



PDF Download
3736405.pdf
26 December 2025
Total Citations: 0
Total Downloads: 277

Latest updates: <https://dl.acm.org/doi/10.1145/3736405>

RESEARCH-ARTICLE

Improving Code Reviewer Recommendation: Accuracy, Latency, Workload, and Bystanders

PETER CHRISTOPHER RIGBY

SETH ROGERS

SADRUDDIN SALEEM

PARTH SURESH

DANIEL SUSKIN

PATRICK RIGGS

[View all](#)

Published: 12 December 2025

Online AM: 27 May 2025

Accepted: 21 March 2025

Revised: 08 March 2025

Received: 28 December 2023

[Citation in BibTeX format](#)

Improving Code Reviewer Recommendation: Accuracy, Latency, Workload, and Bystanders

PETER C. RIGBY, Department of Computer Science and Software Engineering, Concordia University, Montreal, Quebec, Canada and Meta Platforms Inc, Menlo Park, California, USA

SETH ROGERS, SADRUDDIN SALEEM, PARTH SURESH, DANIEL SUSKIN, PATRICK RIGGS, CHANDRA MADDILA, and NACHIAPPAN NAGAPPAN, Meta Platforms Inc, Menlo Park, California, USA

AUDRIS MOCKUS, The University of Tennessee, Knoxville, Tennessee, USA and Meta Platforms Inc, Menlo Park, California, USA

Aim. The code review team at Meta is continuously improving the code review process. In this work, we report on three randomized controlled experimental trials to improve code reviewer recommendation.

Method. To evaluate the recommenders, we conduct three A/B tests which are a type of randomized controlled experimental trial. The unit is either the code diff (Meta's term for a pull-request) or all the diffs that an author creates during the experimental period. We set goal metrics, i.e., those we expect to improve, and guardrail metrics, those that we do not want to negatively impact, i.e., analogous to safety metrics in medical trials. We test the outcomes using a *t*-test, Wilcoxon test, or Fisher test depending on the type of data.

Expt. 1. We developed a new recommender, RevRecV2, based on features that had been successfully used in the literature and that could be calculated with low latency. In an A/B test on 82k diffs in Spring 2022, we found that the new recommender was more accurate and had lower latency. The new recommender did not impact the amount of time a diff was under review. The results allowed us to roll-out the recommender in the Summer 2022 to all of Meta.

Expt. 2. Reviewer workload is not evenly distributed, our goal was to reduce the workload of top reviewers. Based on the literature and using historical data, we conducted backtests to determine the best measure of reviewer workload. We then ran an A/B test on 28k diff authors in Winter 2023 on a workload-balanced recommender, RevRecWL. Our A/B test led to mixed results. When a low workload reviewer had reasonable expertise, authors selected them, however, the top recommended low workload reviewer was often not selected. There was no impact on our guardrail metrics of the amount of time to perform a review. This workload-balancing replaced the recommender from the first experiment as the recommender in production at Meta.

This work was done while Suskin was at Meta.

Authors' Contact Information: Peter C. Rigby (corresponding author), Department of Computer Science and Software Engineering, Concordia University, Montreal, Quebec, Canada and Meta Platforms Inc, Menlo Park, California, USA; e-mail: pcr@meta.com; Seth Rogers, Meta Platforms Inc, Menlo Park, California, USA; e-mail: sethrogers@meta.com; Sadruddin Saleem, Meta Platforms Inc, Menlo Park, California, USA; e-mail: sadruddin@meta.com; Parth Suresh, Meta Platforms Inc, Menlo Park, California, USA; e-mail: parthsuresh@meta.com; Daniel Suskin, Meta Platforms Inc, Menlo Park, California, USA; e-mail: suskin@meta.com; Patrick Riggs, Meta Platforms Inc, Menlo Park, California, USA; e-mail: riggspec@meta.com; Chandra Maddila, Meta Platforms Inc, Menlo Park, California, USA; e-mail: cmaddila@meta.com; Nachiappan Nagappan, Meta Platforms Inc, Menlo Park, California, USA; e-mail: nnachi@meta.com; Audris Mockus, The University of Tennessee, Knoxville, Tennessee, USA and Meta Platforms Inc, Menlo Park, California, USA; e-mail: audris@meta.com.



This work is licensed under Creative Commons Attribution-NoDerivatives International 4.0.

© 2025 Copyright held by the owner/author(s).

ACM 1557-7392/2025/12-ART24

<https://doi.org/10.1145/3736405>

Expt. 3. Engineers at Meta often select a team rather than an individual reviewer to review a diff. We suspected the bystander effect might be slowing down reviews of these diffs because no single individual was assigned the review. On diffs that only had a team assigned, we randomly selected one of the top three recommended reviewers to review the diff with BystanderRecRnd. We conducted an A/B test on 12.5k authors in Spring 2023 and found a large decrease in the amount of time it took for diffs to be reviewed. We did not find that reviewers rushed reviews. The results were strong enough to roll this recommender out to all diffs that only have a team assigned for review.

Implications. Aside from the direct findings from our work, our findings suggest there can be a discrepancy between historical backtesting and A/B test experimental findings, and that more A/B tests are necessary to test recommenders in production. Outcome measures beyond accuracy are important. This is especially true in understanding how recommenders change a reviewer's workload. We also see that the latency in displaying a recommendation can have a large impact on how often authors select recommendations making the reporting of latency an important metric for future work.

CCS Concepts: • **Software and its engineering**;

Additional Key Words and Phrases: Code Review, Recommender Systems, Experimentation, A/B Testing

ACM Reference format:

Peter C. Rigby, Seth Rogers, Sadruddin Saleem, Parth Suresh, Daniel Suskin, Patrick Riggs, Chandra Maddila, Nachiappan Nagappan, and Audris Mockus. 2025. Improving Code Review Recommendation: Accuracy, Latency, Workload, and Bystanders. *ACM Trans. Softw. Eng. Methodol.* 35, 1, Article 24 (December 2025), 23 pages.

<https://doi.org/10.1145/3736405>

1 Introduction

Code review is a critical activity in the software development lifecycle that ensures that at least one other experienced developer examines the code for defects and other issues. Code review has evolved from the inspection of massive completed artifacts [13] to the continuous code review of the difference between the changed files in the form of pull-requests, patchsets, and diffs [4, 34]. A key part of the success of code review is the reviewer who both learns from the change and also suggests improvements. Finding an optimal code reviewer in OSS and a large organization is a complex task; hence, substantial research has been devoted to code review recommenders which are tools that suggest who should review the code. At Meta, the reviewers or review teams are recommended to the author who selects the best person or team. The author can also assign reviewers who are outside of the recommendations. Much of that literature uses the assumption that the actual code reviewer was an optimal choice to evaluate the accuracy of the recommenders on historic data. Instead, we run A/B tests of newly developed recommenders to assess if the modifications made to recommenders have the desired effects, goals metrics, and no undesired side effects, guardrails metrics.

At Meta, we are constantly evolving our code review system. In this work, we describe a series of experimentally validated improvements that we made to our code reviewer recommenders. Our starting point was developed in 2018 and we refer to it as RevRecV1. To address latency and accuracy issues with RevRecV1, we developed RevRecV2 that is based on a literature review of successfully used features. Building on RevRecV2, we use measures of reviewer **workload (WL)** to spread review work and develop RevRecWL. Finally, we combat the bystander effect when a group is assigned to do a review with BystanderRecRnd. One of our major contributions is to experiment with our recommenders in production and to conduct randomized controlled experimental trials (A/B test) on each recommender across hundreds of thousands of reviews and 10's of thousands of

engineers. We provide an overview of the motivation for each recommender and briefly summarize the results.

Expt. 1. Accuracy and Latency Improvements with RevRecV2. On the code review team's feedback group, some engineers complained that they were being incorrectly recommended for reviews. When we investigated the current recommender, RevRecV1, we found it was purely based on who had last modified the lines in the current diff (i.e., blame lines), and it would suggest reviewers who had moved onto other projects because they had edited the files in the past. We reviewed the literature and developed a recommender that took the most promising features into account.

Result Summary: In an A/B test, our recommended reviewers were more likely to be the ones actually conducting the review with a 14.19 percentage point improvement over the previous model. The recommendations were also displayed more quickly with a 14 times reduction in latency at the 90th percentile. The recommendations were selected by authors 27% more often with the new model. We did not see any statistically significant regressions in our guardrail metrics including the review cycle time and the reviewer viewing time. In Summer 2022, RevRecV2 was rolled out to 100% of engineers.

Expt. 2. Reviewer WL Balancing with RevRecWL. The number of reviews that each engineer does is known to be skewed [1, 19, 35]. At Microsoft [3] and Ericsson [40], they implemented a recommender that weighted the expertise-based recommender score by the WL of the candidate reviewer. We replicate these works and experiment with novel WL measures, including the number of scheduled meetings to understand if WL balancing is effective.

Result Summary: When a reasonable candidate with lower WL was available, we saw a large reduction in WL. When an inappropriate candidate was recommended, simply because they had low WL, the author looked down the list for a more experienced developer, and we saw a drop in the top-ranked candidate being selected. We did not see any statistically significant change in our guardrails: review cycle time or latency. In Winter 2023, RevRecWL was rolled out to 100% of engineers.

Expt. 3. Reducing the Bystander Effect with BystanderRecRnd. The bystander effect occurs when nobody is explicitly assigned responsibility for a situation and no individual takes action or they are delayed in taking action [14]. Beyond noting that it may be an issue in code review [27], we did not find any research in the software engineering literature. At Meta, teams and groups of developers can be assigned a review without any explicit individual. To reduce the bystander effect, we use our recommenders to explicitly assign a reviewer.

Result Summary: At Meta, we randomly assigned one of the top three recommended reviewers to review the diff and found a drop of -11.6% in TimeInReview with no regressions in our guardrail metrics. In Spring 2023, we rolled BystanderRecRnd out to 100% of diffs that only had a team assigned to do the review.

The remainder of this article is structured as follows. In Section 2, we describe Meta and its code review process. In Section 3, we review the literature to identify the features that have been successfully used to recommend reviewers and the design of our recommenders. In Section 4, we describe our experimentation methodology. In Section 5, we update the recommender at Meta to include these features and experiment with a new model in production and report results for Expt. 1. In Section 6, we weight the recommender by the WL of the candidate reviewers in an attempt to balance WL and report results for Expt. 2. In Section 7, for Expt. 3, we reduce the impact of bystander effect when no individual is assigned a review by randomly assigning one of the recommended reviewers. In the final sections of the article, we discuss threats to validity in Section 8, position our work in the literature in Section 9, and conclude the article in Section 10.

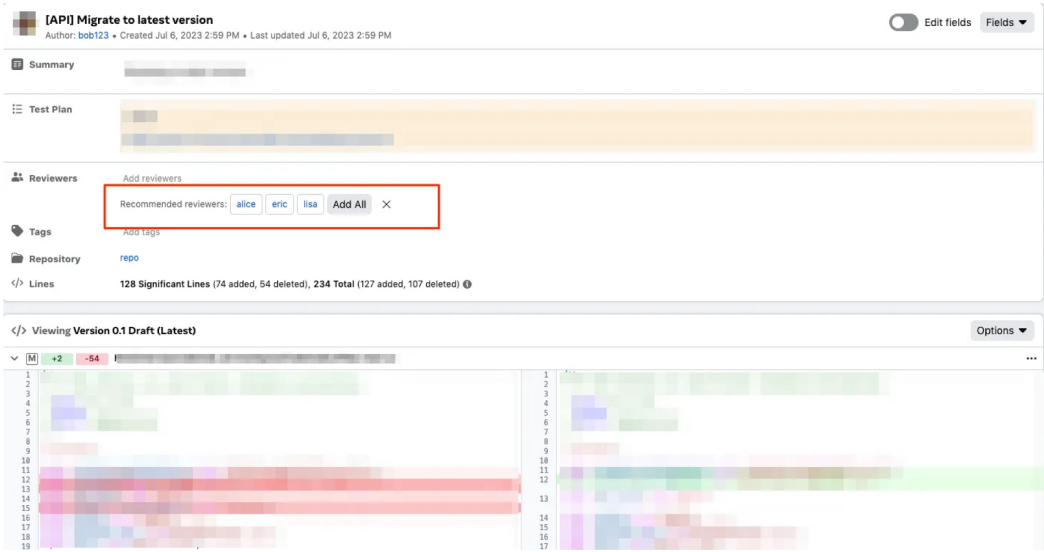


Fig. 1. A code change, i.e., diff, under review at Meta. The recommended candidates for the review are clickable by the author and ordered left to right with the top-ranked candidate on the left (see the red box in the figure). We have anonymized the diff and used mock names.

2 Background and Code Review Process

Meta is a large online services company that works across the spectrum in the communications, retail, and entertainment industries. Meta has tens of thousands of employees with offices spread across the globe (North America, Europe, Middle East, and Asia). Meta has its own dedicated infrastructure teams where the tools used are a mix of commercial and in-house developed systems. We study code reviews from the projects using Phabricator¹ which includes user facing social network products and virtual and augmented reality projects as well as software engineering infrastructure, such as calendar, tasks, and release engineering tooling.

2.1 Code Review Process

Code review at Meta follows a similar structure to the contemporary practices used at Google [38] and Microsoft [4], and we briefly summarize the practice here. Figure 1 provides an example of a diff review in Phabricator with portions redacted and using mock names. The author fills in a number of fields including the title and a detailed summary of the diff change. A test plan is provided that details which steps the author took to validate their changes, and provides reviewers a guide to test the change themselves. Line-by-line highlighting is used to show additions and removals, and inline comments are used to indicate any compilation or linting issues. The diff will also show the output of tests that were required to run based on the files included in the change.

To start a code review, the authors must select one or more reviewers and/or reviewer groups. The author can explicitly type in names, or select recommended reviewers. The latency is a key consideration for a recommender because slow recommendations will not appear in time to be selected by the author. Once selected, the reviewer or a group of reviewers is notified via e-mail or chat. The reviewers provide feedback, ask questions, and make suggestions, which should be addressed by the author. There may be multiple rounds of feedback and changes before a reviewer

¹Phabricator is an open source project available at <https://www.phacility.com/phabricator/>.

accepts/rejects the diff. Meta uses a single track/monorepo development model, so once the diff is accepted by at least one reviewer it will be integrated for further testing and ultimately “shipped” into production.

For this article, we focus on the reviewer recommendation experience. We see the recommended reviewers highlighted with a red box in Figure 1. We adapt the approaches behind the recommendations, but leave the interface consistent and unchanged.

2.2 Reviewer Recommenders at Meta

The reviewer recommender used in production at Meta before we began our work was developed in 2018. For this study, it will be our starting comparison point and we will refer to it as RevRecV1. RevRecV1 considered the following attributes: the blame lines, i.e., the person who has authored the most lines in a file is an expert, authorship frequency, i.e., the person who has made the most changes to a file is an expert, and review frequency, i.e., the number of times a reviewer has accepted the changes to a file. RevRecV1 is trained using a genetic algorithm, an ML approach popular in 2018. While the recommender has been effective and makes thousands of recommendations every day, the code review team identified a number of issues with it. First, many developers complain about incorrect recommendations on the internal feedback group. Second, blame is a computationally expensive command that can delay the loading of the code review page for a diff. For these reasons, the code review team decided to test alternative strategies for recommending reviewers to improve accuracy and reduce latency.

3 Literature, Features, and Recommender Design

We reviewed the literature to identify the reviewer recommendation approaches and model features that were most successful in existing recommenders. Like Microsoft [3, 45] and Ericsson [40], we cannot use features in production that would add latency to our recommender. We also found that very few recommenders that are studied in the literature report latency and even fewer are actually deployed into production. In this section, we update the literature review by Al-Zubaidi et al. [1] and use their categories: code ownership, reviewing experience, review participation rate, and familiarity with the author. Table 1 shows the features we selected for RevRecV2. In Section 3.2, we discuss how we weight RevRecV2 by different types of WL to create RevRecWL.

Code Ownership

The influence of code ownership on code quality has been extensively investigated in the literature [7, 15, 32, 41] with recommenders to identify expert developers for a particular part of code or task [29]. Researchers have used a wide range of granularity, from lines [16, 17, 32] to modules [7], to estimate ownership of developers. Studies on code review find that code owners are usually selected to review changes [4, 18, 38]. In this work, we quantify ownership as the number of files under review that a candidate reviewer has modified in past diffs.

Reviewing Experience

Thongtanunam et al. [41] recommend reviewers based on the files that an engineer has reviewed. Intuitively, engineers who have reviewed the files in the diff in the past, will be good candidate reviewers. Zanjani et al. [45] build upon this notion of reviewer expertise by adding the number of comments a candidate reviewer has made on the files under review. In this work, we count the number of comments as well as other reviewer actions such as accepting or rejecting the diff.

Familiarity with Patch Author

Moving beyond code ownership, reviewer recommenders have begun to consider relationships among the reviewer and the author [8]. The simplest metric is the number of times a reviewer has

Table 1. The Features for RevRecV2 and Their Importance in the Model Deployed in Section 5

Category	Feature	Description	Importance
Familiarity with Author	Time spent	The time the candidate spent in the diff review tool looking at the author's diffs in the past 90 days	33.89%
Familiarity with Author	Explicit time spent	The time the candidate spent in the diff review tool looking at the author's diffs when they were explicitly assigned as a reviewer in the past 90 days	6.69%
Review Participation Rate	Assigned reviews	Number of diffs the candidate was assigned as reviewer. At Meta reviewers are assigned to many diff either directly or as part of a team or review group	31.28%
Review Participation Rate	Resigned diffs	Number of diffs the candidate resigned from. When a reviewer does not feel qualified to review a diff they can resign from that review	0.00%
Reviewing Experience	Rejected diffs	Number of diffs the candidate rejected	11.56%
Reviewing Experience	Reviewer comments	Number of diffs the candidate commented or acted on as a reviewer	10.10%
Reviewing Experience	Total comments	Number of diffs the candidate commented on	1.57%
Code Ownership	Authored diffs	Number of diffs the candidate authored in the past which touched any of the files in the diff	4.91%

We only include features that can be calculated with low latency, we use the feature categories suggested by Al-Zubaidi et al. [1]. WL is only part of RevRecWL and is not included in this table.

commented on an author's diffs in the past. More sophisticated usages include social relationship graphs [44]. With 10's of thousands of developers, and thousands of diffs every day, social network graphs are not performant. In this work, we suggest a novel usage of how long a reviewer looks at an author's diffs: the time a reviewer spent looking at the diffs the author wrote in the code review tool. While Meta has sophisticated tracking of how long an engineer is looking at a Web page and uses this to capture how long a reviewer looks at a review, other projects can easily add this to the review tool. For example, the SmartBear code review tool has been tracking this information since 2006 [11] to identify reviewers that looked at the change for too short a period of time given the size of the diff, i.e., "rubberstamping." The web abounds with code to monitor time spent on a Web page.²

Review Participation Rate

On open source software projects, the number of reviews that a developer has been assigned and participated in is a strong predictor of whether they will perform a review [35, 37]. We measure this participation rate in two ways, the number of diffs that a reviewer was assigned and the number of diffs that a reviewer resigned from. These two measures capture how likely a reviewer is to participate in reviewing.

WL

Recent works have incorporated the reviewer WL into reviewer recommendation. Early works quantified the relationship between time waiting for a review and review WL. Baysal et al. [6] studied WebKit to understand the impact of metrics on review time. They found that when reviewers had longer queues of waiting reviews, reviews tended to take longer. They also found that more

²A JavaScript library that tracks time spent on a page <https://stackoverflow.com/questions/4667068/how-to-measure-a-time-spent-on-a-page>.

active reviewers had quicker response times than less active reviewers. Studying four open source projects, Ruangwan et al. [37] found prior participation and prior experience were the best predictor of whether a reviewer will respond to a review request. The familiarity of the author and reviewer had a minor impact on participation.

Microsoft weighted their code reviewer recommender, *chRev*, by the number of open reviews a candidate, *WhoDo*, had and found between a 14% and 21% improvement in the time taken for reviews depending on the project [3]. In contrast, Strand et al. [40] evaluate a WL-aware recommender at Ericsson. In a backtest, they found that a recommender aware of the file change history and the level of recent WL activity was able to make recommendations with an MRR of 0.37 or an average rank of 2.7. Despite these reasonable recommendations, after manually evaluating the performance of the recommender on 47 changes they determined that it did not reduce lead time or WL for code review and they decided to not roll-out the recommender.

3.1 Design and Feature Importance for RevRecV2

Prior works have mostly ranked reviewers based on a set of features and rules, e.g., [3, 28]. Since reviewer recommendation is a ranking problem, we select learn-to-rank approach [9], LambdaMART. A pairwise comparison of each reviewer is made and this ordering is then used to determine the final order. In our case we use multiple additive regression trees as the function to determine the ordering of reviewers. The LambdaMART is the boosted tree version of LambdaRank, which is based on RankNet. All have proven to be very successful algorithms for solving real-world ranking problems. An approximate description of how it works would take more space than this article and is presented in [9]. At Meta, there is infrastructural support for LambdaMART models at scale using measures collecting data from internal databases on code review and coding.

Feature Importance. In decision trees, feature importance is determined by how much each feature contributes to reducing the uncertainty in the target variable by the amount of reduction in the entropy that is achieved by splitting on a particular feature. Since LambdaMART uses boosted regression trees, the average reduction in entropy is calculated over the trees. The single-tree entropy is calculated by iterating over the non-leaf nodes of the tree and obtaining the weighted reduction in node entropy from the split at that node.

To train RevRecV2, we used the features shown in Table 1. The features were selected by the engineering team manually by using their domain knowledge of code review practices and their understanding of what metrics could be deployed in production. Specifically, they decided which features are most likely to contribute to the model's prediction and then selected a subset that would not slow down the recommender. Some features were included because the engineering team felt that they were inherently interesting. For example, although resigning from a diff has little predictive power, the engineers felt that a resignation demonstrated a clear example of a wrong reviewer suggestion. Since the number of features was small and prediction performance was stable, additional feature engineering was not conducted in this study especially since experiments showed substantial improvements over the baseline model. We trained the model on over 4.3 million instances of these features for reviewers. We evaluated the model on 1.1 million recommended candidates.

In Table 1, we show the feature importance across the 1.1 million diffs. The most important features that contribute to this accuracy improvement relate to the familiarity of the author with the review (time spent on the author's prior diffs, 33.89%) and the reviewer participation rate (assigned reviews 31.28%). In contrast, code and review ownership are still useful but much less important. The number of files in the current diff that the candidate reviewer has authored in the past accounts for only 4.91% (authored diffs). The number of diffs that a reviewer has acted on and rejected has an importance of 10.10% (reviewer comments) and 11.56% (rejected diffs).

The time spent when assigned a diff (explicit time spent) accounts for 6.69%. The total number of comments and number of resigned diffs have low importance with 1.57% and 0.00%, respectively. Our recommender clearly shows that being a more active reviewer and having reviewed the author's diff frequently in the past are much stronger predictors of who will actually perform a review than code ownership.

3.2 Design for RevRecWL

We design RevRecWL, which adds WL to RevRecV2. Our goal is to identify the top expert developer with the *lowest current WL*. To this end, we adapt the approach taken at Microsoft [3] that balances the WL of reviewers by weighting the score by the current WL of candidate reviewers for a diff review, r :

$$\text{Load}(r) = e^{\theta \cdot \text{Workload}}. \quad (1)$$

θ is a parameter between 0 and 1 to control the amount of load balancing.

At Microsoft [3], they found that initial feedback from developers indicated that reviewers from other projects complained that they were being incorrectly recommended. We made three compromises to convince the engineering team to run the experiment: (1) we set $\theta = 0.1$ which reduces the influence of WL favoring expertise, (2) we only re-rank the top five suggestions from RevRecV2, and (3) instead of assigning the top-ranked reviewer, we allow the author to select one of the top five reviewers. These latter two are important guarantees, first, people with low WL, but little experience will not be inadvertently added to the list eliminating the issue that Microsoft experienced. Second, the Top5 accuracy of RevRecWL = RevRecV2. Since the top five developers are displayed in Phabricator, this means in the most extreme case, the top developer from RevRecV2 will be displayed as the 5th developer, if they have high WL, but the same reviewers will be displayed only the order will change.

For a diff review, r , RevRecWL final scoring function is

$$\text{RevRecWL}(r) = \frac{\text{RevRecV2}(r)}{\text{Load}(r)}. \quad (2)$$

Instead of choosing a single measure of WL, we experiment with three types of WL weights, two are novel: the number of upcoming meetings in the next 7 days, the reviewer activity in the last 7 days [3], and time spent reviewing in the last 7 days. We calculate each over a 7-day period because developer patterns on a particular day tend to vary, but the entire work week tends to be more predictable [10]. In Section 6, we present our historical analysis to determine the best WL measure and our A/B test to determine if we can use RevRecWL in production.

3.3 Design for RecBystander

The bystander effect is a well-known psychological problem that when a group of people are present, it is less likely that any individual will take action [14]. At Meta, developers work in a monorepo, so there are times when the author does not know which engineer to assign the review to. In other cases, the author might not mind and anyone on a particular team can do the review. We decided to randomly assign one of the top three reviewers from RevRecV2 to the review. The results for BystanderRecRnd are presented in Section 7.

4 Experimental Method and Outcome Metrics

Industry experiments are rarely reported in software engineering literature with most empirical studies being observational, where past data are analyzed via a regression analysis in an attempt to suggest causal relationships. In observational studies, the treatment may be associated with

some unobserved property of the subjects or tasks that affects the outcome, i.e., allocation bias. Furthermore, the code recommender literature (e.g., [42]) assumes that the actual reviewer observed in the historic data is the gold standard (correct answer) for any proposed recommender. This ignores the possibility that the actual reviewer may not always represent the optimal choice and that the developer would not be affected by the recommendations they might be shown if a different recommender were to be used. This “backtesting” approach consists of calculating recommendations based on data predating the reviewer assignment and then checking if the actual reviewer is from among the TopN recommended reviewers.

A more reliable way to establish treatment effects is to run a randomized controlled experimental trial [21]. In software engineering, these trials are usually called A/B tests (see, for example [24], for the discussion of various types of experimentation in software engineering). Per definitions in [12]: an experiment is a study in which an intervention is deliberately introduced to observe its effects and in randomized experiments units are assigned to receive the treatment or an alternative condition by a random process. Industry experiments in software engineering are rare [23, 39] for a number of reasons. For example, experiments involving creation of systems or features in parallel (see, e.g., [2]) are prohibitively expensive. Involving human subjects also requires addressing numerous ethical and privacy concerns. Some of the early experiments in software engineering did, however, involve code reviews [31].

Specifically, we are running experiments with a randomized intervention where each unit is in an A or B condition. The randomization assures that observed (and unobserved) context will be evenly distributed under the A or B conditions. In our case, each experiment includes an older recommender, condition A or baseline condition, and a new recommender, condition B. Like in clinical trials of drugs, for example, a drug should have an intended effect, such as reducing inflammation. Similarly, the new recommender was designed with a goal in mind, e.g., WL balancing. The drug, even if it is highly effective at reducing inflammation, may have undesired side effects, such as higher mortality and such side effects are recorded in clinical trials. If the side effects are stronger or more frequent in the experiment than in the baseline condition, the trial may need to be stopped. Similarly, we do not want to negatively impact our code review process, so we have guardrail metrics like how long a reviewer spends looking at a diff because we do not want people to start rubberstamping and spending less time on review because of our change in recommendations. If we see regressions in these guardrail metrics the experiment can be stopped early or will not be deployed to the entire company.

A/B test requires that a treatment (specific recommender) be assigned to subjects (engineers in parallel design or individual diffs in crossover design) at random with another option being the baseline (the existing recommender). The main advantage of random assignment is the removal of the allocation bias, i.e., it ensures that the distribution of the potential confounders is independent of the treatment. Also, the researcher cannot assign treatments (and, thus, cannot steer certain reviewers to certain treatments). Since the user interface in each treatment is exactly the same, the engineer is also not aware if they are in the baseline or treatment conditions so they cannot intentionally act differently depending on the condition.

In contrast, if we were conducting an observational study, we would need to include numerous control variables, for example, the size and complexity of the diff (and of the underlying codebase), author and reviewer expertise, programming language, type of functionality being implemented, time of the year, day of the week, and numerous other factors that may affect the outcome and can (or in many contexts cannot) be measured. Using randomization helps to control for both observed and unobserved variables. Results from A/B tests are, therefore, relatively easy to analyze. Because of the random assignment, on average, the only differences between recommenders are due to recommenders.

While software engineering experiments are rare, the software industry commonly uses the so-called A/B testing [25] to describe the evaluation protocol for, typically, user interface modifications. For example, when a user goes to a Web page, they are randomly assigned either a regular version of a service or a modified version. Typically many different trials are ongoing at the same time so the randomization follows a factorial design to ensure that effects of each intervention (and, if possible, some of the interactions) can be estimated from the data. To implement A/B test for reviewer recommender, we ran a one-factor two-treatment controlled experiment as the two recommenders were the only distinction between the treatments. In many such experiments, the user may see a visual difference and recognize that they are receiving experimental treatment and act differently, but in this case, no such difference was observable; hence, the treatment was “blinded” from the subject.

To conduct our experiments, we used an internal tool designed to support A/B testing. The tool is programmed to do randomization where, in some experiments, we used parallel design where each developer gets only one recommender, and in others, we use a crossover design where each diff gets a randomly chosen recommender so that each developer gets assigned one recommender for some of the diffs and another for other diffs. The tool is then programmed to collect both goal and guardrail metrics and to run during a preset period of time. Once the experiment is complete, it presents an analysis page where average value and a CI (at $pval = 0.01$) are shown for each metric. The tool uses Welch’s t -test for unpaired samples to calculate the CI. If appropriate permissions are secured, researchers may access raw collected data for usually 90 days but up to 6 months after the experiment. After that the data are deleted only the summaries (means, sample sizes, and SD are available).

From the developers’ perspective, the review recommendation is a service that, when an engineer loads the diff page, gets automatically invoked and provides its recommendations. It is during that invocation that each diff may get a different recommender (or a developer keeps the same algorithm for subsequent diffs in case of parallel design). Once the page is loading, the developer may interact with the recommender (due to computational latency the recommender may not always appear immediately), or type in reviewer name directly. Hence, the record, in addition to the time of the action and the actual reviewer(s) chosen, also indicates if the recommender was interacted with (clicked on).

As in any experiments involving human subjects, the experimental design is reviewed to ensure the subjects will not be harmed and the tool used provides strong privacy protections. For example, the researcher does not have automatic access to the data that can be used to identify subjects, the data are provided by the tool in an aggregate form (means and SD for each condition), and UID (user identifiable data) is completely removed after 90 days.

4.1 Goal and Guardrail Outcomes Metrics

The vast majority of reviewer recommendation work does not release the recommender in production and only evaluates the accuracy on historical data. In contrast, the culture at Meta uses a sophisticated A/B test of every new feature and change. In Table 2, we show the recommenders that are being compared, the unit and size, and the time period of the experiment.

The team defines goal and guardrail metrics in advance of the experiment and then an A/B test is used to test the existing feature (the control or A condition) with the new feature (the test or B condition). The goal metrics measure the desired impact of the intervention. In contrast, guardrail metrics are the outcome metrics we do not want to negatively impact. In both cases, the null hypothesis or H_0 is that nothing will change under the intervention. In Table 3, we introduce the metrics that we use as outcomes in this article: TopN accuracy, TimeInReview, TimeSpent, Latency, Clicks on recommendations, and WL. Depending on the experiment, these outcomes will either be

Table 2. We Conduct Three Experiments

Experiment	Description	Expt. 1	Expt. 2	Expt. 3
Conditions: A vs. B	We conduct controlled experiments with a control or A condition vs. a test or B condition.	RevRecV1 vs. RevRecV2	RevRecV2 vs. RevRecWL	TeamOnly vs. BystanderRecRnd
Experimental Unit	The experimental unit is either the diff or the author. In either case, the unit is evenly divided across control and test conditions.	82k diffs	28k authors	12.5k authors
Experiment Date	We cannot report the exact timeframe that we ran the experiment, but we report the season in which each experiment was conducted.	Spring 2022	Winter 2023	Spring 2023

In each experiment, we have a control recommender vs. a test recommender. We report the experimental unit and when the experiment was run.

Table 3. For Each Experiment, We Describe Each Outcome Measure, Indicate Whether It Is a Goal Measure (in Bold) or a Guardrail Measure, and Describe the Statistical Test that Is Appropriate for the Type of Data

Outcome Measures	Description	Expt. 1	Expt. 2	Expt. 3	Stat Test
TopN Accuracy	How often does one of the recommended reviewers review the diff? We report Top1 and Top3 accuracy.	Goal	Guardrail	NA	Fisher
TimeInReview	The time when a diff is waiting for review or is being actively reviewed. It excludes the time when the author is reworking the code. We want to make sure that our recommender does not slow down reviewers.	Guardrail	Guardrail	Goal	<i>t</i> -test
TimeSpent	The amount of time reviewers spend examining each diff. This is time spent on the review page for a diff. We want to make sure that reviewers are not rushing through reviews and rubberstamping diffs.	Guardrail	Guardrail	Guardrail	<i>t</i> -test
Latency	The amount of time from when the diff is published until the recommendations are displayed. We monitor the 90th and 99th percentiles. The goal is to make sure that most diffs have recommendations produced by the time the diff page has loaded.	Goal	NA	NA	None
Clicks	The number of times that an author selects one of the recommended reviewers. We want to know how often the recommendations are selected.	Goal	Guardrail	NA	Fisher
WL	We measure the amount of time a candidate reviewer has spent looking at diffs in the last week. The distribution is skewed and our goal is to distribute review WL more evenly.	NA	Goal	NA	Wilcoxon signed-rank test

goal metrics or guardrails. For continuous data, we use a *t*-test, and for count data we use a Fisher test. Since WL is skewed we use a Wilcoxon signed-rank test. We often have multiple hypotheses, so we use a 99% CI. We also report the actual p-value when the result is not statistically significant. We report the percentage point change rather than effect size measurements like small, medium, and large because engineers prefer percentage points as they can more easily discuss the magnitude and importance of the change in the context of the problem they are solving.

Table 4. Expt. 1: The Change in Accuracy, Clicks, Latency, TimeInReview, and TimeSpent for RevRecV1 vs. RevRecV2 on 82k Diffs

Outcome Metric	Type	Expectation	Change	p-Value
Top1 Accuracy	Goal	Increase	+9.14 pp.	$p \ll 0.01$
Top3 Accuracy	Goal	Increase	+14.19 pp.	$p \ll 0.01$
Clicks	Goal	Increase	+27.4%	$p \ll 0.01$
Latency p90	Goal	Decrease	-14x	—
Latency p99	Goal	Decrease	-9x	—
TimeInReview	Guardrail	No increase	—	$p = 0.33$
TimeSpent	Guardrail	No decrease	—	$p = 0.58$

Latency measures were calculated in a dashboard, no statistical test was conducted, and the magnitude of the change is very large from an engineering perspective. Guardrail metrics are safety checks that we do not expect to see change.

5 Expt. 1: Accuracy and Latency

In Section 3, we examined the code reviewer recommendation literature. We use features from each category of measures we identified in the literature [1]: code ownership, reviewer experience, reviewer participation rate, and familiarity with the author. We also use a novel measure of how long a reviewer has viewed the author's changes in the last 90 days called TimeSpent.

We conducted an A/B test experiment to understand if the new model is effective in production. Our experimental unit is the diff under review. Diffs in the control group, A, continued to receive reviewer recommendations from RevRecV1, the existing model in production. Diffs in the test group, B, received reviewer recommendations from our new model. Our experiment ran in late Spring 2022, on 82k diff which were evenly and randomly split between the two conditions. We use a t -test on continuous data and a Fisher test on count data. Since we have multiple goal metrics, we use a 99% CI, i.e., $p < 0.01$.

Our goal metrics, the metrics we wanted to improve, were the Top1 and Top3 recommendation accuracy. TopN accuracy in this context is the fraction of times where at least one person belonging to the TopN predictions has either commented or acted, e.g., accepted, the diff. We also monitor when a recommendation is clicked, and we expect to see more accurate recommendations being clicked more often. Before we released our new recommender into production, we calculated the latency. In a dashboard, we tracked the slowest diffs, i.e., the latency at the 90th and 99th percentiles. Because latency was measured outside of the experimental framework, we were not able to calculate statistical significance on this measure.

Our guardrail metrics, the metrics we did not want to be negatively influenced, where the time diffs were in review (cycle time) and the TimeSpent that reviewers looked at each diff. We did not want to increase cycle time, and we did not want recommended engineers to rubber stamp diffs and spend less time looking at them.

Experimental Results: RevRecV1 vs. RevRecV2

For Expt. 1, our goal was to improve the accuracy and latency. We had RevRecV1 as the control and RevRecV2 as the test. The unit was 82k diffs in the Spring 2022. As we can see in Table 4, the experiment achieved our goals. We also found that the reduced latency improved the number of clicks on recommendations because they were displayed sooner. We did not want to see reviewers rushing reviews and did not expect to see a drop in how long reviews took.

Table 5. Improvement in Accuracy for RevRecV2

	Top-1	Top-3
RevRecV1	43.40	59.05
RevRecV2	52.54	73.24
Improvement	9.14 points	14.19 points

The improvement is statistically significant with $p \ll 0.01$.

Table 6. Improvement in Latency for RevRecV2

	Latency at p90	Latency at p99
RevRecV1 (control)	4.43 s	12.37 s
RevRecV2 (test)	300 ms	1.26 s
Improvement	14× improvement	9× improvement

We measure how long it takes for the recommender to make a recommendation in production, i.e., the latency. We are interested in the longest latency, so we report the 90th and 95th percentiles, i.e., p90 and p95.

Table 7. We Report the Number of Diffs that the Models Were Able to Create Recommendations for and How Often the Author Clicked on the Recommended Reviewer

	Diff with Recs	Clicked Recommendation
RevRecV1 (control)	38,247	6,420
RevRecV2 (test)	41,692	8,179
Improvement	9.00%	27.4%

The improvement in clicked recommendations is statistically significant with $p \ll 0.01$.

In Table 5, we see that the new model is much more *accurate* than RevRecV1. Looking at Top1 accuracy and Top3 accuracy, we see 9.14 and 14.19 percentage point improvements, respectively, over RevRecV1. These results were statistically significant with $p \ll 0.01$.

When comparing *latency* of the new model to RevRecV1, we see major wins in Table 6. The 90th percentile latency decreased from 4.43 s to 0.3 s (a 14× improvement), and p99 latency decreased from 12.37 s to 1.26 s (a 9× improvement). This reduces latency largely because our new model uses measures that are stored in a database and does not require blame information to be calculated on the fly. As noted before, we tracked this information in a dashboard outside our experimental framework and were unable to calculate statistical significance. However, from a developer's perspective, the latency reduction of 14 times is substantial.

We also see that diffs in the test condition have more *clicks* of the suggested reviewer feature, with a 27.3% increase in the number of diffs where a suggested reviewer was added using the recommender reviewer component on the diff review page, see Table 7.

We did not see any statistically significant change in how long diffs were in review. Importantly, we did not see any statistically significant change in how long reviewers examined the code under

review indicating that reviewers are not rubberstamping diffs. Based on these results, RevRecV2 has been rolled out to 100% of engineers at Meta since early Summer 2022.

In the A/B test, we saw a Top3 accuracy of 73%, a 14 point improvement over the previous model. We also saw a 14 times reduction in latency at p90. The recommendations were selected by authors 27% more often with the new model. These improvements were all statistically significant with $p \ll 0.01$. We did not see any statistically significant changes in the review cycle time or the reviewer viewing time. RevRecV2 was rolled out to 100% of engineers at Meta in early Summer 2022.

6 Expt. 2: Balancing Reviewer WL

Engineers have different skillsets which will always skew reviewer WL [36], it is still interesting to understand if a recommender that is aware of WL might distribute the load of reviewing and potentially drive down cycle time by suggesting the least busy expert. For example, Microsoft weighted their code reviewer recommender by a candidate's WL and reduced the time it takes for a review [3]. A similar experiment at Ericsson [40] balanced reviewer WL, but was unsuccessful and was rolled back. As a result, we build upon the Microsoft work and we take the score from RevRecV2 that is now in production and weight it by the WL of a candidate as described in the design and methods Section 3.2.

Our goal metric is to reduce the median amount of time spent reviewing the reviewer who made an action on a diff. We expect a decrease in accuracy in Top1 accuracy. However, because we only re-rank reviewers and still display all five reviewers, the same reviewers will be shown only if the order will have changed.

Our other guardrail metrics are the same as in the previous section. We do not want to see an increase in review cycle time. We want people to remain thorough in reviewing, so the time spent per diff should remain unchanged. We do not want to see fewer overall clicks on our recommendations.

6.1 Historical Backtest Results

To determine which type of WL is most effective at distributing reviews, we conduct a historical back test on 1.2 million historical diff reviews that were performed in early 2023. The backtest is followed by a production controlled experiment. We backtest three types of WL weights, the number of upcoming meetings in the next seven days, the reviewer activity in the last seven days [3], and time spent reviewing in the last seven days. We calculate each over a 7-day period because developer patterns on a particular day tend to vary, but the entire work week tends to be more predictable [10]. We also measure the impact on the Top1 and Top3 accuracy.

The top portion of Table 8 shows the results. We see that the Top1 accuracy decreases dramatically using WL balancing by around -20 percentage points compared to RevRecV2. The WL differences are less consistent, with upcoming meetings performing poorly, and review activity and time spent performing well with respective decreases of -57% and -73%. The Top3 accuracy shows similar patterns with time spent having the largest drop in WL.

Since we are only re-ordering the reviewers suggested by RevRecV2 and are still displaying the Top5 recommended reviewers, we decided to use the most aggressive weight metric, time spent reviewing in the last seven days, because it had the largest drop in WL. We implemented the algorithm in production and gated it for an A/B test.

Table 8. Backtest Comparison of Metrics and Results from WL Balancing in Production

Method	WL Metric	Top1 Accuracy	Top1 WL	Top3 Accuracy	Top3 WL
Backtest	Upcoming meetings	-21.66	-1.46	-12.66	1.55
Backtest	Reviewer activity	-20.61	-57.11	-13.40	-30.03
Backtest	Time spent	-21.37	-72.79	-20.89	-41.14
Expt. 2	Time spent	-4.80	-18.06	-5.34	-2.84

We report the percentage point decrease in accuracy and the percentage decrease in WL relative to results from RevRecV2.

Table 9. For Expt. 2 on WL Balancing, We Had RevRecV2 as the Control and RevRecWL as the Test

Outcome Metric	Type	Expectation	Change	p-Value
Workload Top 1	Goal	Decrease	-18.06 pp.	$p \ll 0.01$
Workload Top 3	Goal	Decrease	—	$p = 0.19$
Accuracy Top 1	Guardrail	Decrease must be > -21 pp.	-4.90 pp.	$p \ll 0.01$
Accuracy Top 3	Guardrail	Decrease must be > -20 pp.	-5.34 pp.	$p \ll 0.01$
Clicks Top 1	Guardrail	No decrease	-10 pp.	$p \ll 0.01$
All Clicks	Guardrail	No decrease	—	$p = 0.09$
TimeInReview	Guardrail	No increase	—	$p = 0.75$
TimeSpent	Guardrail	No decrease	—	$p = 0.85$

The unit was 28k diff authors. We conducted the experiment in Winter 2023. From our backtest in Table 8, we expected to see a drop in reviewer WL as reviews become more evenly distributed. From the backtest shown in Table 8, we also expected to see a tradeoff/drop in accuracy of up to -21 pp. The table shows that we did not see as much of a drop in WL, but we also did not see the same drop in accuracy. We did see an unexpected drop in Top 1 Clicks, however, the overall clicks remained constant.

6.2 Results for Expt. 2. RevRecWL in Production

We discussed our experimentation framework and metrics in background Section 4.1. To summarize, our goal metric is to reduce the WL of the Top recommended reviewers. Our backtest indicates that we will see a drop in the accuracy. *This reduction is the tradeoff for a more evenly distributed WL.* Specifically, we expect the accuracy to drop by no more than -20 percentage points. Our other guardrail metrics remain the same: we do not want to see fewer clicks on our recommendations, we do not want people to spend less time reviewing the diff, and we do not expect to see any change in how long reviews are waiting to be reviewed. The unit in the experiment was the author of a diff, so each author would either have RevRecV2 or RevRecWL. We had 14k authors in each condition for a total of 28k authors.

Results: The in production results are show in Table 9. We were surprised how different the results from the backtest were from the controlled production experiment, see the last row in Table 8. When the author selected the top recommendation, the accuracy decreased much less than in the backtest -5% vs. -21% decrease in accuracy and the WL only decreased -18% vs. -73% in the production vs. backtest. The WL drop at Top3 was not statistically significant.

Table 9 also shows the guardrail metric, number of clicks explains the discrepancy, and we saw a decrease in selection of the Top1 candidate by 30% of all diffs to 20%. Overall, however, the difference in the number of clicks did not change in a statistically significant manner. What this means is that when someone with a low WL was an inappropriate reviewer, the author simply

Table 10. Expt. 3: The Bystander Effect

Outcome Metric	Type	Expectation	Change	p-Value
TimeInReview	Goal	Decrease	-11.6	$p \ll 0.01$
TimeSpent	Guardrail	No decrease	—	$p = 0.37$

We compared the TeamOnly assignment control to BystanderRecRnd test. The experiment involved 12.5k authors evenly divided between the two conditions in Spring 2023. The bystander effect is real, and it was effectively minimized by randomly assigning one of the top three recommended reviewers.

ignored them and moved lower in the list of recommended reviewers. This result also reinforces how important it is to first model expert reviewers with RevRecV2 and then simply re-rank.

The remaining guardrail metrics time spent per diff and review cycle time were not changed in a statistically significant manner. This means that we were unable to replicate the Microsoft [3] reduction in the time pull-requests spent in review.

When a reasonable candidate with lower WL was available, we saw a large reduction in WL. When an inappropriate candidate was recommended, simply because they had low WL, the author looked down the list for a more experienced developer, and we saw a drop in the top-ranked candidate being selected. We did not see any statistically significant change in overall clicks or review cycle time. Since there was a reduction in WL for the Top1 reviewer, i.e., a more evenly distributed WL, we replaced RevRecV2 with RevRecWL and rolled it out to 100% of diffs at Meta.

7 The Bystander Effect

In psychology, the “bystander effect” occurs when a task is assigned to a group of people and leads to a lower likelihood of any individual acting than when an individual is explicitly assigned to a task [14]. At Meta, diff authors can assign teams and groups of developers rather than selecting individual reviewers. In some cases, an author might not mind which developer on a team reviews a diff. Furthermore, Meta uses a monorepo where developers often make changes that crosscut the codebase and are owned by another development team. The author might not know who the best reviewer is and will instead select a team. Selecting a reviewer group creates the risk of each member thinking, “someone else will probably review this.” We decided to leverage the wins from our reviewer recommendation investments last year to assign a single individual reviewer to diffs where only reviewer groups were assigned in an attempt to reduce the Bystander Effect.

The code review team decided to take a simple randomization strategy to overcome the bystander effect. BystanderRecRnd, is built directly upon RevRecV2, and we simply chose one of the top three reviewers at *random*. Our goal is that reviewers will feel a sense of ownership for diffs where they were assigned as a reviewer, and review it more quickly, thus reducing the TimeInReview metric. We do not want to see regressions in how long a developer views a diff. We do not track latency, clicks, or WL as they are unlikely to be impacted by this experiment. Our experimental unit was 12.5k authors that were evenly divided between the control and test conditions.

Results. In Table 10, we see that BystanderRecRnd was extremely successful and reduced TimeInReview by -11.6% with $p \ll 0.01$. There was no statistically significant change in the guardrail metric of the amount of time reviewers spent on each diff. We rolled BystanderRecRnd out to 100% of diffs at Meta where there was only a team or group assigned the review.

When a new feature or change is rolled out at Meta, we post an announcement in our feedback groups. In this case, we had one developer voice concern that explicitly assigning an individual to a diff will take time away from other diffs. To investigate this concern, we looked at all diffs during the time of study and saw no statistically significant change in the time in review per diff for diffs that already had an individual assigned to the diff.

The bystander effect occurs in code review when only a team is assigned to review a diff instead of an individual. At Meta, we randomly assigned one of the top three experts to review the diff and found a drop of -11.6% in `TimeInReview` ($p \ll 0.01$), with no statistically significant regressions in how long reviewers spent per review, i.e., `TimeSpent`. We rolled `BystanderRecRnd` out all diffs that only had a team assigned, and they will now also have an individual assigned.

8 Threats to Validity

8.1 Generalizability

Drawing general conclusions from empirical studies in software engineering is difficult because any process depends on a potentially large number of relevant context variables. The analyses in the present article were performed at Meta, and it is possible that results might not hold true elsewhere. We cannot release our data, even in an anonymized format, because it would violate legally binding employee privacy agreements. However, the software systems under study cover millions of lines of code and 10 s of thousands of developers who are both colocated and working at multiple locations across the world. We also cover a wide range of domains from user facing social network products and virtual and augmented reality projects to software engineering infrastructure, such as calendar, task, and release engineering tooling.

There are many reviewer recommenders available in the literature above, and instead of reimplementing these recommenders, we have used the feature categories suggested by Al-Zubaidi et al. [1], and included features from each categories that would not increase latency in our model. We have used those that can be extracted from Meta's history and those that can be calculated in a timely manner. It is possible that smaller organizations or smaller projects can use computationally expensive features, but unfortunately they do not work at our scale.

8.2 Construct Validity

The implicit assumption when, using historical data to assess TopN accuracy, is that the person who performed an action on a diff is a good reviewer. In our backtest, we saw that the TopN accuracy can be quite different from the person who actually conducts the review in a controlled experiment in production. Virtually all prior work has assumed that authors would accept the recommendations, our work sheds doubt on historical benchmarks as the gold standard of accuracy.

In our experiments, we used the outcome measures that are commonly used by the code review team at Meta. The two most commonly used measures are `TimeInReview` and `TimeSpent`. Most works capture the wall time from when the pull-request is published until it is merged or abandoned. In contrast, at Meta, we only capture the time that a diff is actively being reviewed or is waiting for review measures. Clicks and latency are uncontroversial measures that are simple to calculate.

Engineers have a variety of tasks and in this work we only backtested the number of meetings, reviewing activity, and the amount of time spent viewing diffs as measures of WL. In the end, we selected `TimeSpent`, but we look forward to seeing what type of WL is effective at other companies.

8.3 Internal Validity

Each of the recommenders presented in this article has been tested in a production experiment based on goal and guardrail metrics. However, each experiment was run at a different time, with RevRecV2 experimentation in the Spring 2022, RevRecWL in Winter 2023, and BystanderRecRnd in Spring 2023. We cannot report the exact time-frames, but report the number of diffs or diff authors so that the size of the sample can be evaluated by the reader. As part of engineering at Meta we track metrics across time and did not see any changes in accuracy, TimeInReview, and TimeSpent outside the experiments that we ran.

While the accuracy of the recommender is unaffected by collinearity among the features, the relative feature importance can be impacted. For example, we have the feature of the total amount of time a reviewer examined an author's diff and the time they did this when they were explicitly assigned to review the author's diffs. These features are definitely correlated and one is even a subset of the other. We kept both features because the engineering team thought that the features were both interesting, and as can be seen in the table, both add predictive power to the model. While the exact feature importance might vary depending on the project or environment, we based our feature selection on categories of features found in the literature and feel that they are a robust representation of features that are useful in reviewer recommendation.

8.4 Conclusion Validity

Conclusion Validity [43] is defined as the degree to which conclusions reached (e.g., about relationships between factors) are reasonable within the data collected. We use experiments that randomize treatments, thus eliminating researcher bias.

We do not explicitly control for factors including the size and complexity of the diff (and of the underlying codebase), author and reviewer expertise, programming language, type of functionality being implemented, time of the year, and day of the week. Instead we use randomization to help control for observed and unobserved variables, and we also have large sample sizes.

We report all experiments that were conducted and the results for all goal metrics (whether statistically significant or not). The p-values are adjusted for multiple comparisons in case more than one goal metric was used. Finally, since we only compare the distribution of response variables to a single predictor (experimental condition), we minimize the subjectivity due to researchers selecting one vs. another subset of predictors for reporting.

9 Literature and Discussion

We interleave our discussion with the literature showing how we advance the state-of-the-art reviewer recommenders.

9.1 General Reviewer Recommenders

The first reviewer recommendation system we are aware of was introduced by Balachandran [5]. They used authorship of the changed lines in a code review (using blame) to identify who had worked on that code before and suggested a ranked list of this set as potential reviewers. Similarly, Thongtanunam et al. [42] proposed REVFINDER, a reviewer recommender based on file locations. REVFINDER is able to recommend reviewers for new files based on reviews of files that have similar paths in the filesystem. The approach was evaluated on over 40,000 code reviews across three OSS projects and recalls a correct reviewer in the Top10 recommendations 79% of the time on average. Like RevRecV1 these approaches would suggest reviewers who had moved onto other projects because they had edited the files in the past.

There has been a lot of interest in consuming features beyond file history when building reviewer recommendation models. Rahman et al. [33] propose CORRECT an approach to recommend reviewers based on their history across all of GitHub as well as their experience with certain specialized technologies associated with a pull-request. Jiang et al. [22] examine the impact of various attributes of a pull-request on a reviewer recommender, including file similarity, PR text similarity, social relations, and “activeness” and time difference. They find that adding measures of activeness to prior models increases performance considerably. Zhang et al. [46] built a socio-technical graph to capture the social interactions among authors and reviewers. Further, they use Graph Convolution Networks to encode the social interaction information into high-dimensional space and formulate reviewer recommendation as a link prediction problem. We describe the features that we were able to include in our model in Table 1 building upon the categories suggested by Al-Zubaidi et al. [1].

9.2 Latency and Performance

Some of these recommenders use social network features and other features that require substantial processing. As noted by Microsoft [3, 45] and Ericsson [40], these cannot be used in production systems due to processing delay. We are unaware of any paper that reports results on latency of reviewer recommendations. Interestingly, when we reduced the latency from RevRecV1 to RevRecV2, we saw that lower latency resulted in authors seeing recommendations earlier before they had decided on who would do the review and many more clicks on recommendations. Future works should provide information on prediction latency.

9.3 Reviewer WL and Recommenders

Kovalenko et al. [26] found that expertise-based recommenders tend to suggest reviewers that are always obvious to authors 52% and 63% of the time at Microsoft and JetBrains, respectively. As a result, authors often clicked on because it is easier to click than type a reviewer’s name. Another interesting finding was that authors often found that outdated recommendations were made based on people who use to work on the files. Interviewed developers wanted more information about the current knowledge level of the candidate reviewers as well as how busy, i.e., what the WL of the reviewers are.

Microsoft weighted their code reviewer recommender, *chRev*, by the number of open reviews a candidate, *WhoDo*, had and found between a 14% and 21% improvement in the time taken for reviews depending on the project [3]. We were unable to replicate this result at Meta and found no statistically significant change in review cycle time. We did find a decrease in WL of selected reviewers. Unfortunately, Microsoft did not report the change in reviewer WL, so we are unable to make comparisons.

Al-Zubaidi et al. [1] treat reviewer recommendation as a multi-objective search problem where they maximize for the chance of participating in review and minimize the shannon entropy, i.e., skewness, of the people selected. Their recommender considers review WL, but has low precision ranging from 16% to 20% depending on the open source project. They do not use a measure of WL as an outcome, so it is unclear if their approach succeeds in spreading WL. They do not report on latency or performance, and it is unclear if this search-based approach could be used at large-scale companies.

Strand et al. [40] evaluate a WL-aware recommender at Ericsson. In a backtest, they found that a recommender aware of the file change history and the level of recent WL activity was able to make recommendations with an MRR of 0.37 or an average rank of 2.7. Despite these reasonable recommendations, after manually evaluating the performance of the recommender on 47 changes they determined that it did not reduce lead time or WL for code review and they decided to not

roll-out the recommender. Like Ericsson, we did not find a change in cycle time, however, we did find changes in the WL of the reviewer.

Hajari et al. [20] used the Gini coefficient to capture how unequal WL is and on open source projects found a near-perfect Pareto principle with the top 20% of reviewers doing 80% of the reviews. The authors simulate the change in Gini over multiple years of data and conclude that a WL-aware recommender similar to those used by Microsoft [3] can reduce WL. Our work threatens the conclusions of this article because WL-balanced recommendations can be ignored with the “normal” experts still doing most of the work. It is interesting future work to see if the Gini coefficient changes over the period of multiple years when a WL-balanced recommender is released in production.

Murphy-Hill et al. [30] examine WL at Google and discover a systemic imbalance in gender and code review selection. Authors manually select men more often than women as reviewers. Since recommenders are trained on prior data, the recommenders contain this bias. In our work, we do not explicitly factor gender into our recommender; however, we re-rank reviewers based on WL, so if women are assigned fewer reviewers, they will automatically be assigned more reviews as WL is balanced. We also use random in our bystander experiment, which Murphy-Hill et al. finds balances the gender of recommended reviewers.

10 Conclusion

At Meta, we make the following contributions related to accuracy and latency, WL, and the bystander effect:

- (1) RevRecV2: In the A/B test, we saw a Top3 accuracy of 73%, a 14 point improvement over the previous model. We also saw a 14 times reduction in latency at p90. The recommendations were selected by authors 27% more often with the new model. We did not see any statistically significant changes in the review cycle time or the reviewer viewing time. RevRecV2 has been rolled out to 100% of engineers at Meta.
- (2) RevRecWL: When a reasonable candidate with lower WL was available, we saw a large reduction in WL. When an inappropriate candidate was recommended, simply because they had low WL, the author looked down the list for a more experienced developer, and we saw a drop in the top-ranked candidate being selected. We did not see any statistically significant change in review cycle time or latency. As a result, we decided to only re-rank the top three reviewers and rolled RevRecWL out to 100% at Meta.
- (3) BystanderRecRnd: The bystander effect occurs in code review when only a team is assigned to review a diff instead of an individual. At Meta, we randomly assigned one of the top three experts to review the diff and found a drop of -11.6% in TimeInReview with no regressions in our guardrail metrics. We rolled BystanderRecRnd out to 100% of engineers.

Our contributions to the general field of software engineering are as follows:

- (1) We updated 2020 literature review by Al-Zubaidi et al. [1]. We developed a recommender based on measures from each of their categories. Interestingly, we found that reviewer relationships, i.e., how often a reviewer had reviewed an author in the past, were the strongest predictor. Expertise was less important.
- (2) We noted a large discrepancy between backtesting (the typical way code review tools are evaluated in research literature) and A/B test. While, obviously, not in all instances A/B test can be performed, future studies may discover ways to do backtesting in a way that is more reflective of what would be obtained via A/B test. In many companies code review

is implemented as a service, so A/B test is feasible given that privacy and other ethical considerations are adhered to.

- (3) We described a set of goal and guardrail metrics that can be used in code reviewer recommendation. We feel that TimeSpent is particularly important because it allows researchers to ensure that reviewers are not rubberstamping reviews without thoroughly considering the code. The metric can be easily captured using tools that capture the time spent on a Web page.
- (4) We are unaware of any works that reported on the impact of latency on reviewer recommendation. When our slowest 10% of recommendations (the 90th percentile recommendation time) went from 4.4 s to 300 ms, we saw a 27% increase in the number of recommendations that were clicked on. We suggest that future work should report how long it takes to generate a recommendation, if the 90th percentile is above 300 ms, then it is likely that the suggestion will load after the author has already selected other reviewers.
- (5) Although prior works attempted to balance reviewer WL, these works did not capture the change in WL, instead measuring other variables like TimeInReview. We suggested a simple measure of WL and showed that some degree of WL balancing can work. We hope that Microsoft, Ericsson, and other companies will report on the impact of WL balancing.
- (6) We are unaware of any works that quantify the impact of assigning a team rather than an individual. We were impressed how a simple random assignment of one of top recommended reviewers could effectively mitigate the bystander effect. This finding has implications beyond software firms and may impact how tasks are assigned generally in organizations. We look forward to future work on task assignments.

References

- [1] Wisam Haitham Abbood Al-Zubaidi, Patanamon Thongtanunam, Hoa Khanh Dam, Chakkrit Tantithamthavorn, and Aditya Ghose. 2020. 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 (PROMISE '20)*. ACM, New York, NY, 21–30. DOI: <https://doi.org/10.1145/3416508.3417115>
- [2] Bente C. D. Anda, Dag I. K. Sjøberg, and Audris Mockus. 2008. Variability and reproducibility in software engineering: A study of four companies that developed the same system. *IEEE Transactions on Software Engineering* 35, 3 (2008), 407–429.
- [3] Sumit Asthana, Rahul Kumar, Ranjita Bhagwan, Christian Bird, Chetan Bansal, Chandra Maddila, Sonu Mehta, and B. Ashok. 2019. Whodo: Automating reviewer suggestions at scale. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 937–945.
- [4] Alberto Bacchelli and Christian Bird. 2013. Expectations, outcomes, and challenges of modern code review. In *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 712–721.
- [5] Vipin Balachandran. 2013. Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation. In *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 931–940.
- [6] Olga Baysal, Oleksii Kononenko, Reid Holmes, and Michael W. Godfrey. 2013. The influence of non-technical factors on code review. In *Proceedings of the 2013 20th Working Conference on Reverse Engineering (WCRE)*, 122–131. DOI: <https://doi.org/10.1109/WCRE.2013.6671287>
- [7] Christian Bird, Nachiappan Nagappan, Brendan Murphy, Harald Gall, and Premkumar Devanbu. 2011. Don't touch my code!: Examining the effects of ownership on software quality. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*. ACM, 4–14.
- [8] Amiangshu Bosu, Jeffrey C. Carver, Christian Bird, Jonathan Orbeck, and Christopher Chockley. 2016. Process aspects and social dynamics of contemporary code review: Insights from open source development and industrial practice at Microsoft. *IEEE Transactions on Software Engineering* 43, 1 (2016), 56–75.
- [9] Christopher J. C. Burges. 2010. From ranknet to lambdarank to lambdamart: An overview. *Learning* 11, 23–581 (2010), 81.
- [10] Lawrence Chen, Peter C. Rigby, and Nachiappan Nagappan. 2022. Understanding why we cannot model how long a code review will take: An industrial case study. In *Proceedings of the 30th ACM Joint European Software Engineering*

- Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '22)*. ACM, New York, NY, 6 pages. DOI : <https://doi.org/10.1145/3540250.3558945>
- [11] J. Cohen. 2006. *Best Kept Secrets of Peer Code Review*. Smart Bear Inc., Austin, TX, 117.
 - [12] Thomas D. Cook, Donald Thomas Campbell, and William Shadish. 2002. *Experimental and Quasi-Experimental Designs for Generalized Causal Inference*, Vol. 1195, Houghton Mifflin Boston, MA.
 - [13] M. E. Fagan. 1976. Design and code inspections to reduce errors in program development. *IBM Systems Journal* 15, 3 (1976), 182–211.
 - [14] Peter Fischer, Joachim I. Krueger, Tobias Greitemeyer, Claudia Vogrincic, Andreas Kastenmüller, Dieter Frey, Moritz Heene, Magdalena Wicher, and Martina Kainbacher. 2011. The bystander-effect: A meta-analytic review on bystander intervention in dangerous and non-dangerous emergencies. *Psychological Bulletin* 137, 4 (2011), 517.
 - [15] Matthieu Foucault, Marc Palyart, Xavier Blanc, Gail C. Murphy, and Jean-Rémy Falleri. 2015. Impact of developer turnover on quality in open-source software. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 829–841.
 - [16] Thomas Fritz, Gail C. Murphy, and Emily Hill. 2007. Does a programmer’s activity indicate knowledge of code? In *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*. ACM, 341–350.
 - [17] Tudor Girba, Adrian Kuhn, Mauricio Seeberger, and Stéphane Ducasse. 2005. How developers drive software evolution. In *Proceedings of the 8th International Workshop on Principles of Software Evolution (IWPSE '05)*. IEEE, 113–122.
 - [18] LauraMacLeod, Michaela Greiler, Margaret-AnneStorey, Christian Bird, and Jacek Czerwonka. 2018. Code reviewing in the trenches: Understanding challenges, best practices and tool needs. *IEEE Software* 35, 4 (2018), 34–42. DOI : <https://doi.org/10.1109/MS.2017.265100500>
 - [19] Fahimeh Hajari. 2022. *Balance Expertise, Workload and Turnover into Code Review Recommendation*. Ph.D. Dissertation. Concordia University.
 - [20] Fahimeh Hajari, Samaneh Malmir, Ehsan Mirsaedi, and Peter C. Rigby. 2024. Factoring expertise, workload, and turnover into code review recommendation. *IEEE Transactions on Software Engineering* 50, 4 (2024), 884–899. DOI : <https://doi.org/10.1109/TSE.2024.3366753>
 - [21] Alejandro R. Jadad and Murray W[. Enkin. 2007. *Randomized Controlled Trials: Questions, Answers and Musings*. John Wiley & Sons.
 - [22] Jing Jiang, Yun Yang, Jiahuan He, Xavier Blanc, and Li Zhang. 2017. Who should comment on this pull request? Analyzing attributes for more accurate commenter recommendation in pull-based development. *Information and Software Technology* 84 (2017), 48–62.
 - [23] Natalia Juristo. 2016. Experiences conducting experiments in industry: The ESEIL FiDiPro project. In *Proceedings of the 4th International Workshop on Conducting Empirical Studies in Industry*, 1–3.
 - [24] Natalia Juristo and Ana M. Moreno. 2013. *Basics of Software Engineering Experimentation*. Springer Science & Business Media.
 - [25] Ron Kohavi, Diane Tang, and Ya Xu. 2020. *Trustworthy Online Controlled Experiments: A Practical Guide to A/B Testing*. Cambridge University Press.
 - [26] Vladimir Kovalenko, Nava Tintarev, Evgeny Pasyukov, Christian Bird, and Alberto Bacchelli. 2020. Does reviewer recommendation help developers? *IEEE Transactions on Software Engineering* 46, 7 (2020), 710–731. DOI : <https://doi.org/10.1109/TSE.2018.2868367>
 - [27] Andrew Meneely, Alberto C. Rodriguez Tejeda, Brian Spates, Shannon Trudeau, Danielle Neuberger, Katherine Whitlock, Christopher Ketant, and Kayla Davis. 2014. An empirical investigation of socio-technical code review metrics and security vulnerabilities. In *Proceedings of the 6th International Workshop on Social Software Engineering (SSE 2014)*. ACM, New York, NY, 37–44. DOI : <https://doi.org/10.1145/2661685.2661687>
 - [28] Ehsan Mirsaedi and Peter C. Rigby. 2020. Mitigating turnover with code review recommendation: Balancing expertise, workload, and knowledge distribution. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE '20)*. ACM, New York, NY, 1183–1195. DOI : <https://doi.org/10.1145/3377811.3380335>
 - [29] Audris Mockus and James D. Herbsleb. 2002. Expertise browser: A quantitative approach to identifying expertise. In *Proceedings of the 24th International Conference on Software Engineering (ICSE '02)*. IEEE, 503–512.
 - [30] Emerson Murphy-Hill, Jillian Dicker, Amber Horvath, Maggie Morrow Hodges, Carolyn D. Egelman, Laurie R. Weingart, Ciera Jaspán, Collin Green, and Nina Chen. 2023. Systemic gender inequities in who reviews code. *Proceedings of the ACM on Human-Computer Interaction* 7, CSCW1, Article 94 (Apr. 2023), 59 pages. DOI : <https://doi.org/10.1145/3579527>
 - [31] Adam Porter, Harvey Siy, Audris Mockus, and Lawrence Votta. 1998. Understanding the sources of variation in software inspections. *ACM Transactions on Software Engineering and Methodology* 7, 1 (1998), 41–79.
 - [32] Foyzur Rahman and Premkumar Devanbu. 2011. Ownership, experience and defects: A fine-grained study of authorship. In *Proceedings of the 33rd International Conference on Software Engineering*. ACM, 491–500.

- [33] Mohammad Masudur Rahman, Chanchal K. Roy, and Jason A. Collins. 2016. Correct: Code reviewer recommendation in github based on cross-project and technology experience. In *Proceedings of the 2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 222–231.
- [34] Peter C. Rigby and Christian Bird. 2013. Convergent contemporary software peer review practices. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 202–212.
- [35] Peter C. Rigby, Daniel M. German, Laura Cowen, and Margaret-Anne Storey. 2014. Peer review on open-source software projects: Parameters, statistical models, and theory. *ACM Transactions on Software Engineering and Methodology* 23, 4 (2014), 35.
- [36] Peter C. Rigby and Margaret-Anne Storey. 2011. Understanding broadcast based peer review on open source software projects. In *Proceedings of the 2011 33rd International Conference on Software Engineering (ICSE)*. IEEE, 541–550.
- [37] Shade Ruangwan, Patanamom Thongtanunam, Akinori Ihara, and Kenichi Matsumoto. 2019. The impact of human factors on the participation decision of reviewers in modern code review. *Empirical Software Engineering* 24 (2019), 973–1016.
- [38] Caitlin Sadowski, Emma Söderberg, Luke Church, Michal Sipko, and Alberto Bacchelli. 2018. Modern code review: A case study at Google. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*. ACM, 181–190.
- [39] Dag I. K. Sjøberg, Jo Erskine Hannay, Ove Hansen, Vigdis By Kampenes, Amela Karahasanovic, N.-K. Liborg, and Anette C. Rekdal. 2005. A survey of controlled experiments in software engineering. *IEEE Transactions on Software Engineering* 31, 9 (2005), 733–753.
- [40] Anton Strand, Markus Gunnarson, Ricardo Britto, and Muhmmad Usman. 2020. Using a context-aware approach to recommend code reviewers: Findings from an industrial case study. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP '20)*. ACM, New York, NY, 1–10. DOI : <https://doi.org/10.1145/3377813.3381365>
- [41] Patanamom Thongtanunam, Shane McIntosh, Ahmed E. Hassan, and Hajimu Iida. 2016. Revisiting code ownership and its relationship with software quality in the scope of modern code review. In *Proceedings of the 38th International Conference on Software Engineering*. ACM, 1039–1050.
- [42] Patanamom Thongtanunam, Chakkrit Tantithamthavorn, Raula Gaikovina Kula, Norihiro Yoshida, Hajimu Iida, and Ken-ichi Matsumoto. 2015. Who should review my code? A file location-based code-reviewer recommendation approach for modern code review. In *Proceedings of the 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 141–150.
- [43] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. 2012. *Experimentation in Software Engineering*, Vol. 236, Springer.
- [44] Yue Yu, Huaimin Wang, Gang Yin, and Tao Wang. 2016. Reviewer recommendation for pull-requests in GitHub: What can we learn from code review and bug assignment? *Information and Software Technology* 74 (2016), 204–218.
- [45] Motahareh Bahrami Zanjani, Huzefa Kagdi, and Christian Bird. 2016. Automatically recommending peer reviewers in modern code review. *IEEE Transactions on Software Engineering* 42, 6 (Jun. 2016), 530–543. DOI : <https://doi.org/10.1109/TSE.2015.2500238>
- [46] Jiyang Zhang, Chandra Maddila, Ram Bairi, Christian Bird, Ujjwal Raizada, Apoorva Agrawal, Yamini Jhavar, Kim Herzig, and Arie van Deursen. 2023. Using large-scale heterogeneous graph representation learning for code review recommendations at microsoft. In *Proceedings of the 45th International Conference on Software Engineering: Software Engineering in Practice*, Melbourne, Australia, ICSE-SEIP '23. IEEE Press, 162–172.

Received 28 December 2023; revised 8 March 2025; accepted 21 March 2025