potential impact of applying the proposed factor and its effectiveness on studied approaches and projects. To this end, we apply the filter across these intervals and measure the metrics below:

Staleness Reduction Ratio (SRR) quantifies the improvement in recommendation staleness of a CRR approach upon integrating our time-based filter. SRR is calculated as the relative increase in the proportion of all the recommendations:

$$SRR = \frac{SSR_{\text{No Filter}} \quad SSR_{\text{Time-based Filter}}}{SSR_{\text{No Filter}}} \qquad (2)$$

$SSR_{\text{No Filter}}$ and $SSR_{\text{Time-based Filter}}$ represent the original staleness without, and with the filter applied, respectively.

Developers' Work Load Ratio (DWLR), inspired by prior work [32], evaluates the potential workload on non-stale recommended reviewers should they accept all recommendations. Workload is estimated using the reviewer's share of total review tasks.

F1-Score assesses the performance of the proposed filter in predicting stale recommendations. In this context, true positives are correctly replaced stale recommendations, false positives are non-stale recommendations mistakenly replaced, true negatives are non-stale recommendations accurately left unchanged, and false negatives are stale recommendations that were not identified and thus remained.

Present Reviewers Expertise (PRE) assesses the impact of the time-based filter on the expertise level of non-stale recommended reviewers. For each PR, after excluding stale reviewers, we measure the expertise of remaining recommended reviewers based on their prior contributions to the files involved in the PR. Expertise is defined following the formulation of Mirsaeedi and Rigby [32], i.e., focusing on the proportion of modified files which previously contributed to by the reviewer before the submission of the PR.

Results. SRR decreased substantially, with reductions ranging from 21.44% to 92.16% for cHRev, 22.42% to 92.39% for Sofia, and 21.62% to 89.45% for WLRRec. Remarkably, this filter also enabled LearnRec, a naïve baseline approach, to reduce stale recommendation rates by 19.93% to 92.48%. Thus, the time-based filter successfully achieves its primary goal of substantially lowering stale recommendations.

Reducing $P_{ContributionGap}$ enhances SRR but restricts the pool of available reviewers, thereby increasing the workload for active contributors. As anticipated, narrowing the interval for recent contributions can exclude active contributors, potentially increasing the workload for other developers. The median DWLR for the top-3 most recommended non-stale reviewers across studied periods increases with shorter $P_{ContributionGap}$ intervals—8.33%–13.33% for 1 year, 9.69%–15% for 6 months, 11.11%–16.67% for 3 months, and 13.41%–19.14% for 1 month, compared to 6.59%–12.29% without the filter. This trend highlights the trade-off in setting $P_{ContributionGap}$ values for the time-based filter.

The time-based filter considerably improves CRR approaches that overlook contribution recency or seek recommendation diversity, with F1-Scores ranging from 0.0254-0.1894 for cHRev, 0.0196-0.1560 for Sofia, and 0.2611-0.3587 for WLRRec. The time-based filter excels in contexts with higher stale recommendation rates. For cHRev and Sofia, the high precision shows that the time-based filter effectively identifies long standing stale reviewers, but struggles with recent stale ones, hindering their recall due to low rates of stale recommendations. In contrast, WLRRec suffers from a lower precision, but enjoys a higher recall, due to its higher rate of stale recommendations. LearnRec benefits across both metrics, indicating the filter's broad applicability for this baseline approach.

Applying the proposed filter to CRR approaches maintains or improves the expertise of non-stale recommended reviewers in our studied cases. The median PRE by cHRev and Sofia remains unchanged, while their mean expertise increases slightly, ranging from 0.73%–2.44% for cHRev and 0.70%–2.30% for Sofia. Conversely, WLRRec shows a notable median PRE improvement of 16.66%–40%, with mean expertise rising by 3.32%–10.64%. This enhancement results from the filter's exclusion of stale reviewers, thus increasing the likelihood of selecting reviewers with relevant expertise. Further analysis with the Wilcoxon signed-rank test shows significant changes in the distribution of PRE across settings, except for WLRRec over a 1-year interval in Roslyn for reviewer sets one and two, and Kubernetes for set three, highlighting significant impact of the time-based filter on reviewer selection. WLRRec's behavior over a 1-year interval is likely linked to stale recommendations from reviewers unfamiliar with modified files but who have recent interactions with the author and a high review invitation responsiveness. The recency of these interactions means the 1-year filter has little effect on these recommendations.

Detailed measurements for each experiment are provided in our online appendix for further reference. [4]

Filtering out inactive developers does not compromise recommendation quality; it actually enhances the expertise of the available recommended reviewers, with median improvements in PRE ranging from 0.73% to 10.64% across studied approaches. However, it may impose an additional workload on active reviewers. Thus, selecting an optimal cut-off interval is essential to minimize stale recommendations without overburdening developers by excluding too many potential reviewers.

9 Threats to Validity

Construct Validity. In this study, we explore stale recommendations in CRR approaches, de ning staleness as the interval between a developer's last contribution and subsequent recommendations. Because exact departure times are not usually documented, we should estimate the developer departure. We approximate the developer departure as one day after their last contribution if there has been no activity for at least 180 days. While this method may threaten the construct validity of our study, it is a widely accepted approximation in prior research [18, 27, 44].

Internal Validity. Contributors labelled as stale reviewers may be only taking a temporary hiatus. The last two quarterly periods are removed from the analysis to mitigate this threat, i.e., the shortest period of absence that will cause us to label a reviewer as stale is six months. We may also mislabel a recommendation as stale if it occurs shortly after the reviewer's last contribution, even though they remain active. However, we nd that such instances are rare. In the Roslyn project, stale recommendations for reviewers who departed within one day, one week, and one month of their last contribution comprised only 0.38%, 2.26%, and 7.94% of all stale reviewers, respectively. In Rust, the corresponding percentages were 0.32%, 2.17%, and 7.7%, and in Kubernetes, 0.37%, 3.13%, and 10.89%, indicating that these cases represent a small proportion of all stale recommendations.

External Validity. Although we study ve different CRR approaches, the outcome of our analysis may not generalize to all settings. However, our ndings highlight the degree to which current approaches are susceptible to stale recommendations and their characteristics for three studied projects of varying scales and domains. Nevertheless, replication of the study in other contexts may prove fruitful.

10 Conclusions and Lessons Learned

CRR approaches have been criticized for producing unactionable recommendations [22, 51]. Stale recommendations (i.e., recommended reviewers who no longer contribute to the project) are a concrete type of incorrect recommendations that hinder CRR performance and erode developer trust, especially when the recommended reviewer has long abandoned the project. Since stale reviewers can no longer effectively contribute to the project, they are truly incorrect recommendations, providing a unique opportunity for improvement. Therefore,in this paper, we examine three projects using ve different CRR approaches to understand their nature. Our investigation focuses on the prevalence of these recommendations (Section 5), the distribution of stale reviewers (Section 6), and the duration for which stale reviewers continue to appear in recommendations (Section 7). From our ndings, we derive the following actionable insights for both practitioners and developers of CRR tools:

The effect of stale recommendations is particularly substantial when experienced developers leave, as the proportion of stale recommendations among all CRRs is directly in uenced by the rate of expertise turnover (as shown in RQ1) . Consequently, for projects facing an exodus of experienced developers, we advise (1) employing CRR approaches that emphasize the recent contributions of reviewers, such as So a; and (2) whenever feasible, tuning the hyperparameters of CRR systems to weigh the contribution recency more prominently.

Our ndings suggest that the recency of developer activities should be incorporated as a stage before recommendations are generated. Doing so can mitigate the predominance of other factors, such as developer expertise, that may overshadow the recency of developer contributions. We observe a positive effect in approaches like cHRev and So a, where considering developer recency as a separate preparatory stage reduces the likelihood of stale recommendations by 19.93%-92.48%, depending on the cut-off parameter of the lter, and the project and CRR approach under scrutiny.

Identifying and replacing frequently recommended stale reviewers are crucial. Throughout our study, we noted instances where pinpointing the top-3 stale reviewers and substituting them with active developers reduces incidences of stale recommendations by up to 22.10% overall, with medians over the studied periods of 37.16%, 32.62%, 17.01% for the cHRev, So a, and WLRRec approaches, respectively. Therefore, identifying frequently recommended stale reviewers and removing them from the pool of available developers can already alleviate the general issue of staleness.

Implementing a threshold for the latest reviewer contribution can help mitigate the issue. Our analysis reveals that CRR approaches like cHRev, So a, WLRRec, and LearnRec allow stale reviewers to persist over extended periods of time. This trend worsens as projects mature, as evidenced by the increasing distribution of lingering time among recommendations (Figure 9). To counteract this, we propose implementing a maximum threshold for the latest reviewer contribution. Doing so will reduce the effect of recommendations drifting away from the active pool of developers. Our evaluation of various intervals indicates a trade-off between the developers' workload and the staleness of recommendation while having either a positive or negligible impact on the knowledge of non-stale reviewers. Unfortunately, it is not possible to recommend one best cut-off interval to mitigate staleness due to the contribution of various factors, such as knowledge turnover rate. However, our ndings can help orient practitioners to obtain more useful recommendations in their speci c circumstances. Using this mitigation strategy, one can strike a balance between the potential for sharing task knowledge between stale reviewers and active developers and their decreasing likelihood of responding as the time since their latest contribution grows.

References

[1] 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.

[2] G. Avelino, E. Constantinou, M. T. Valente, and A. Serebrenik, "On the abandonment and survival of open source projects: An empirical investigation," in 2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM). IEEE, 2019, pp. 1–12.

[3] 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.

[4] L. Bao, X. Xia, D. Lo, and G. C. Murphy, "A large scale study of long-time contributor prediction for github projects," IEEE Transactions on Software Engineering, vol. 47, no. 6, pp. 1277–1298, 2019.

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

[6] 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.

[7] F. J. Damerau, "A technique for computer detection and correction of spelling errors," Communications of the ACM, vol. 7, no. 3, pp. 171–176, 1964.

[8] E. Dogan, E. Tüzün, K. A. Tecimer, and H. A. Güvenir, "Investigating the validity of ground truth in code reviewer recommendation studies," in 2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM). IEEE, 2019, pp. 1–6.

[9] V. Etemadi, O. Bushehrian, and G. Robles, "Task assignment to counter the effect of developer turnover in software maintenance: A knowledge diffusion model," Information and Software Technology, vol. 143, p. 106786, 2022.

[10] Y. Fan, X. Xia, D. Lo, and S. Li, "Early prediction of merged code changes to prioritize reviewing tasks," Empirical Software Engineering, vol. 23, pp. 3346–3393, 2018.

[11] N. Fatima, S. Nazir, and S. Chuprat, "Understanding the impact of feedback on knowledge sharing in modern code review," in 2019 IEEE 6th International Conference on Engineering Technologies and Applied Sciences (ICETAS). IEEE, 2019, pp. 1–5.

[12] M. Foucault, M. Palyart, X. Blanc, G. C. Murphy, and J.-R. Falleri, "Impact of developer turnover on quality in open-source software," in Proceedings of the 2015 10th joint meeting on foundations of software engineering, 2015, pp. 829–841.

[13] 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 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 2021, pp. 30–41.

[14] C. Gupta, P. R. Inacio, and M. M. Freire, "Improving software maintenance with improved bug triaging," Journal of King Saud University-Computer and Information Sciences, vol. 34, no. 10, pp. 8757–8764, 2022.

[15] F. Hajari, S. Malmir, E. Mirsaeedi, and P. C. Rigby, "Factoring expertise, workload, and turnover into code review recommendation," IEEE Transactions on Software Engineering, 2024.

[16] Y. Hong, C. Tantithamthavorn, P. Thongtanunam, and A. Aleti, "Comment finder: a simpler, faster, more accurate code review comments recommendation," in Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2022, pp. 507–519.

[17] M. A. Huselid, "The impact of human resource management practices on turnover, productivity, and corporate financial performance," Academy of management journal, vol. 38, no. 3, pp. 635–672, 1995.

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

[19] J. Jiang, J.-H. He, and X.-Y. Chen, "Coredevrec: Automatic core member recommendation for contribution evaluation," Journal of Computer Science and Technology, vol. 30, no. 5, pp. 998–1016, 2015.

[20] J. Jiang, Y. Yang, J. He, X. Blanc, and L. Zhang, "Who should comment on this pull request? analyzing attributes for more accurate commenter recommendation in pull-based development," Information and Software Technology, vol. 84, pp. 48–62, 2017.

[21] F. Kazemi, M. Lamothe, and S. McIntosh, "Exploring the notion of risk in code reviewer recommendation," in 2022 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE, 2022, pp. 139–150.

[22] V. Kovalenko, N. Tintarev, E. Pasynkov, C. Bird, and A. Bacchelli, "Does reviewer recommendation help developers?" IEEE Transactions on Software Engineering, vol. 46, no. 7, pp. 710–731, 2020.

[23] M. Kula, "Metadata embeddings for user and item cold-start recommendations," arXiv preprint arXiv:1507.08439, 2015.

[24] L. Li, L. Yang, H. Jiang, J. Yan, T. Luo, Z. Hua, G. Liang, and C. Zuo, "Auger: automatically generating review comments with pre-training models," in Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2022, pp. 1009–1021.

[25] N. Li, W. Mo, and B. Shen, "Task recommendation with developer social network in software crowdsourcing," in 2016 23rd Asia-Pacific Software Engineering Conference (APSEC). IEEE, 2016, pp. 9–16.

[26] Z. Li, S. Lu, D. Guo, N. Duan, S. Jannu, G. Jenks, D. Majumder, J. Green, A. Svyatkovskiy, S. Fu et al., "Automating code review activities by large-scale pre-training," in Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2022, pp. 1035–1047.

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

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

[29] L. MacLeod, M. Greiler, M.-A. Storey, C. Bird, and J. Czerwonka, "Code reviewing in the trenches: Challenges and best practices," IEEE Software, vol. 35, no. 4, pp. 34–42, 2017.

[30] S. Mani, A. Sankaran, and R. Aralikatte, "Deeptriage: Exploring the effectiveness of deep learning for bug triaging," in Proceedings of the ACM India joint international conference on data science and management of data, 2019, pp. 171–179.

[31] C. Miller, D. G. Widder, C. Kästner, and B. Vasilescu, "Why do people give up flossing? a study of contributor disengagement in open source," in IFIP International Conference on Open Source Systems. Springer, 2019, pp. 116–129.

[32] E. Mirsaeedi and P. C. Rigby, "Mitigating turnover with