Using Nudges to Accelerate Code Reviews at Scale

Qianhua Shan, David Sukhdeo, Qianying Huang, Seth Rogers, Lawrence Chen, Elise Paradis
Peter C. Rigby, Nachiappan Nagappan
Meta Platforms, Inc.
Menlo Park, New York, and Bellevue, USA
qshan@fb.com,davesukhdeo@fb.com,qyhuang@fb.com,sethrogers@fb.com,lchen@fb.com,pcr@fb.com,nnachi@fb.com

ABSTRACT

We describe a large-scale study to reduce the amount of time code review takes. Each quarter at Meta we survey developers. Combining sentiment data from a developer experience survey and telemetry data from our diff review tool, we address, "When does a diff review feel too slow?" From the sentiment data alone, we learn that 84.7% of developers are satisfied with the time their diffs spend in review. By enriching the survey results with telemetry for each respondent, we determined that sentiment is closely associated with the 75th percentile time in review for that respondent's diffs, *i.e.* those that take more than 24 hours.

To encourage developers to act on stale diffs that have had no action for 24 or more hours, we designed a `NudgeBot` to notify, *i.e.* nudge, reviewers. To determine who to nudge when a diff is stale, we created a model to rank the reviewers based on the probability that they will make a comment or perform some other action on a diff. This model outperformed models that looked at files the reviewer had modified in the past. Combining this information with prior author-review relationships, we achieved an MRR and AUC of .81 and .88, respectively.

To evaluate `NudgeBot` in production, we conducted an A/B cluster-randomized experiment on over 30k engineers. We observed substantial statistically significant decrease in both time in review (-6.8%, p=0.049) and time to first reviewer action (-9.9%, p=0.010). We also used guard metrics to ensure that most reviews were still done in fewer than 24 hours and that reviewers still spend the same amount of time looking at diffs, and saw *no* statistically significant change in these metrics. `NudgeBot` is now rolled out company wide and is used daily by thousands of engineers at Meta.

CCS CONCEPTS

• **Software and its engineering → Collaboration in software development**.

KEYWORDS

Code Review, Efficiency, Nudging

[*] Rigby is an associate professor at Concordia University in Montreal, QC, Canada.

ESEC/FSE '22, November 14–18, 2022, Singapore, Singapore

ACM ISBN 978-1-4503-9413-0/22/11…$15.00
https://doi.org/10.1145/3540250.3549104

ACM Reference Format:
Qianhua Shan, David Sukhdeo, Qianying Huang, Seth Rogers, Lawrence Chen, Elise Paradis, Peter C. Rigby, Nachiappan Nagappan. 2022. Using Nudges to Accelerate Code Reviews at Scale. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '22), November 14–18, 2022, Singapore, Singapore.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3540250.3549104

1 INTRODUCTION

Code review is an effective process for identifying defects and spreading knowledge. However, it is manual and time intensive and can be a bottleneck for rapid releases and continuous integration. In the early days of inspection, Fagan [15] reported that inspections took weeks or months to complete. In the late '90s, Porter *et al.* [27] simplified inspection and found that it could be reduced to around one week. In contrast, Rigby *et al.* [28, 30] showed that the contemporary lightweight review process used by open source projects had a review interval of approximately one day, which was confirmed to be similar to the review time at Microsoft [29] and Google [33]. The goal of this paper was to investigate whether and how time in review could be further reduced using nudges on slow code reviews [23].

Code review is an important part of the software development process at Meta. Every code change (called a "diff" and equivalent to a pull request) must be approved by a peer before being shipped. Meta's code review tool is called the `DiffTool` (see Figure 1 for an example of a diff under review). The code review team at Meta supports the `DiffTool` tool, among other code review tools, and aims to make code reviews as quick, productive, and delightful as possible.

Twice annually, the team leverages data from a company-wide "Developer Experience Survey" (or DevEx) to identify ways to improve the tool. Survey data from the second half of 2020 showed that while developers are overwhelmingly satisfied with code review in general, they are less satisfied with the speed with which their code is reviewed. In this paper, we address the following research questions related to perception of review speed, whom to nudge when diffs reviews are slow, and the `NudgeBot` tool and cluster-randomized A/B experiment.

RQ1, Survey and Telemetry: When does a diff review feel too slow?

Each half at Meta, the code review team examines data from a Developer Experience Survey to understand how well the tool is facilitating development and code review. While developers are overwhelmingly satisfied with code review in general, some are

dissatisfied with its speed. To dig deeper, we triangulated this survey's satisfaction data with quantitative time in review telemetry data.

We found that each developer's slowest 75th percentile of diff reviews strongly influences their level of frustration with diff review time. Based on this time in review distribution, Meta set a goal to reduce the number of stale diffs by nudging reviewers on diffs that had not received any interaction for 24 hours or more.

RQ2, `Who2Nudge` Model: Who should be nudged when a diff is stale?

To determine who should be nudged on a stale diff, we built a statistical model. Using features including size of change, how the diff was assigned, and the relationships among author and reviewer, the model predicts the diffs developers are most likely to review. We found that the most important features in the `Who2Nudge` Model are: 1) the amount of time the reviewer spent viewing the author's diffs in the past 90 days, 2) how the reviewer was assigned, and 3) the total number of assigned reviewers. The `Who2Nudge` Model has an AUC and MRR of 0.88 and 0.81, respectively.

RQ3, Experiment and Rollout: Does stale diff nudging work in practice?

We ran a cluster-randomized experiment with `NudgeBot` on code review, to determine if it reduces the number of stale diffs, *i.e.* 24 hours with no action, by nudging a subset of reviewers to take action. We observed statistically significant decreases 1) in review cycle time (-6.8%, p=0.049), 2) the proportion of diffs that take longer than three days to close (-11.89%, p=0.004), and 3) the time to first action on a diff (-9.9%, p=0.010).

Meta also ensures that guard metrics are not negatively impacted. We saw no statistically significant change ($p > 0.38$) in our guard metrics, including the percentage of diffs reviewed in fewer than 24 hours, *i.e.* diffs that are not nudged, and the total time eyeball time that reviewers spend looking at diffs in review, *i.e.* they do not rush their reviews.

The remainder of this paper is structured as follows. In Section 2, we introduce how code review is done at Meta, discuss the survey and available telemetry data, and provide an overview of experimentation at Meta. In Section 3, we use the survey and associated telemetry data to provide quantitative evidence of when diff should be considered stale. In Section 4, we train and evaluate a model of who should be nudged on a stale diff. In Section 5, we describe our cluster-randomized experiment and rollout of `NudgeBot` in production. In Section 6, we discuss threats to validity. In Sections 7 and 8, we discuss our work in the context of the literature, describe our contributions, and conclude the paper.

2 BACKGROUND AND DATA

Meta is a large online services company that works across the spectrum in the communications, retail, 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. The code review tool discussed here (`DiffTool`) is also widely used across the world released publicly by Meta as part of their open source efforts.

2.1 Code Review at Meta

Following the descriptions of code review at Microsoft [3] and Google [33], we provide a detailed description of code review at Meta. Code changes are called "diffs" and each diff goes through a review in the `DiffTool`. The `DiffTool` dashboard is a single-purpose surface used only for diff review, whereas most other surfaces are multi-purpose, with diffs comprising only a portion of the experience. For example, the Internal Home page (shown in Figure 2) displays various chats, meetings, and other action items a user may care about in addition to a widget containing up to five relevant diffs.

The author uploads the diff and, after checking it and potentially adding comments to guide reviewers, "publishes" the diff. The act of publishing a diff sets the diff's status to "needs review" whereupon it becomes visible to all reviewers. The author can assign individual reviewers and/or groups of reviewers, both before and after publishing. A recommender based on how often a potential reviewer has previously modified the files in the diff also suggests potentially competent reviewers. Numerous customized rules help assign developers to a review, *e.g.,* any diff that modifies file X automatically assigns reviewer A.

When reviewers are assigned to a new diff, they receive notifications across multiple surfaces, including email, desktop notifications, the Internal Home page, and the `DiffTool` dashboard. Sometimes the author may also directly message a reviewer with a link to the diff, or otherwise notify the reviewer outside of `DiffTool`-specific channels.

Once the reviewer has decided that they will review a particular diff, they open the "diff page," *i.e.* the web page with that specific diff's contents (See Figure 1). From there the reviewer can read the author's summary, the author's testing notes, and the code change itself. The reviewer can optionally add one or more comments to the diff, which are visible to everyone. The reviewer can also take one of several actions: accept, back-to-author, resign, or commandeer. The accept action marks the diff as ready to be shipped, *i.e.* landed on trunk. The "back-to-author" action is similar to the "reject" or "request changes" action in other code review tools, indicating to the author that changes or other follow-up is required before the diff can be shipped. The resign action removes the reviewer from the diff, and the commandeer action takes control of the diff, whereupon the user ceases to be a reviewer and becomes the author of the diff.

The diff author, meanwhile, has several actions at their disposal. If the author wishes to amend their code change, they can do so and update the diff accordingly. The author may also comment on their own diff, for instance to explain an update to the code change or to address reviewer feedback. If the diff has been accepted, the author can ship it to production. If the diff is in review, the author can mark their diff "changes planned" to indicate to reviewers that the contents are not actually ready for review, effectively pausing the review. And if the diff has been accepted, marked "back-to-author" or marked "changes planned", the author can request another review, typically after updating the code change. Lastly, the author

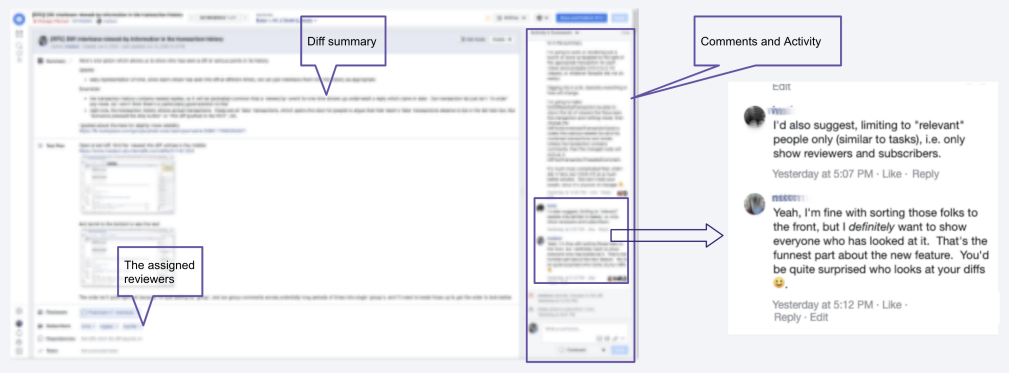

Figure 1: A redacted view of a diff under review in `DiffTool`. Authors and reviewers can interact via the diff review page in Phabricator. The diff under review has several sections including the diff summary, the actual changes that happened, the assigned code reviewers, the interactions between the various reviewers and author, the test case information and status, the results of static analysis, historical information etc.

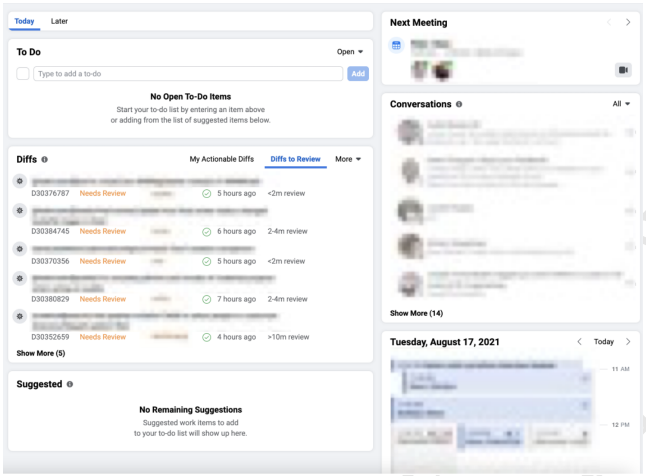

Figure 2: A redacted view of the Internal Home page: a multi-purpose surface that includes a widget showing up to five relevant diffs

can "abandon" the diff at any time, which indicates that the change is dead and will not be shipped. Throughout this process, the diff author's ability to ship their code change is gated by this diff approval process.

2.2 Meta Developer Experience Survey

The Developer Experience Survey is a company-wide survey run twice annually. The data we use here were collected between October 19 and November 2, 2020 (*i.e.* H2 2020). Participants were recruited from among all Meta developers who had published more than 10 diffs and more than 50 lines of code over the past 90 days at study launch time.

The H2 2020 survey took about 12 minutes to complete, and covered all areas of a developer's experience, from the organization of work through to coding, code review, testing, landing, etc. The section on code review asks about developers' satisfaction with 1) the time it takes to get your diffs reviewed by peers; 2) your ability to effectively review your peers' diffs; and 3) the quality of reviews on your diffs. We received approximately 2000 responses to questions about participants' code review experience, which were shown to only a subsample of all survey respondents

Since employee IDs are associated with participants' survey answers, we can run analyses that join three data sources: employee demographics (such as tenure, organization, role), telemetry data (such as tool usage, coding languages, etc.), and survey data *at the user level* as long as the confidentiality of each individual's response is preserved. This allows direct research into how sentiment data is correlated with demographic and behavioral data, and supports deeper insight about the intersection of experience and behavior.

2.3 Meta Developer Telemetry & Metrics

Meta logs business activity associated with each developer's work. With respect to code review, every diff action is logged: publishing a diff, commenting on a diff, adding a reviewer to a diff, accepting a diff, shipping a diff, etc. This telemetry also captures interactions between diffs and both humans and bots, for instance when an automated rule triggers a comment on a diff.

Every time a user views a diff is also logged, including the specific surface used to open the diff (*e.g.*, opening a diff via email notification vs the central diffs dashboard), regardless of whether the user took any action. Various heuristics are applied to estimate the "eyeball time" that a user spent looking at a particular diff. Specifically, we track the duration that a particular diff is shown in the active tab of a particular user's browser, and sessionize user activity when the user goes idle (stop or pause).

This telemetry can be aggregated into author metrics and reviewer metrics. For instance, we can compute how many diffs an author has published or how long their diffs took to review during a given time frame. Similarly, we can compute how many diffs a reviewer has acted upon or how much time was spent looking at others' active diffs. Finally, we can also compute linkages between users, for instance how many times User A has commented on User

B's diffs, or how much time User C has spent looking at User D's diffs.

2.4 A/B Experiments at Meta

Meta has a culture and infrastructure that allows engineers and data scientists to run controlled experiments on all major changes and new features. In the controlled A/B trial, the old control feature "A" is experimentally compared with the new feature "B". The experimentation framework allows experimenters to: 1) assign users to groups through feature toggles, 2) collect experimental data, and 3) run basic hypothesis testing. Before the experiment begins, the expected goal outcome measures are specified, and guard metrics are used to ensure that other important outcomes are not negatively impacted. The code review team at Meta leverages A/B testing to determine the efficacy of new features and make launch decisions based on the experiment results. Specifically, the instrumentation infrastructure allows to answer: Do we observe substantial and statistically significant improvements in goal metrics without regressions in the guard metrics?

We note that the code review team at Meta is faced with unique challenges with A/B testing compared to externally-facing user products:

- Given that the `DiffTool` is an internal tool, it does not have as many users as externally-facing user products. Since there is smaller sample size, less data available, and more noise, the code review team usually launches A/B tests to all developers at Meta. In this experiment we have over 30k participants.
- `DiffTool` has large social effects, since code review is a highly social activity. For instance, Alice's behavior as a diff reviewer affects Bob's experience as a diff author. Therefore, to contain the spillover effect between test and control groups, a cluster-randomized experiment design is often needed [34].

We discuss the full design of the stale diff nudge experiment in the methodology Section 5.2.

3 RQ1, SURVEY AND TELEMETRY

When does a diff review feel too slow?

For RQ1, the first step was to define time in review. We computed this as the total time a diff spends in the "Needs Review" status, i.e. when the diff was explicitly waiting on a reviewer action, summing over all rounds of review. To illustrate this with an example, consider a hypothetical diff that was submitted for review at 9am, sent back to the author for revisions at 11am, re-submitted for review at 2pm, and ultimately accepted at 3pm. We would sum the two "in review" durations, i.e. 9am-11am and 2pm-3pm, to say this diff spent 3 hours in review. In this way, we specifically capture the extent to which the developer was blocked waiting for the reviewer(s). From there, we computed the 75th percentile, *i.e.* p75, for each developer's time in review, aggregating over all of the developer's diffs during July to November 2020, the quarter corresponding to the survey. We selected the 75th percentile because this represents the lower bound of the 25% of diffs that take the longest for each engineer.

We proceeded to join this data to each developer's response to the survey question *"How satisfied are you with the time it takes to get your diffs reviewed by peers?"* To preserve the confidentiality of each developer's survey response, we binned all developers into deciles based on their p75 diff review time and only show these numbers for decile aggregations. From the survey alone, we learned that 84.7% of developers are satisfied with the time their diffs spend in review, i.e. 15.3% dissatisfied. With the combination of survey and telemetry, we are able to explore whether the developer's actual experience waiting for review in fact influences their subjective perception of review timeliness.

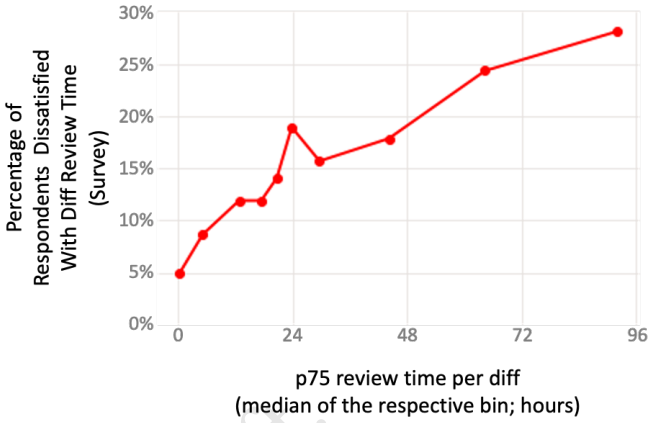

Figure 3: Percentage of users dissatisfied with diff review time vs the time it takes for their slowest 25% of diffs to be reviewed. To preserve anonymity, we aggregate by decile, which means that each datapoint is an aggregation of ~185 users. We see that as dissatisfaction increases with slower diff reviews.

Figure 3 plots the level of dissatisfaction against the developer's 75th percentile of time in review. We see a clear relationship between an increase in dissatisfaction and the amount of time it takes for engineers to get their diff reviewed. There is no sharp cliff or threshold that separates a "good" experience from a "bad" one. Rather, as an engineer's diff review time increases so does their frustration. After discussion and examination of the time in review distribution, the code review team decided to consider a diff that has had no action for 24 hours as stale. This threshold targets the slowest 25% of diffs that have been waiting for review.

> As a engineer's diff review time increases so does their frustration. We set a nudge threshold at 24 hours, which roughly corresponds with the slowest 25% of engineers' diffs.

4 RQ2, WHO TO NUDGE MODEL

Who should be nudged when a diff is stale?

We created a model to predict whether a reviewer will comment on or otherwise act upon, *e.g.,* accept or reject, a diff. The features are presented in Table 1. We also created submodels to understand how well subcategories of features perform. We briefly provide background on the features and their respective categories below.

Table 1: The Features and Feature Categories for RQ3: `Who2Nudge` Model. Note: the relationship features include raw, normalized by author and normalized by review.

Category	Feature	Description
Diff info	rule based change	Is this diff created using a rule? An example, if a human wants to *rename* a widely used class across the entire code base, they would apply a tool-based rule to implement the diff.
Diff info	num files changed	Number of files in the diff
Diff info	author is bot	Is the code author a bot or developer
Diff info	total num reviewer	Total number of reviewers assigned to the diff. (not all assignees will review the diff)
Diff info	num manually added reviewers	Total number of reviewers that were added by the author or another developer to the diff.
Assignment	expertise attribution score	The target reviewer's relevance score as computed by prior file line edits (prior work includes [18, 37, 40])
Assignment	expertise attribution rank	The target reviewer's rank as computed by prior file line edits
Assignment	auto added	Was the target reviewer added by a bot or rule (true), or directly by a human being (false)?
Assignment	only group added	Was the target reviewer added explicitly (false) or only as part of a team (true)?
Relationship	diff reviewer count	The number of diffs created in the last 90 days where the target reviewer was a reviewer on a diff by the author, whether individually or through a group (prior works include [25, 39])
Relationship	diff reviewer subscriber count	The number of diffs created in the last 90 days where the target reviewer was a reviewer or subscriber on a diff by the author, whether individually or through a group
Relationship	diff eyeball time open status as reviewer	The total number of seconds the target reviewer has spent viewing the author's diffs as an assigned reviewer, while such diffs were in an open status, in the last 90 days. Open status is any of: needs review, accepted, waiting for author, changes planned
Relationship	diff eyeball time open status	The total number of seconds the target reviewer has spent viewing the author's diffs, while such diffs were in an open status, in the last 90 days. Open status is any of: needs review, accepted, waiting for author, changes planned
Relationship	author reviewer same team	Are the author and reviewer on the same team
Relationship	author reviewer title	Proportion of all diffs by employees with author's job title that were acted/commented on by employees with the target reviewer's job title within the past 90 days. Captures, for example, how likely is a data scientist to comment on a software engineer's diff.

Diff info: Does information about the diff impact the diff ranking for a reviewer? We examined the impact of predictors including the type of change, the size of the change, and how many reviewers have been notified about the diff. These features have less common in prior research on reviewer recommendation because they do not suggest anything about the relative importance of each reviewer. Nevertheless, diffs with many reviewers may have a lower probability that any *particular* reviewer act upon the diff, and diffs with more complex changes may attract engagement from more reviewers. We reasoned that these features may interact with features from other categories.

How assigned: How was the reviewer selected to review the diff? Was the reviewer added directly by the author, or through a group, by a rule, or by prior knowledge of the files under review? The first three are straightforward selections. The fourth uses expertise attribution as measured by the number of lines in the files under review that a developer has modified in the past.

Relationships: We examined the organizational relationship of the reviewer and author, the number of times they have reviewed each other's code, and the eyeball time or viewing time that they have spent on each other's diffs. For non-organizational relationships, we included three variants: unnormalized, normalized by author, and normalized by reviewer. For example, using the eyeball time features, if we normalize by author, we would divide by the total time that all reviewers have spent looking at that author's past diffs.

4.1 Random Forest and Data

To evaluate our `Who2Nudge` Model and features, we determined whether a reviewer will review the diff by the end of a diff review session. If a diff went through multiple rounds of reviews and edits, each round of review was considered as a distinct review session for our model training and evaluation. We assigned a positive label to a (diff review session, reviewer) pair when the reviewer made an action such as accepting the diff, rejecting the diff, or leaving a comment. We assigned a negative label to a (diff review session,

reviewer) pair when the reviewer resigns from the diff or takes no action. We built a *random forest* for each set of features and combined them in a final model. We split the data into a time-ordered test set and a training set, using data available *before* the review session began. Our training set included all diff review sessions during a 1.5 month period in Summer 2021 (9,170,299 diff review sessions); our test set contains all diff review sessions the subsequent week (331,239 diff review sessions).

To determine feature importances, we were able to extract the magnitudes directly from the random forest. However, the directionality of each feature's influence is not readily accessible from random forest models. To infer the directionality of each feature's influence, we constructed a decision tree using features for the relevant submodel and manually inspected the tree's splits along with the values in each leaf node. These decision trees were used only for discussion purposes, namely to enrich our understanding of the feature importances with directionalities along with the magnitudes at which the most important splits occur.

We use two outcome measures: MRR and AUC. MRR (Mean Reciprocal Rank) is defined as the average of the reciprocal of the rank of the first relevant diff for any particular set of predictions for a developer. AUC (Area Under the Curve) is defined as the area under the ROC curve, and can be interpreted as the probability that a classifier will rank a randomly chosen positive instance higher than a randomly chosen negative one. For a useless classifier that gives random predictions, we would have an AUC equal to 0.5, and for a classifier that predicts perfectly, an AUC equal to 1. We calculated AUC across all reviews.

4.2 Model Results

Our model needs to nudge developers who are likely to take action, so we predict the probability of a developer participating in a diff review. Table 2 shows the AUC and MRR for each submodel and the final model. Below, we discuss the importance of feature categories and individual features for each submodel.

Using only basic *diff info*, we achieve an AUC of 0.73 and an MRR of 0.67. Examining the model, we see that the total number of reviewers assigned to a review has an importance of 0.95. This implies that when there are few potential reviewers for a diff, it is simplier to decide who to nudge. The remaining features contribute little. For instance, the importance score of the number of files in the diff is low, 0.013.

Expertise Attribution. Many review recommenders use prior editing experience in the files under review in their rules or models, *e.g.,* [24, 40]. Reviewer with more expertise on the files under review are more likely to act on a diff and we see a reasonable AUC and MRR of 0.66 and 0.69, respectively.

How Assigned. This submodel includes the Expertise Attribution features along with whether the reviewer was automatically assigned or assigned as part of a group. By adding these "how assigned" features, we see a substantial improvement in AUC and MRR to 0.75 and 0.79, respectively. Reviewers that were directly added are more likely to act on the diff, and the most influential factor, at 0.58, is whether or not the reviewer was added as an individual vs added as part of a group. The expertise attribution rank was the next influential factor at 0.23.

Author-Reviewer Relationships. The organizational relationship and latent review relationships between the author and reviewers have an AUC and MRR of 0.82 and 0.74, respectively. Interestingly, the formal position and whether the author and reviewer were on the same team had the lowest importance at 0.0007 and 0.0008, respectively. The strongest features are related to normalized eyeball time: the more time reviewers had previously spent looking at a particular author's diffs, the more likely they were to act on the review. The total importance of the eyeball time features was 0.71, which includes importances of 0.56 when normalized by author, 0.12 when normalized by reviewer, and 0.03 when unnormalized. This also indicates that normalization by author is the most effective approach, much more so than normalization by reviewer.

The *final* `Who2Nudge` *Model* combines all the features and has improved AUC and MRR of 0.88 and 0.81, respectively. These results were sufficiently robust to begin productionizing the model and begin the rollout and experiment of `NudgeBot`. We did, however, eliminate the feature "author reviewer title" because of the additional complexity and processing time required in its calculation and its low importance to the model. For the final production model, we improved efficiency by batching predictions, instead of separate predictions for each diff-reviewer pair. We also eliminated the feature "author reviewer title" because of the additional complexity and processing time required in its calculation and its low importance to the model.

> The most important features in the `Who2Nudge` Model are the amount of time the reviewers have viewed the authors' diffs in the past, how the reviewer was assigned, and the total number of assigned reviewers. The AUC and MRR of the final `Who2Nudge` Model are 0.88 and 0.81.

5 RQ3, EXPERIMENT AND ROLLOUT

Does stale diff nudging work in practice?

With `NudgeBot`, our core hypothesis is that nudging reviewers for stale diffs can reduce the review cycle time, and improve percentage of diffs closed within 48 hours and 72 hours, since we only nudge diffs after 24 hours. Our main quantifiable concern for potential negative side-effects is that reviewers may spend less time reviewing new diffs when they receive notifications for stale diffs. We first describe the tool design and changes we made from an opt-in trial. We then describe our company wide experiment and the impact on our goal and guard measures.

5.1 `NudgeBot` Design and Opt-in Trial

At Meta changes to the engineering infrastructure first pass through an opt-in trial. The `DiffTool` team used an early iteration of `NudgeBot` along with 15 other teams that opted into trail. Figure 4 provides an example of `NudgeBot` sending a chat message displaying three stale diffs and providing developers with the option to be reminded later. Below we discuss some of the important design decisions and feedback that lead to changes:

- When to nudge. Based on the distribution of time in review and survey responses (see Section 3), we decided to consider a diff stale and nudge a reviewer when no action had been

Table 2: The `Who2Nudge` Model and the component parts of the model. We see that how the diff was initially assigned and how often a reviewer has viewed the author's diffs in the past are the strongest predictors.

	Diff Info	Expertise Attribution	How Assigned	Relationships	`Who2Nudge` Model
MRR	0.67	0.69	0.75	0.74	0.81
AUC	0.73	0.66	0.79	0.82	0.88

taken on it for 24 hours. We attempted to create a model to predict how long a diff will be in review, but were unsuccessful [10].

- Medium. At Meta chat is by far the most popular communication method, we nudge on chat, but also send email nudges, display the nudges in the general notification list, and the diff dashboard.
- Default to #silent. Developers receive many IMs every day, and adding a `NudgeBot` messages can lead to more noise. We want developers to be aware of diffs that need attention but do not want to distract them from the task at hand. Our bot defaults to sending messages silently that developers will look at in-between tasks.
- Time of day. While we discussed incorporating nudges within the calendar, eg nudging after a meeting, we decided to send the nudges one hour after the start of the workday. We also allowed developers to select the "remind me later option." This option will nudge the reviewer again after a meeting to avoid disrupting a longer focus block.
- Batch notifications. We do not want `NudgeBot` to be noisy with a notification each time a diff is stale. We batch notifications and send them once in a day. We also do not notify the same developer more than once about a stale diff. We initially sent up to 10 diffs that needed review. Negative developer feedback indicated that the dashboard showing stale diffs took up too much space, so we reduced it to three diffs.
- Active reviewers. Sending a nudge to developers when they are already reviewing lots of diffs lead to negative feedback comments. We do not nudge extremely active reviewers, which we quantified as 40 or more actions on diffs in the last 7 days.
- Wording. The initial nudge wording provided a long explanation of why we were nudging particular reviewer. Based on feedback from developers, we simplified it to a short sentence asking developers to "unblock" a diff and simply show how long the diff had been waiting for review.
- Opt-out. Developers are able to opt-out from `NudgeBot`. To date, less than 1% of engineers opt-out. This is impressive, given that we send between 2k and 4k nudges per day. Over the course of the company-wide experiment, we see less than four developers opting out per day, and 1 to 2 developers opting back in per day.

5.2 Experimental Design

Experiments are only valid when we can contain the treatment effect within the test group, and control group does not expose to any of the treatment effect. This is formalized in Causal Inference theory as the "stable unit treatment value assumption" (SUTVA) [32]. However, SUTVA is not held perfectly with code review, since code review is inherently social activity between authors and reviewers collaborating on a diff. Consider the following scenario: Bob and Alice are frequent diff review collaborators. With user-level randomization, Alice is assigned to the test group to receive the nudging feature, while Bob is assigned to the control group. Alice will get nudged for overdue diffs authored by Bob, but Bob will not get nudged for overdue diffs authored by Alice. Since Alice and Bob collaborate frequently on code review, although Bob does not receive the nudging experience, his code review behavior may still change since he knows Alice gets nudged for overdue diffs authored by him. This is spill over of treatment effects, which may lead to inaccurate treatment effect estimates [6, 34]. To contain the spill over effects between test and control group in the experiment, we set this up as a cluster-randomized experiment.

In a cluster-randomized experiment, treatment is assigned at user cluster level, rather than individual user level [34]. For instance, users who belong to the same cluster will enter treatment or control group together. In the context of the overdue diff nudging experiment, teammates and their frequent diff collaborators will have the same experience with the nudging feature. To generate the clusters of developers, we used the Louvain community detection algorithm [8] to create clusters of developers based on the eyeball time spent reviewing each other's diffs together in the last 90 days. User clusters are generated with the following steps:

- We first construct a graph with developers at Meta as nodes, and use the total number of seconds person A spent looking at person B's diffs in the past 90 days while such diffs were in an open status (needs review, accepted, changes planned, waiting for author) as the edge weight between the nodes. We chose 90 days because it is common for developers to switch teams or change code review collaborators.
- We then use the Louvain community detection algorithm to create clusters based on these edge weights. Louvain algorithm identifies communities in graph with two steps: modularity optimization and community aggregation [8].

The outcome is that developers who have *commonly* reviewed each other's diffs in the last 90 days are more likely be in the same community cluster, to reduce spillover effects. The clustering was locked as of October 8, 2021. New employees who joined Meta after October 8, 2021 received the control experience, and they were excluded from the experiment analysis.

The experiment took place over 28 days in Fall 2021. The experiment was launched at Meta, with 15k developers in the test group, and 16k developers in the control group. We anticipated there would be queue-draining effect of stale diffs when the experiment started because nudging reviewers should encourage them to

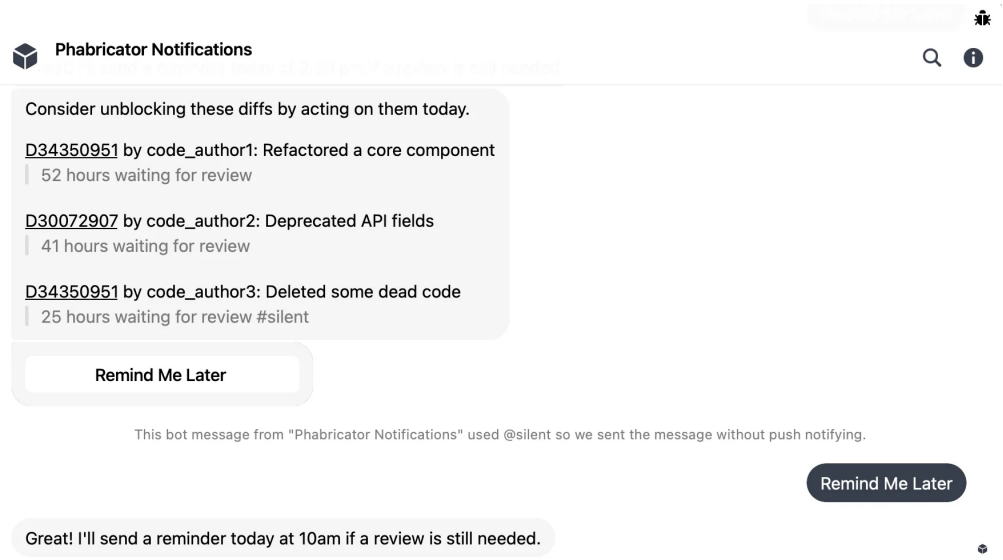

Figure 4: NudgeBot sends up to three diffs that are stale to a developer who is likely to review them one hour after the start of the workday. The chat message is sent with #silent, which will not push notify the developer, but will allow them to view the message between blocks of focus. The time the diff has been waiting is shown along with the username of the author and a clickable link to the diff. If the reviewer is busy, they can select "Remind Me Later." NudgeBot then examines the calendar and selects the next fragmented time, *e.g.,* between two meetings, to remind the developer later in the day. Emails are sent with the same information and same ability to be reminded later. The diff numbers, log message, etc are mocked to preserve confidentiality, but they are representative of real diffs at Meta.

review diffs that have been waiting for a long time. To avoid abnormal fluctuations of the review cycle time goal metrics, we discarded the first 7 days and analyzed data over the subsequent 21 days. The experiment included reviews on over 330k diff. We confirmed that the mean difference between control and experiment conditions was normally distributed, which allows us to use a 95% confidence interval and the Z-statistic to calculate statistical significance.

Outcomes. At Meta each experiment change must specify outcome goal metrics as well as existing guard metrics that should not be negatively impacted. Our hypothesis is that with NudgeBot the amount of time a diff is under review will decrease. We measure this in three ways: (1) the time a diff waits in with "needs review" status, (2) we targeted the slowest 25% of diffs, so we look at the number of diffs that take over 3 days to close, (3) we want to encourage early actions, so we measure the time to first action.

When we nudge stale reviews, we make sure that diffs that were typically reviewed quickly do not suffer because reviewers are spending more time on stale diffs. Our guard measures are: (1) the percentage of diffs reviewed in 24 hours or less, (2) the time a reviewers speeds looking at each diff (eyeball time).

5.3 Experimental Results

We observed statistically significant improvement on the review cycle time goal metrics, specifically:

(1) The average time of diffs in "needs review" status (time in review) reduced by -6.8% (p=0.049).
(2) The percentage of diffs taking longer than 3 days to close, excluding weekends [10], was reduced by -11.89% (p=0.004).
(3) The average time to first reviewer comment or action reduced by -9.9% (p=0.010).

We did *not* observe any statistically significant regressions in any of the guardrail metrics, specifically:

(1) Percentage of diffs reviewed in 24 hours (p=0.43)
(2) Total reviewer eyeball time while the diff is in needs review status (p=0.39)

> We observed substantial statistically significant decreases in our goal metrics reducing time in review, the number of diffs taking longer than 3 days, and the time for a first reviewer action by -6.8%, -11.89%, -9.9%, respectively. We did not observe any statistically significant regressions in any of our guard metrics including diffs that take under 24 hours and the amount of time reviewers spend looking at diffs. Given the positive results we observed from experiment, the code review team rolled out NudgeBot to 100% of developers at Meta.

6 THREATS TO VALIDITY

6.1 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 paper were performed at Meta, and it is possible that results might not hold true elsewhere. For this reason, we cannot

assume a priori that the results of our study will generalize beyond the specific environment in which our research was conducted. Researchers become more confident in a theory when similar findings emerge in different contexts [5].

6.2 Construct Validity

Who2Nudge *Model* The labels for this model were a binary classifications of whether or not a reviewer "reviewed" a diff during any particular review iteration, which we defined as any action (including accept/reject but not resign) or making any comment. One could argue that only actions should count, not comments. Conversely, one could argue that reviewer edits to a diff's metadata (*e.g.,* a reviewer changing a diff's test plan), or even reviewer interactions between review sessions (*e.g.,* while a diff is in the "waiting for author" status) should be considered. Ultimately, the precise definition of what counts as a user "reviewing" a diff requires subjective judgement over what to include, and we are confident that our definition is reasonable and aligns with standard understanding.

6.3 Internal Validity

Data collected by the Developer Experience Survey in H2 2020 might not have been fully representative of the developer population at Meta, since responses were not weighted. Our post-hoc analyses confirmed undersampling of people with very short and very long average diff review times, however the impact of this skewed distribution only minimally influenced overall estimates of sentiment towards code review times.

We did not perform any hyperparameter tuning for our Who2Nudge Model, which leaves open the possibility that these models can be further improved through a rigorous hyperparameter optimization. By using unoptimized defaults, the hyperparameters were fixed before looking at the test set to ensure no contamination. In addition, there is no reason that a hyperparameter search on random forest would result in dramatic changes in which features are considered the most important, which is what we focused on in the end. The first hyperparameter of "depth" would mostly impact features that are less important, since most major features would already be accounted for in the first few layers of the model's trees. In addition, the other hyperparamer of "number of trees" would be expected to have negligible impact on ranked feature importance given that each tree in the random forest is independent.

Code review is a social activity and any experiment risks having collaborating developers in different experimental conditions, *i.e.* "spillover effects." We are unaware of prior work in empirical software engineering that controlled for these network effects. To limit the impact of collaborating developers having different control and experimental conditions, we used a cluster-randomized experimental design to algorithmically-generated clusters. There will always be some spillover effects because of error from Louvain community detection algorithm and developers switching teams during the experiment, but we believe these will be minimal, especially compared to prior work that did not mitigate these effects.

7 LITERATURE AND DISCUSSION

Code review was first formalized over 45 years ago by Fagan [15] where author, reviewers, mediators met in person to examine completed work artifacts. The inspect was conducted over weeks and sometimes months. Much of the early work simply changed the process, often adding more rigidity through process(*e.g.,* [19]). The use of time in review as an outcome metric dates back to the field-changing experiments by Porter *et al.* [27] that demonstrated that formality of the review process simply extended time in review without identifying additional defects. They showed that expertise was the most important factor in identifying defects. Votta [38] proposed asynchronous review and Perry *et al.* [26] demonstrated that this lighter-weight process found the same number of defects as conducting reviews in meetings. In these late 1990's experiments the review time was on the order of a week. Rigby *et al.* [28, 30] used similar metrics to show how effective the extremely lightweight, hugely iterative code review process was on major open source projects. On the Linux and Apache project the review time was on the order of a day. These results generalized to industry with reviews being conducted in less than one day [29]. At Meta, we showed that reviews happen very quickly, often being completed in a few hours after publishing the diff (median 2.5 hours), and with around 75% being completed in under 24 hours.

Surveys and interviews have been used extensively to understand the code review process. Previous survey research about code review has focused on the motivations for conducting code reviews [3], challenges to code reviews [22], reviewers' experiences of confusion when reviewing code [13], and negative experiences with pushback during code reviews [14]. Some works have triangulated code review data with survey and interview data to understand the process and the challenges [20, 21, 31].

To our knowledge, this study is the first to link telemetry data on the time it takes to perform a review with developer sentiment. Only the Egelman *et al.* [14] study combined survey and telemetry data, and attempted to predict code review pushback from three change request metrics: rounds of a review (*i.e.* number of times an author or reviewer sent a batch a comments for a selected change request), active reviewing time (*i.e.* time spent by the reviewer on code review-related activities), and active shepherding time (*i.e.* time spent by the author on code review-related activities). While all three were useful for recalling instances of pushback, they only had low precision: between 6% and 11%. Overall, the literature to date does not seem to have explored systematically how an individual's sentiment about code review (in general, or about time spent under review in particular) is related to that individual's objective experience of code review as monitored by telemetry.

Review recommendation. Identifying the right reviewers for a given change is a challenging and critical step in the code review process [16, 40, 41]. Inappropriate selection of reviewers can slow down the review process [36] or lower the quality of inspection [3, 9]. The research on reviewer recommenders focuses on the problem of automatically assigning review requests to the developers. Early works focused the files and paths that developers had reviewed in the past [4, 17, 18, 37]. More advanced techniques have also been developed including machine learning and socio-technical factors [12, 25, 35, 39]. Recent research has expanded the

focus of recommenders to find developers who have currently a low review workload [1, 2, 11] and to ensure that knowledge is more widely spread among the development team [24]. In our nudging experiments, we recommended reviewers who are more likely to act on a diff on a stale diff using more advanced measures, including how long developers have actually viewed each other's diff (eyeball time). We found that this normalized eyeball time is the best predictor in our `Who2Nudge` Model, and achieved a high AUC and MRR, .88 and .81, and we successfully integrated this recommender in production for `DiffTool`.

Overdue Diff Nudging. The inspiration for our work is the Nudge system deployed at Microsoft [23] for overdue pull requests. The Microsoft context is substantially different from the one at Meta. For example, Meta uses a mono repo (single repository for code control) rather than the Microsoft branch structure [7]. Also the systems built are different, the domain, the programming language are all different. We discuss similarities and differences in our results and designs, which highlight our novel contributions. First, Microsoft models the end-to-end time including both the author's and reviewer's time. Unfortunately, they report very low accuracy with a mean average error of 32.60 hours and a mean relative error of .58. We also modelled time in review at Meta and found a similarly large error [10]. Instead of using this inaccurate model, we decided to set a goal of nudging reviews at 24 hours, which represents the 25% of slowest diffs and also corresponds to an increase in negativity sentiment in our developer survey. Second, both Microsoft and Meta's bots were A/B tested, but Microsoft did not run a cluster-randomized experiment. Without a cluster-randomized experiment, teammates can have different experiences of diff review, introducing a confounder into the results [34]. Our experiment also covered many more reviews than that reported in the Microsoft experiment. Third, Microsoft reduces the average end-to-end time spent in for code review from 8 days to 3 days. We explicitly only measure the time when the diff is waiting for a review, and the median time in review at Meta is 2.5 hours. We also do not nudge the author of the diff, as our tooling already includes a set of notifications that keep the author appraised of the status of their diff. Fifth, regardless of these differences, at both Microsoft and Meta, nudging stale diffs improves the turnaround time for reviews, and the feedback on nudging is overwhelmingly positive. At Microsoft, 73% of nudges get a positive rating. At Meta, we see few developers posting concerns on the `NudgeBot` feedback group and less than 1% of developers have opted-out of receiving nudges after a company-wide rollout.

8 CONTRIBUTIONS AND CONCLUDING REMARKS

Code review is a computationally expensive, manually-intensive practice that is necessary to find defects and also for compliance of products at Meta. We make code review more efficient and enjoyable with the following specific contributions.

- We describe the review process used by over 30,000 developers at Meta, which has interesting differences with those reported by Microsoft [3] and Google [33].
- We demonstrate determine when a diff review feels slow: a review that takes more than 24 hours or is above the 75th percentile of diff review time is associated with lower satisfaction in our developer survey.
- We train and test the `Who2Nudge` Model to determine which developer is mostly likely to take action when nudged. Our model has an AUC and MRR of 0.88 and 0.81, and is used in production.
- We designed and developed `NudgeBot`. We conducted an opt-in trial and adapted the design based on feedback from developers.
- To evaluate `NudgeBot`, we designed a cluster-randomized experiment to ensure that collaborating diffs had a similar experience [34]. We are unaware of cluster-randomized experiment design being used in the context of software engineering. Our study involved over 30,000 developers at Meta.
- Our `NudgeBot` experiment lead to substantial statistically significant decreases in our goal metrics, with time in review being reduced by -6.8% (p=0.049), a -11.89% (p=0.004) decrease in the percentage of diffs taking longer than 3 days to close, and a -9.9% (p=0.010) decrease in the time to first comment or action from a reviewer. We did not observe any statistically significant regressions in any of our guard metrics including diffs under 24 hours and the amount of time reviewers spend looking at diffs.
- Given the positive results we observed from experiment, the code review team rolled out `NudgeBot` to all developers at Meta.

ACKNOWLEDGEMENTS

The authors thank Katherine Zak, Nolan Sandberg, Ivan Mistrianu, Tobi Akomolede, and Akin Olugbade for their feedback and help with this work.

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* (Virtual, USA) *(PROMISE 2020)*. Association for Computing Machinery, New York, NY, USA, 21–30. https://doi.org/10.1145/3416508.3417115

[2] 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.

[3] 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.

[4] 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.

[5] V.R. Basili, F. Shull, and F. Lanubile. 1999. Building knowledge through families of experiments. *IEEE Transactions on Software Engineering* 25, 4 (1999), 456–473. https://doi.org/10.1109/32.799939

[6] Guillaume Basse and Edoardo Airoldi. 2017. Limitations of design-based causal inference and A/B testing under arbitrary and network interference. *arXiv preprint arXiv:1705.05752* (2017).

[7] Christian Bird and Thomas Zimmermann. 2012. Assessing the Value of Branches with What-If Analysis. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering* (Cary, North Carolina) *(FSE '12)*. Article 45, 11 pages.

[8] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. 2008. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment* 2008, 10 (oct 2008), P10008. https://doi.org/10.1088/1742-5468/2008/10/p10008

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

[10] Lawrence Chen, Peter C. Rigby, and Nachiappan Nagappan. 2022. Understanding why we cannot model how long a code review will take at Meta *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 6 pages. https://doi.org/10.1145/3468264.3473930

[11] Moataz Chouchen, Ali Ouni, Mohamed Wiem Mkaouer, Raula Gaikovina Kula, and Katsuro Inoue. 2021. WhoReview: A multi-objective search-based approach for code reviewers recommendation in modern code review. *Applied Soft Computing* 100 (2021), 106908.

[12] A. Chueshev, J. Lawall, R. Bendraou, and T. Ziadi. 2020. Expanding the Number of Reviewers in Open-Source Projects by Recommending Appropriate Developers. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 499–510. https://doi.org/10.1109/ICSME46990.2020.00054

[13] Felipe Ebert, Fernando Castor, Nicole Novielli, and Alexander Serebrenik. 2021. An exploratory study on confusion in code reviews. *Empirical Software Engineering* 26, 1 (2021), 1–48.

[14] Carolyn D Egelman, Emerson Murphy-Hill, Elizabeth Kammer, Margaret Morrow Hodges, Collin Green, Ciera Jaspan, and James Lin. 2020. Predicting developers' negative feelings about code review. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 174–185.

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

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

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

[18] Gaeul Jeong, Sunghun Kim, Thomas Zimmermann, and Kwangkeun Yi. 2009. Improving code review by predicting reviewers and acceptance of patches. *Research on software analysis for error-free computing center Tech-Memo (ROSAEC MEMO 2009-006)* (2009), 1–18.

[19] John C Knight and E Ann Myers. 1993. An improved inspection technique. *Commun. ACM* 36, 11 (1993), 50–61.

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

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

[22] L. MacLeod, M. Greiler, M. Storey, C. Bird, and J. Czerwonka. 2018. Code Reviewing in the Trenches: Challenges and Best Practices. *IEEE Software* 35, 4 (2018), 34–42. https://doi.org/10.1109/MS.2017.265100500

[23] Chandra Maddila, Sai Surya Upadrasta, Chetan Bansal, Nachiappan Nagappan, Georgios Gousios, and Arie van Deursen. 2020. Nudge: Accelerating Overdue Pull Requests Towards Completion. *arXiv preprint arXiv:2011.12468* (2020).

[24] Ehsan Mirsaeedi 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* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 1183–1195. https://doi.org/10.1145/3377811.3380335

[25] Ali Ouni, Raula Gaikovina Kula, and Katsuro Inoue. 2016. Search-Based Peer Reviewers Recommendation in Modern Code Review. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 367–377. https://doi.org/10.1109/ICSME.2016.65

[26] D.E. Perry, A. Porter, M.W. Wade, L.G. Votta, and J. Perpich. 2002. Reducing inspection interval in large-scale software development. *IEEE Transactions on Software Engineering* 28, 7 (2002), 695–705. https://doi.org/10.1109/TSE.2002.1019483

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

[28] Peter Rigby, Daniel German, and Margaret-Anne Storey. 2008. Open source software peer review practices. In *2008 ACM/IEEE 30th International Conference on Software Engineering*. 541–550. https://doi.org/10.1145/1368088.1368162

[29] 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.

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

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

[32] Donald B. Rubin. 1990. Formal modes of statistical inference for causal effects. *Quality Engineering* 36 (1990), 185–188.

[33] 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.

[34] Martin Saveski, Jean Pouget-Abadie, Guillaume Saint-Jacques, Weitao Duan, Souvik Ghosh, Ya Xu, and Edoardo M. Airoldi. 2017. Detecting Network Effects: Randomizing Over Randomized Experiments. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (Halifax, NS, Canada) *(KDD '17)*. Association for Computing Machinery, New York, NY, USA, 1027–1035. https://doi.org/10.1145/3097983.3098192

[35] 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* (Seoul, South Korea) *(ICSE-SEIP '20)*. Association for Computing Machinery, New York, NY, USA, 1–10. https://doi.org/10.1145/3377813.3381365

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

[37] Patanamon Thongtanunam, Chakkrit Tantithamthavorn, Raula Gaikovina Kula, Norihiro Yoshida, Hajimu Iida, and Ken-ichi Matsumoto. 2015. Who should review my code? a file location-based code-reviewer recommendation approach for modern code review. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 141–150.

[38] Lawrence G. Votta. 1993. Does Every Inspection Need a Meeting? *SIGSOFT Softw. Eng. Notes* 18, 5 (Dec. 1993), 107–114. https://doi.org/10.1145/167049.167070

[39] 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.

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

[41] H. Alperen Çetin, Emre Doğan, and Eray Tüzün. 2021. A review of code reviewer recommendation studies: Challenges and future directions. *Science of Computer Programming* 208 (2021), 102652. https://doi.org/10.1016/j.scico.2021.102652