Nudge
Accelerating Overdue Pull Requests toward Completion

Maddila, C.S.; Upadrasta, Sai Surya Upadrasta; Bansal , Chetan; Nagappan, Nachiappan; Gousios, G.; van Deursen, A.

DOI
10.1145/3544791

Publication date
2022

Document Version
Final published version

Published in
ACM Transactions on Software Engineering and Methodology

Citation (APA)
Maddila, C. S., Upadrasta, S. S. U., Bansal , C., Nagappan, N., Gousios, G., & van Deursen, A. (2022). Nudge: Accelerating Overdue Pull Requests toward Completion. *ACM Transactions on Software Engineering and Methodology*, *32*(2), Article 35. https://doi.org/10.1145/3544791

Nudge: Accelerating Overdue Pull Requests toward Completion

CHANDRA MADDILA, SAI SURYA UPADRASTA, CHETAN BANSAL, and NACHIAPPAN NAGAPPAN, Microsoft Research
GEORGIOS GOUSIOS and ARIE VAN DEURSEN, Delft University of Technology

Pull requests are a key part of the collaborative software development and code review process today. However, pull requests can also slow down the software development process when the reviewer(s) or the author do not actively engage with the pull request. In this work, we design an end-to-end service, Nudge, for accelerating overdue pull requests toward completion by reminding the author or the reviewer(s) to engage with their overdue pull requests. First, we use models based on effort estimation and machine learning to predict the completion time for a given pull request. Second, we use activity detection to filter out pull requests that may be overdue but for which sufficient action is taking place nonetheless. Last, we use actor identification to understand who the blocker of the pull request is and *nudge* the appropriate actor (author or reviewer(s)). The key novelty of Nudge is that it succeeds in reducing pull request resolution time, while ensuring that developers perceive the notifications sent as useful, at the scale of thousands of repositories. In a randomized trial on 147 repositories in use at Microsoft, Nudge was able to reduce pull request resolution time by 60% for 8,500 pull requests, when compared to overdue pull requests for which Nudge did not send a notification. Furthermore, developers receiving Nudge notifications resolved 73% of these notifications as positive. We observed similar results when scaling up the deployment of Nudge to 8,000 repositories at Microsoft, for which Nudge sent 210,000 notifications during a full year. This demonstrates Nudge's ability to scale to thousands of repositories. Last, our qualitative analysis of a selection of Nudge notifications indicates areas for future research, such as taking dependencies among pull requests and developer availability into account.

CCS Concepts: • **Software and its engineering** → **Integrated and visual development environments**; **Software maintenance tools**; **Software configuration management and version control systems;**

Additional Key Words and Phrases: Pull-based software development, pull request, merge conflict, distributed software development

ACM Reference format:
Chandra Maddila, Sai Surya Upadrasta, Chetan Bansal, Nachiappan Nagappan, Georgios Gousios, and Arie Van Deursen. 2023. Nudge: Accelerating Overdue Pull Requests toward Completion. *ACM Trans. Softw. Eng. Methodol.* 32, 2, Article 35 (March 2023), 30 pages.
https://doi.org/10.1145/3544791

Chandra Maddila and Nachiappan Nagappan work done while at Microsoft Research.

Authors' addresses: C. Maddila, C. Bansal, and N. Nagappan, Microsoft Research, 14820 NE 36th St, Redmond, WA, USA; emails: chandu.maddila@gmail.com, chetanb@microsoft.com, nachiappan.nagappan@gmail.com; S. S. Upadrasta, Microsoft Research, Vigyan 1st floor, 9, Lavelle Road, Ashok Nagar, Bengaluru, India; email: upadrastasaisurya1@gmail.com; G. Gousios and A. Van Deursen, Delft University of Technology, Building 28, Van Mourik Broekmanweg 6, 2628 XE Delft, The Netherlands; emails: {g.gousios, arie.vandeursen}@tudelft.nl.

1049-331X/2023/03-ART35 $15.00
https://doi.org/10.1145/3544791

1 INTRODUCTION

With the adoption of collaborative software development platforms like GitHub and Azure DevOps, pull requests have become the standard mechanism for distributed code reviews. Pull requests enable developers as automated agents to collaboratively review the code before it gets integrated into the mainline development. Once the reviewers have signed off on the changes these can be merged with the main branch and deployed. Pull requests has recently become an active area of research in the software engineering community. Various aspects of pull requests have been studied, such as reviewer recommendation [7, 49], prioritization [43], and duplication [44]. Additionally, several bots and extensions have been built for platforms like GitHub and Azure DevOps to automate various software development workflows [20, 23].

While pull requests streamline the code review process significantly, they can also slow down the software development process. For instance, if the reviewers are overloaded and lose track of the pull request, then it might not be reviewed in a timely manner. Similarly, if the pull request author is not actively working on the pull request and reacting to the reviewers' comments, then the review process could be slowed down significantly. Hence, if the pull request's author and reviewers do not actively engage, then the pull requests can remain open for a long time, slowing down the coding process and possibly causing side effects such as merge conflicts. Yu et al. [48] did a retrospective study of the factors impacting pull request completion times. They found that pull request latency requires many independent variables to explain adequately, with the size of the pull request and the presence of a continuous integration pipeline as major factors. Long-lived feature branches can also cause several unintended consequences [6]. Some of the most common side effects caused by long-lived feature branches or pull requests are as follows:

- They hinder communication. Pull requests that are open for longer periods of time hide a developer's work from the rest of their team. Making code changes and merging them quickly increases source code re-usability by making the functionality and optimizations built by a developer available to other developers.
- In large organizations with thousands of developers working on the same codebase, the assumptions that a developer may make about the state of the code might not hold true, the longer they have their feature branches open. The developers become unaware of how their work affects others.
- Long-lived pull requests cause integration pain. When the code is merged more frequently to the main branch, integration testing can be done earlier, issues can be detected faster and bugs can be fixed at the earliest possible moment.
- Branches that stay diverged from the main branch for longer periods of time can cause complex merge conflicts that are hard to solve. Dias et al. [21] studied over 70,000 merge conflicts and found that code changes with long check-in times are more likely to result in merge conflicts.
- Overdue pull requests prevent companies from delivering value to their customers quickly. Organizations can deliver more value to their stakeholders by releasing new features or bug fixes in the organization's products or services earlier if the corresponding code is merged faster.

To address these concerns, we designed and deployed Nudge, a service for accelerating overdue pull requests toward completion. As its name suggests, Nudge sends a reminder if a pull request is overdue. We carefully designed Nudge so that it (1) actually achieves faster pull request resolution, (2) minimizes the number of notifications it sends to avoid disturbing developers unnecessarily, and (3) can operate at the scale of thousands of repositories and developers.

To realize these objectives, Nudge relies on effort estimation to predict the completion time for a given pull request. Next, it determines activities and identifies the actor (the reviewer(s) or the author) blocking the pull request from completion. It then notifies the identified actor through the comment functionality of the pull request environment.

To design and build Nudge, we first perform correlation analysis to understand which factors impact pull request completion time. We look at factors related to the pull request, its author, the underlying system, the team, and the role of the developer in the team. Unlike Yu et al. [48], we only consider factors that are known at the time of the pull request creation.

Next, we use effort estimation for predicting the pull request completion time at the time of pull request creation. Effort estimation models have been long studied in software engineering research. We build a model for predicting the completion time of a pull request on the rich body of work in the effort estimation literature. Prior work [25] has focused on effort estimation at the feature and project level but not at the level of individual pull requests. We use several metrics from the defect prediction literature like code churn [34], reviewer information [28], and ownership information [25] to build our pull request lifetime prediction model.

While effort estimation models have been shown to be accurate [8], they cannot account for contextual and environmental factors such as workload of the pull request reviewer(s) of the author. Therefore, to improve the notification precision, we implement *activity detection*, which monitors any updates on the pull request, such as new commits or review comments, and adjusts the notification accordingly. Furthermore, to determine *who* needs to receive the notification, we implement *actor identification* to infer the actor (pull request author or specific reviewer(s)) who is blocking the pull request from completion.

To assess to what extent Nudge has been able to meet its objectives, we conducted a number of experiments. To assess pull request resolution time and developer perception of notifications, we deployed Nudge to 147 repositories, using its telemetry functionality to collect data for a period of 9 months. During this period, Nudge identified 12,356 pull requests that were taking longer than the time Nudge predicted. We employed Nudge via a randomized trial by sending a notification to a subset of 8,500 (55%) randomly selected pull requests, thus allowing us to compare their resolution time with those for which no notification was sent. Our findings indicate a reduction of 60.62% in average pull request lifetime thanks to the use of Nudge. The vast majority (81.53%) of the notified pull requests are closed within a week.

To be able to assess the developer's perception of the Nudge notifications, we give users of Nudge recommendations the option to provide feedback, both via a negative/neutral/positive tick box and an open text field. We find that 73% of the pull requests received a positive resolution from the developers. We used the open answers to identify areas for future improvements, such as taking dependencies between pull requests into account (in case one pull request is blocking another).

To assess the scalability of Nudge, we monitored its deployment on 8,000 different systems at Microsoft from January 2021 until December 2021. During this period, Nudge sent 210,000 notifications authored by 40,000 unique developers. Since this is an actual deployment, unlike the randomized trial in our experiments, we have no "untreated" data points to compare to. Nevertheless, we see that 83.65% of the nudged pull requests are closed within a week, which is consistent with the findings from the randomized trial. Also, user satisfaction is similar, with 71% of the notifications receiving a positive resolution. From this, we conclude that the design of Nudge permits operation at the scale of thousands of repositories and that the positive results in terms of time reduction and user satisfaction remain valid.

Thus, the novelty of this article lies in the following key contributions:

(1) We propose a novel approach to warning developers and reviewers of pull requests when they are running late, combining effort estimation, activity detection, and actor identification (Sections 4–6).
(2) We design and deploy a scalable implementation of our approach in a tool called Nudge (Section 7).
(3) We demonstrate that the use of Nudge leads to a 60% speed-up of delayed pull requests and that over 70% of the developers warned about their pull requests appreciate such warnings as positive (Section 8).
(4) We apply Nudge to 8,000 systems and demonstrate that its benefits remain present at scale (Section 8).

This article is a substantially revised extension of our earlier publication [29]. New in the present article is the use of activity detection and actor identification, the evaluation of these, and the discussion of the application of Nudge to 8,000 systems in the period January–December 2021.

2 RELATED WORK

Our research relies on effort estimation techniques to determine the amount of time needed to decide whether a given pull request can be merged. Software effort estimation is a field of software engineering research that has been studied extensively in the past four decades [11, 14, 16, 18, 32]. Typically, in this line of research, one tries to predict either the effort needed to complete the entire project or the effort needed to finish a feature. One of the earliest effort estimation models was the COCOMO model proposed by Barry W. Boehm in his 1981 book, Software Engineering Economics [14], which he later updated to COCOMO 2.0 in 1995 [13]. This work was followed up by Briand et al. [15] who compared various effort estimation modeling techniques using the dataset curated by the European Space Agency. In all these cases a model was built for the entire software project and effort was estimated for function points. More recently, Menzies et al. [32] and Bettenburg et al. [11] looked at the variability present in the data and therefore built separate models for subsets of the data.

More recently there has been interest in predicting pull request acceptance, both the eventual decision (merge or abort), as well as the time, needed to make the decision. Soares et al. [38] and Tsay et al. [42] looked at a variety of factors to see which one had an impact on pull request acceptance. More specifically, Terrell et al. [41] and Rastogi et al. [36] looked at gender or geographical location impact on a pull request acceptance. The work closest to our work is by Yu et al. [48], who explored the various factors that could impact how long it took for an integrator to merge a pull request. Unlike their study, we do not examine what factors might impact the time taken to accept a pull request but rather how much time it would actually take for a pull request to be accepted. Hence, unlike past papers that were empirical studies on building knowledge with respect to pull request acceptance, we build a system that will predict how long it will take to accept a pull request and provide actionable feedback to the developers leveraging that knowledge.

As shown by various studies [11, 14, 16, 18, 32], effort estimation is a hard problem. One of the primary reasons that contribute to the errors is changing organizational dynamics, the landscape of the competition, and ever-changing schedules and priorities. Doing effort estimation at the pull request level reduces the uncertainty and the variability up to some degree.

Our ambition to devise a technique to warn developers about pull request delays can be viewed as a software development bot. The extensibility mechanisms provided by software development platforms like GitHub and Azure DevOps have enabled a huge ecosystem [5] of bots and automated services. It has also spawned active research [40] on understanding and building bots to assist

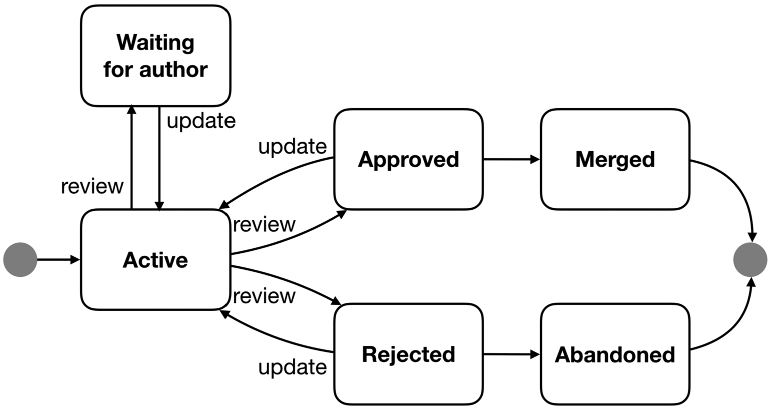


Fig. 1. The lifecycle of a pull request.

with various software engineering tasks. Storey et al. [40] have defined a *Software Engineering Bot* as software that automates a feature, performs a function normally done by humans, and interacts with humans. Lebeuf et al. [26] have proposed a taxonomy for software bots based on the environment, the intrinsic properties of the bot, and the bot's interaction with the environment. In terms of applications, prior work has focused on improving the code review process by automating reviewer recommendation [7, 49], diagnosing issues [9, 12, 30], refactoring [37, 47], and even intent understanding of the code changes [45, 46]. In this work, we built and deployed Nudge, which is a bot for increasing software development velocity and productivity by accelerating PR completion.

Bots warning about potential delays can be found beyond software development systems. General workflow management systems such as if-this-then-that [35] and Microsoft Power Automate [2] can be used for creating various automation workflows in domains such as smart home automation [24], healthcare [22], and smart mobility [31]. One of the biggest challenges with such tools is that they cannot take into account the complexity associated with internal state changes, and interactions between various actors in the systems they operate on. Such general systems are well suited for tasks like sending daily reports or reminders based on simple logic. For example, a pull request reminder system that is built using Power Automate [3] could check if a pull request is active. If it is active, then the system can trigger an email, on a pre-defined cadence, to all the reviewers of the pull request. While technically speaking such general notification systems could play a role in the implementation of Nudge, we offer tight integration with the pull request environment instead. This is not only most natural to the developers involved, but also enables us to determine the various attributes and state changes happening in each pull request based on which an alert has to be triggered, as well as the branching conditions that help determine whom the notification should be redirected to.

3 BACKGROUND: A PULL REQUEST'S LIFECYCLE

In this article, we assume a pull request goes through the lifecycle as depicted in Figure 1. Based on this, a pull request can be in one of the following states:

Active. The pull request has been published by the author. Reviewers are assigned and the pull request is open for code *review.*

Waiting for author. The reviewer has left review comments and expects the author to *update* the code to address them.

Approved. The reviewer was satisfied with the code changes in the pull request and approved it to be merged with the main branch. Thus, the author can merge and finalize or, optionally, decide additional *updates* are called for and re-start the reviewing process from the **Active** state.

Rejected. The reviewers are not satisfied with the code changes and reject the pull request. The author can attempt additional *updates* to restart the reviewing process, but, otherwise, the pull request will be rejected.

Merged. After the reviewers signed off on the pull request, the author successfully merged the code into the main branch.

Abandoned. The author of the pull request decides to not pursue the code changes further.

After the pull requested has been merged or abandoned, the pull request is closed and cannot be re-opened again (developers would need to open a *new* pull request instead).

To transition between these states, there are three different *actors* involved: Authors, reviewers, and non-human actors (bots):

Authors: Authors create a pull request in the first place. They send the pull request for review and keep working on the pull request by reacting to the review comments by pushing new changes (in the form of commits or iterations). Once all the reviewers are satisfied, they make the final decision to merge or not merge a change. They have a significant influence on the pace of the pull request. If they react to the review comments quickly and resolve them, then the pull request will have a better chance of making progress quickly. In Figure 1, they can trigger the transitions labeled with the *update* event.

Reviewers: Reviewers are added by the authors or any other automation tools (based on certain conditions) to pull requests. Reviewers have a responsibility to perform a thorough code review and provide their feedback. The agenda of the reviewers is to ensure the quality of the source code stays high and adheres to the standards imposed by their respective teams or organizations. Reviewers can be individuals in the same team or people with more experience and expertise in the area of source code that is being changed or groups that are a collection of individual reviewers. When a pull request is submitted for a review, reviewers can either approve or reject or make suggestions that need to be acted upon by the author of the change and resolve the comments made by the reviewers. By virtue of their role, reviewers can significantly impact the outcome of the review and the velocity of the pull request. In Figure 1, they can trigger the *review* transitions.

Non-human actors: With the increased use of bots and automation tools, non-human actors can also play a role in determining the velocity with which change progression happens. Tools that enforce security and compliance policies or styling guidelines or that ensure dependencies are not broken are some of the examples. Such bots can place comments like a reviewer would and thus trigger *review* transitions in Figure 1. The non-human actors, which sometimes act as code reviewers, do not contribute to the time taken to review pull requests. However, they impact the pull request status determination algorithm (explained in Section 6.1) by influencing the pull request state changes.

Pull request lifecycle is a complex process involving several actors and activities. However, it is also an important process, since inadequate code reviews can result in bugs and sub-optimal design with both short-term and long-term implications. Prior work has shown that the size of the code changes has a significant impact on the time taken for code reviews. However, there are several other factors that can also impact code review time. Baysal et al. [10] found that the reviewer's workload and past experience can impact the time taken for code reviews. Further other organizational (such as release deadlines) and geographical factors (collaboration across multiple time zones) can also influence the speed. While these factors are critical for faster code reviews but they are hard to change.

Often, developers are working on multiple projects and features at the same time. They are simultaneously working on code changes while reviewing other people's code reviews. So, it's very common to lose track of pending activities that might be blocking the pull requests. This problem is further amplified, since these code reviews are spread across multiple repositories. So, in this work, we build the Nudge tool to provide intelligent reminders to both the authors and the reviewers.

4 NUDGE SYSTEM DESIGN

The side effects manifested by pull requests that are open for longer periods of time and are prevalent in large organizations like Microsoft, as well as in large open source projects. Because of that, there has been a demand inside such organizations for a service that can help engineering teams alleviate the problems induced by long-running pull requests. We designed the Nudge system to address this problem and operationalized it across 147 repositories. We then performed a large-scale testing/validation of the effectiveness of the Nudge system by analyzing various metrics and collecting user feedback. In this section, we describe the design of the Nudge system in detail.

4.1 Design Overview

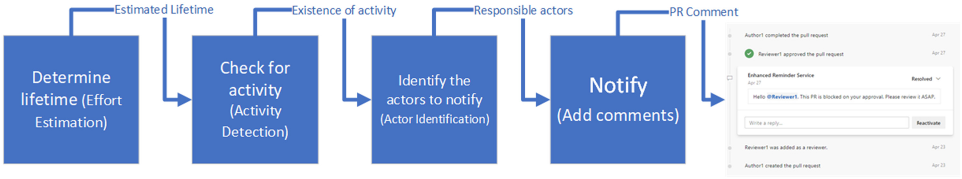
The Nudge system consists of three main components: A machine learning-based effort estimation model that predicts the lifetime of a given pull request, an activity detection module to establish what the current state of the pull request is, and an actor determination module to identify who would be need to take action.

Prediction Model. The Nudge system leverages a prediction model to determine the lifetime for every pull request. The model is a linear regression model as explained in Section 5. We performed the regression analysis to understand the weights of each of the features and how they impact the ability of the model to accurately predict the lifetime for a given pull request. We use historical pull request data to extract some of the features and the dependant variable (pull request lifetime).

For the repositories where we have enough training data, i.e., at least thousands of data points (or pull requests), we train a repository-specific model. If the repository is small or new and it does not have many pull requests that is completed, then we use a global model that is trained on all the repositories' data. Once the repository matures and records enough activity, we train a repository-specific model and deploy it. The models are retrained, through an offline process, periodically, to adjust to the changes in the feature weights and changing repository dynamics. Every time the model is retrained, we use a moving window to fetch the data from the past 2 years (from the date of retraining) to make sure the training data reflect the ever-changing dynamics and takes into account the changes happening to the development processes.

Activity Detection. The role of the activity detection module is to help the Nudge system understand if there has been any activity performed by the author or the reviewer of the pull request of late. This helps the Nudge system not send a notification, even though the lifetime of the pull request has exceeded its predicted lifetime. This module serves as a gatekeeper that gives the Nudge system a "go" or "no go" by observing various signals in the pull request environment.

Actor Identification. The primary goal of this module is to determine the blocker of the change (the author or a reviewer) and engage them in the notification, by explicitly mentioning them. This module comes into action once the pull request meets the criteria set by the prediction module and the Activity Detection modules. Once the Nudge system is ready to send the notification, the Actor Identification module provides information to the Nudge notification system to direct the notification toward the change blocker.


Estimated Lifetime
Existence of activity
Responsible actors
PR Comment
Determine lifetime (Effort Estimation)
Check for activity (Activity Detection)
Identify the actors to notify (Actor Identification)
Notify (Add comments)

Author1 completed the pull request Apr 27

Reviewer1 approved the pull request Apr 27

Enhanced Reminder Service
Apr 27

Hello @Reviewer1. This PR is blocked on your approval. Please review it ASAP.

Reviewer1 was added as a reviewer. Apr 23

Author1 created the pull request Apr 23

Resolved
Write a reply...
Reactivate

Fig. 2. Nudge workflow.

Nudge Workflow. The three modules are combined with a notification system to form Nudge as shown in Figure 2. This results in the following workflow:

(1) The Nudge service workflow starts with calculating the effort needed for a pull request using effort estimation models. When the corresponding batch job is triggered, it first scans all active pull requests and runs the effort estimation model (see Section 5) to determine the lifetime of a pull request and save it to a back-end SQL database. The batch job is triggered every six hours.
(2) Once a pull request's actual lifetime crosses the estimated lifetime (using the effort estimation models), the next module, Activity Detection, is run, which checks for any activity in the pull request environment. If there is an activity observed in the last 24 hours, then the workflow is terminated.
(3) Once the activity detection algorithm determines that there was no activity in the last 24 hours, the Actor Identification algorithm kicks in which determines the change blockers and dependant actors who should take appropriate actions to facilitate the movement of the pull requests.
(4) Finally, notifications are sent to the list of actors identified in the previous step in the form of pull request review comments and email messages. By design, Nudge sends at most one notification per pull request.

4.2 Key Design Considerations

Feature Extraction. Nudge's machine learning model needs to extract various features for every new pull request to perform inference. There are three classes of features that constitute the feature vectors:

(1) Some features are easy to extract and are readily available in the pull request. Examples include the day of the week, the length of its description, and so on.
(2) Some features can be computed based on the information available in the pull request. An example is whether the pull request is a new feature or a bug fix. For this, we run the relevant heuristic algorithm to classify the pull requests accordingly.
(3) Some features are hard to calculate on the fly as they require mining historical data. Examples include the average lifetime of the pull requests created by an author or the average lifetime of the pull requests that edit specific area paths in the source code. For these, we compute their value through a batch job that runs at a scheduled frequency (every 6 hours) and that stores the results in a database. Upon inference, the pre-computed features are queried from the database and appended to the other two types of features to form the feature vectors.

Scale. Nudge has two primary scale challenges it has to deal with. The first is conducting feature extraction, training, and re-training for over 100 repositories. The second challenge is inference and sending notifications on live pull requests created in these repositories. To deal with the first problem, we adopted a strategy to train the model by pre-computing some of the features beforehand, when the data themselves are ingested. This helped in reducing the overall training time. The second strategy we adopted is to not train and build repository-specific models if there are not enough training data. While this primarily helps us in increasing the model's accuracy and efficiency, it also has the effect of reducing the load on the training and retraining pipeline. We have implemented Nudge using a map-reduce-based big data platform that will enable us to scale to 1,000s of repositories in the future.

Notification Presentation. We experimented with several versions of the notifications. The most verbose explained what features the model looks at, what the estimated lifetime for the pull request is, and why we are nudging at a given point in time. A less verbose version just says, "This pull request has been open since N days. Please take appropriate action." We also experimented with the format, icon, color, and so on. We experimented with the different designs of the notification by letting real users try them. Eventually, this helped us come up with a notification that is liked and approved by the end-users.

Feedback Collection. To enable ourselves and repository owners to monitor and evaluate Nudge's usage and impact, we include a feedback collection mechanism. We rely on thumbs up/down feedback as well as optional text left by pull request authors and reviewers. A collection pipeline scrapes this feedback automatically per repository. We also built an internal reporting tool with a dashboard that displays the feedback at the repository level as well as globally and that is refreshed automatically when the numbers are updated.

5 PULL REQUEST LIFETIME PREDICTION

To be able to "nudge" developers on overdue pull requests, the Nudge algorithm, first, needs to determine the expected lifetime of the pull request. In this section, we explain the details of how the data needed to train the lifetime prediction models at pull request level are mined and how the model is developed, validated, and deployed. We can broadly classify this activity into the following three steps:

(1) Leveraging the rich history of prior work done in effort estimation software repository mining to determine the factors that impact pull request acceptance and defect prediction (see References [14, 15, 18, 48] as discussed in Section 2), we identify a set of attributes that needs to be mined for pull requests.
(2) We collect data for these selected attributes on multiple repositories, as well as the actual pull request lifetime data, to establish a training dataset.
(3) We use the training dataset collected in Step (2) to build a pull request lifetime prediction model and evaluate the performance of the model.

5.1 Correlation Analysis

We performed correlation analysis to understand the factors that are associated with the lifetime of pull requests and the magnitude of the association. We collected 22,875 completed pull requests from 10 different repositories at Microsoft. These repositories host the source code of various medium and large-scale services with hundreds to thousands of developers working in those repositories. We omit any pull requests whose age is less than 24 hours (short-lived pull requests) or more than 336 hours (2 weeks, long-lived pull requests). The reason for omitting short-lived pull requests is that they do not need to be nudged. We omit long-lived pull requests as they are outliers, and we do not want the model to learn from poorly handled pull requests that took too long to complete.

We formulated this as a regression problem where we define a dependent variable (pull request lifetime) and a set of independent variables (the 28 features listed in Table 1). We then used a gradient boosting regression algorithm to perform the regression analysis and calculate coefficients (listed in Table 1). The dependent variable in our experiment is the pull request completion time, i.e., the time interval between pull request creation and closing date, in hours. We exclude the 48 weekend hours from the total completion time to make the experiment reflect the real-world deployment scenario where Nudge notifications are not sent on weekends. The features we use in our experiment are related to the pull request itself, the author, the process, and churn. Table 1 lists all the features we use including their correlation to pull request completion time.

Of the 28 features, the four that contribute most to a pull request's lifetime include the following:

Day of the week. This is the day of the week on which the pull request is created. We represent Sunday with 0 and Saturday with 6. A strong positive correlation with this metric indicates that pull request created later in the week are taking more time to complete. Pull requests created toward the end of the week stay idle during the weekend, but, optimistically, reviewers will start to act on them on Monday. We represent days toward the end of the workweek with higher values and check if this affects completion time.

Average duration of pull requests created by the author. This captures how quickly a specific author's pull requests were moving, historically. Developers new to a particular repository or project may take more time to learn the processes followed in the repository. Their changes might be subjected to more thorough reviews and testing that potentially delays the progression of their pull requests. Over time these developers may become faster in completing their pull requests.

Number of reviewers of the pull request. If more people are actively reviewing a pull request and are engaged with it, then more comments and questions are raised. Some teams in Microsoft have policies that mandate the comments to be closed before completing pull requests. So the pull request author has to go through the review comments manually and either agree and resolve them or disagree with them.

Is a .csproj file being edited. A .csproj file in C# is a crucial project configuration file that tracks files in the current project, external package dependencies and their versions, dependencies among different projects, and so on. Modifications to these files tend to indicate a major activity or structural change in the project. That includes adding or deleting files, modifying external dependencies or libraries, bumping up the versions of the dependent libraries or packages, and so on.

The four features that help most reduce the lifetime of a pull request include the following:

Is the pull request a bug fix? In large-scale cloud service development environments at Microsoft, fixing bugs is prioritized. Incident management processes help in expediting such bug fixes that result in faster completion of pull requests. We used the models developed by Wang et al. [46] to determine the intent of the pull requests. These are language models that analyze the pull request title and description to classify the intent. We used these Random Forest models to compute this intent feature along with other features (whether the pull request is deprecating old code, whether the pull request is performing refactoring). This helps account for the semantic intent of the pull request in the lifetime prediction model.

Age of the author in the team. This feature captures how familiar a developer is with the current team, its processes, people, and the product or service the team is working on. The more time a developer spends in a team, the less difficulty they will experience in pushing their change through. We get this information from the human-resources database at Microsoft.

Table 1. Feature Description and the Correlation between Features and the Pull Request Lifetime (Sorted in the Descending Order of Correlation)

Feature Description	**Type**	**Corr.**
The day of the week when the pull request was created	Categorical	0.163
The average time for pull request completion by the developer who initiated it	Continuous	0.159
Total number of required reviewers on the current pull request	Discrete	0.131
Is .csproj file being modified?	Categorical (Binary)	0.103
The average time for completion for the pull requests that have the same project paths changed	Continuous	0.089
Total number of distinct file types that are being modified	Discrete	0.084
The word count of the textual description of the pull request	Discrete	0.072
Is the pull request modifying any config. files or settings	Categorical (Binary)	0.059
Number of active pull requests in the repository	Discrete	0.058
Churned LOC per class	Discrete	0.055
Total churn in the pull request	Discrete	0.039
Number of methods being churned	Discrete	0.037
Is this pull request introducing a new feature?	Categorical (Binary)	0.033
Number of lines changed	Discrete	0.031
Number of distinct paths that are being touched in the current change	Discrete	0.031
Number of conditional statements being touched	Discrete	0.029
Number of loops being touched	Discrete	0.028
Number of classes being added/modified/deleted	Discrete	0.021
Is the PR doing any refactoring of existing code?	Categorical (Binary)	0.021
Number of references or dependencies (on other libraries/ projects) being changed	Discrete	0.017
Number of files that are being modified in pull request	Discrete	0.016
Is the pull request making any merge changes like forward or reverse integration (FIs/RIs)	Categorical (Binary)	0.008
Is the pull request deprecating any old code?	Categorical (Binary)	−0.001
The word count of the textual title of the pull request	Discrete	−0.001
Whether the pull request is created during business hours or off hours?	Categorical (Binary)	−0.019
Is the pull request fixing bugs?	Categorical (Binary)	−0.028
Time spent by the developer in the current team.	Continuous	−0.031
Time since the first activity in the repository by the pull request author	Continuous	−0.046
Time spent by the developer at Microsoft	Continuous	−0.056

Age of the author in the repository. This helps capture the familiarity of a developer with the repository in which they are making changes, and the build, and deployment processes of that repository. Although this may sound similar to the author's age in the current *team* just discussed, familiarity with repositories may vary substantially in heterogeneous teams that work on multiple services (especially, microservices). Here, different members of the same team are mostly making changes that are very specific to the repositories they are actively engaged in. Our correlation analysis has shown that the more familiar a developer

Table 2. Comparison of Different Prediction Models

Algorithm	MAE (in hours)	MMRE
Least squares	44.32	0.68
Bayesian ridge	46.35	0.71
Gradient boosting	32.59	0.58

is with a specific repository, the less time it takes them to merge their changes made in that repository. We compute this based on when the author created or reviewed the first pull request in a given repository.

Age of the author in Microsoft. This helps capture the seniority of a developer. Intuitively, senior people who have more experience tend to make fewer mistakes and will experience less pushback on their changes. The negative correlation here indicates that if someone has more experience, then it takes them less time to merge their changes. We get this information from the human resources database at Microsoft.

5.2 Prediction Model

As indicated, we approach the task of predicting the lifetime of a pull request as a regression problem. We include most of the features from Table 1, dropping the ones with a very low (absolute) magnitude of correlation. We used 0.008 as a cutoff, thus dropping three features. This helped to speed up the training and inference tasks without materially impacting the **Mean Absolute Error (MAE)** (it dropped by 10 minutes (0.17 hours)).

We then performed an offline analysis and evaluation with multiple popular regression algorithms like least-squares linear regression, Bayesian ridge regression, and gradient boosting. To compare the regression algorithms, we used two standard metrics: MAE, and **Mean Magnitude of Relative Error (MMRE)**. These metrics are widely used for understanding the performance of regression tasks. We decided to adopt gradient boosting, as it has better accuracy with respect to both MAE and MMRE. The comparative analysis of the three algorithms, evaluated against MAE and MMRE is shown in Table 2. A detailed discussion on prediction accuracy and its significance in the context of the application we are building is presented in Section 8.

We are not using the prediction outcome (the expected pull request lifetime) for performing traditional effort estimation tasks, such as sprint or project planning or budgeting. In the case of the Nudge system, the primary purpose of the model is to approximate the opportune moment to send a reminder. Therefore, the Nudge system exhibits more tolerance toward the prediction error.

To make sure all the features reflect recent trends, we use the pull request data from the past 2 years each time the model is trained.

For training and evaluation, we use scikit-learn.[1] We used a standard 10-fold cross-validation. We followed the standard practice of one time 10-fold cross-validation [4], without "repeated cross-validation," as follows:

1. we separate the dataset into 10 partitions randomly;
2. we use one partition as the test data and the other nine partitions as the training data;
3. we repeat Step (2) with a different partition than the test data until all data have a prediction result;
4. we compute the evaluation results through a comparison between the predicted values and the actual values of the data.

[1] https://scikit-learn.org/.

6 PULL REQUEST STATUS DETERMINATION

With a mechanism in place to predict the lifetime of a pull request, the next step is assessing whether there has been any activity or state changes that are taking place in a pull request. This serves to determine the opportune moment to send a notification as well as to understand when *not* to send a notification. To do so, we determine the current *activity* and *blocking actor* in terms of the pull request lifecycle model as displayed in Figure 1,

6.1 Activity Detection

Using an earlier version of Nudge, we conducted a quantitative study to understand the impact of not reacting to the activity in a pull request while sending notifications. We found, through manual inspection, that 86 of 119 Nudge comments that are resolved negatively were due to the fact that Nudge did not honor the recent pull request activity. Later, we talked to some of the developers who were either authoring or reviewing those pull requests. The majority of them did not like the Nudge notifications, because they recently interacted with the pull request.

To resolve this problem, Nudge determines the most recent activity in the pull requests. However, pull requests in large organizations can get complex with multiple actors performing different activities through various collaboration points. We distinguish the following collaboration points that trigger the changes to the pull requests (see also Figure 1):

Pull request state changes. A state change in a pull request strongly indicates that one of the actors (author or reviewer) has been acted on the pull request recently.

Comments. Once a pull request is submitted for review, reviewers can add comments to recommend changes or seek clarification on a specific code change. Authors of the pull request can also reply to the comment thread that is started by the reviewers if they have any follow-up questions. In addition to placing the comments and replying to them, the actors can also change the status of the comments. Typical statuses are "Active," which means the comment has just been placed; "Resolved," which means the comment has been resolved by the author of the pull request by making the changes prescribed by the reviewers; "Won't fix," which means the author would like to discard the review recommendation without addressing it; and "Closed," which means the comment thread is going to be closed, as there are no more follow up action items or discussions needed.

Updates. After a pull request has been created, authors can keep pushing new updates in the form of commits. These commits are changes that authors are making in response to review recommendations or improvements the authors themselves decided to push into the pull request. Under some special circumstances, someone other than the author or the reviewer can also push new updates into a pull request but that is a rare occurrence. New updates or iterations are a very strong indicator that the author is making progress on the pull request.

The specific action points may vary depending on the provider of the source control system (GitHub, Azure DevOps, GitLab, etc.). However, conceptually the collaboration points or concepts remain similar. In the context of this work, we focus on Azure DevOps, the source control system used by Microsoft's developers and offered by Microsoft to third-party customers. We track the activities performed through these collaboration points to determine the existence of activity in pull requests and decide whether a Nudge notification should be sent.

Nudge typically sends a notification once the lifetime of a pull request crosses the predicted lifetime (as predicted by the lifetime prediction model). However, it waits at least 24 hours before sending a notification if there has been any activity since last checked (Nudge pipeline runs once every six hours; details about the pipeline are explained in Section 7) or when state transitions have been observed. Based on a user study described in Section 8, we find that activity detection improves user experience, reduces false alarms, and thus increases the usefulness of the Nudge service.

Table 3. Classes That Explains Change Blockers and Responsible Actors

State	Class	Actor waiting for	#PRs
Waiting	Not all review comments have been addressed	Author	34
Waiting	Pull request needs further discussion	Author	47
Approved	Pull request has been approved but the author is not ready to merge it	Author	49
Active	Review has not been started yet	Reviewer	51
Active	Review comments have been addressed but reviewers have not approved yet	Reviewer	19

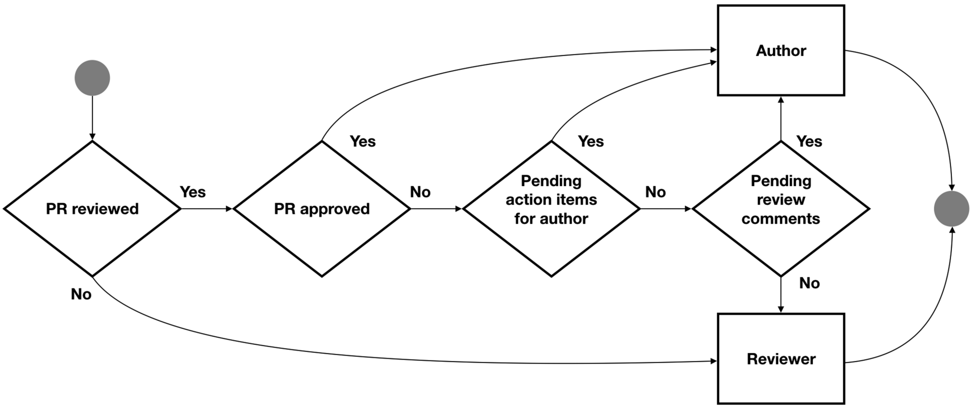


Fig. 3. Flowchart to determine the change blockers for active pull requests.

6.2 Actor Identification

In a pull request, there are different actors involved (as explained in Section 3) that can influence the next state (Approved, Rejected, etc.), and the speed with which a pull request progresses.

We focus on understanding the human change blockers, i.e., authors and reviewers, and the extent to which they influence the change progression. We collected 200 pull requests from 20 medium to large to very large repositories and manually analyzed them to understand for whom they were waiting before they were completed. These are pull requests whose age is at least 14 days and that have not been completed yet. We find there are five mutually exclusive classes that explain the cases in which a pull request is awaiting completion. Table 3 lists the classes and the actor responsible, and the number of pull requests that fall under each class. Seventy pull requests (of 200) are blocked by the reviewers while the remaining 130 are waiting for the author to make progress.

Encouraged by the findings, we devised an algorithm that helps determine the actor that needs to be notified to make progress on a given pull request. When there is an action item pending on the author of the pull request as well as a reviewer, sending notifications to the author is prioritized. The flow chart shown in Figure 3 explains the control flow and how the actors that are responsible for making progress on the change are determined. The algorithm evaluates various decision points to determine the blockers of a change. These decision points represent different states that a pull request, review, or reviewing comments in the pull request take during the lifetime of a pull request. There are three cases where the author needs to act:

PR is approved. A pull request is approved when the reviewers are satisfied with the changes and have no more comments or concerns about the change. The author can proceed to merge the change.

Not all review comments are addressed. The reviewer has left comments seeking some clarity or proposing recommendations. The author is responsible to address the review comments. Authors typically will have two choices: If they agree with the review comment, then they can resolve it, or if they disagree, then they can mark it as "won't fix." This condition is met if the author has review comments that need to be addressed.

Author has pending action items. The author has addressed the review comments, but the reviewer does not want to approve the changes, because they are not satisfied with the resolution provided by the author. These pull requests need further discussion.

In the remaining cases, the reviewers have to act on the pull request to unblock the change as follows:

Review has not started. Upon creating the pull request, authors typically add the reviewers that they would like to get a review from for the specific change. The reviewers are supposed to act on it and provide their comments. If the reviewers are not acting on the pull request after requesting a review, then the onus is going to be on the reviewers to act on the pull request and unblock it.

Review comments are addressed. Once the reviewer has provided their review, the author will act on it and resolve/won't fix the comments by making necessary changes. Then the responsibility shifts back to the reviewer to re-verify the changes and sign off the change. If that is not happening, then reviewers are accountable and should be notified to unblock the change.

7 IMPLEMENTATION

In this section, we present the details about how the Nudge Service is implemented. It relies on Azure DevOps, the git-based DevOps solution offered by Microsoft, which we used to deploy Nudge as an extension.

7.1 Nudge Service Architecture

Figure 4 shows the Nudge service architecture and gives an overview of various components involved. Azure DevOps is the existing git environment, which is connected to a collection of workers hosted on Azure. Listed below are the seven steps (the numbered arrows in the figure) that explain the high-level architecture and interaction between various components in the Nudge system:

(1) A developer creates a pull request or updates an existing pull request by pushing a new commit or iteration into it.
(2) A pull request creation or update event is triggered through the service hook.
(3) A new message is sent to the Azure service bus where it is queued.
(4) An Azure worker picks up the new messages on a first come first serve basis.
(5) The workers run effort estimation, activity detection, and actor identification, storing the results in a database.
(6) Based on the outcome of Step (5), if the Nudge system decides to send a notification, then the worker sends a notification using the Azure DevOps APIs in the form of pull request comments.
(7) Azure DevOps sends a notification email to the developer.

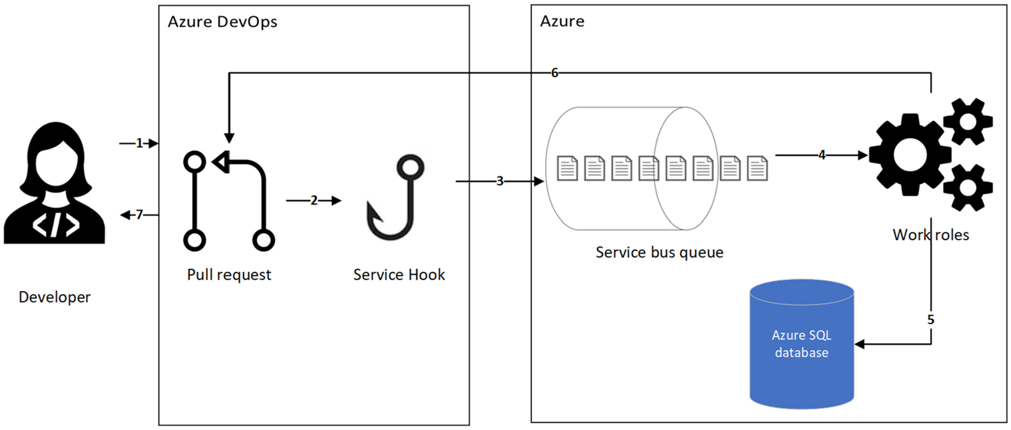


Fig. 4. Nudge architecture.

Nudge performs inference and recalculates the effort each time a pull request is updated (Step (2)). PR updates can change the structure of the PR completely (adding/deleting code changes, adding/removing reviewers, etc.), making it important to react to them and adjust the pull request lifetime prediction accordingly.

Additionally, if the criteria to send a notification are not met for a given pull request, then the Nudge system will check again in six hours, through an Azure batch job, to determine if it can send a notification. The Nudge service continues to do that, every 6 hours, until the pull request is abandoned or completed.

7.2 Azure DevOps

Azure DevOps is a platform providing a git-based version control system. In addition to repositories, it offers planning tools such as work item and bug report management and facilitates code review management. It also has features such as build and releases management to facilitate continuous integration and deployment. The Nudge service is deployed as an extension of Azure DevOps because of the rich collaboration features offered by Azure DevOps. Below are the details about some of the key features that Azure DevOps offers that helped materialize the Nudge service:

Collaboration points: Azure DevOps offers a rich set of collaboration points through which third-party services or extensions can interact with pull requests in Azure DevOps. The collaboration points allow services to add comments on pull requests, add labels to the pull requests, and add or remove reviewers.

Service hooks: Azure DevOps offers service hooks that help any third-party service to listen to the events that are happening inside the pull request environment. Events can be pull request creation events that are fired through service hooks when a pull request is created or pull request update events that are fired when the pull request experiences any updates such as pushing new commits or iterations.

APIs: Azure DevOps exposes a rich set of REST APIs [1] that helps third-party services to access information about various artifacts in the Azure DevOps environment. These APIs can be called through a REST client and return metadata about the pull requests (id, title, author, reviewer information, comments, labels, status, commits that are included in the pull request), commits (title, files changed in a commit), build, and release (status, test outcomes, deployment outcomes).

Votes: Azure DevOps uses a voting mechanism to capture the actions performed by the reviewers on a pull request. A vote on a pull request can have values $\{-10, -5, 0, 5, 10\}$, corresponding to rejected, waiting for the author, no vote, approved with suggestions, and approved, respectively.

7.3 Activity Detection

We use Azure DevOps's REST APIs [1] to collect data that are required to understand if there has been any activity in a pull request. We gather data about various actions or activities that happen inside a pull request (Section 6.1) to determine if there has been any activity as follows:

Commit activity: We use Azure DevOps's `GetPullRequestIterationsAsync` API, which provides details about all the commits that are ever pushed into a pull request. We first get a list of all the commits that are pushed and then take the timestamp of the latest iteration as the latest commit timestamp of a pull request.

Comment activity: To determine whether there has been any commenting activity like adding new comments or replying to existing comments, we use Azure DevOps's `GetThreadsAsync` API. This API returns all the comments that are ever placed in a pull request in the form of threads. We check if any new threads are created or if any new comments are placed in an existing thread. We take the maximum of both of them to determine the latest comment activity that has happened in a pull request. While doing this we exclude any comments that are placed by system accounts or non-human actors following basic heuristics, such as accounts that include words like "system," "bot," "account," and so on.

State changes in pull requests: Changes in pull request state is another important signal that helps determine activity in a given pull request. Unfortunately, there is no direct way of determining state changes in pull requests. We use Azure DevOps's `GetThreadsAsync` API to collect all the comments placed in a pull request. Comments whose content property contains the word "voted" indicates that a state change has happened. Azure DevOps uses a voting mechanism to capture the actions performed by the reviewers on a pull request. A voting event in a pull request looks like the following: "`User1 voted 10 on PR1234`," which, as explained above, corresponds to approval. We use such events to determine the last time a pull request's state has changed.

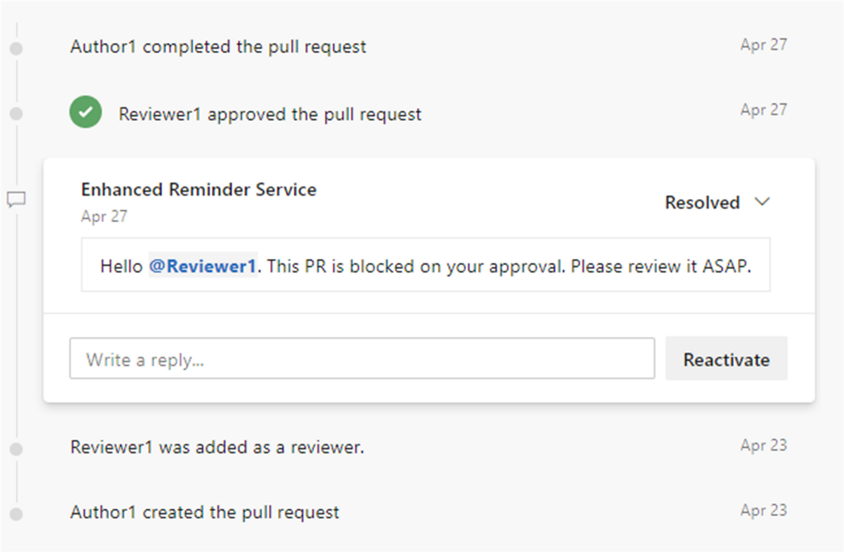
Nudge sends a notification once the lifetime of a pull request crosses the predicted lifetime (Section 5). However, it waits at least 24 hours before sending a notification if there has been any activity observed.

7.4 Actor Identification

We rely on Azure DevOps's REST APIs to collect data for identifying the actors. In line with Section 6.2, we use Azure DevOps as follows:

Check pending action items: To determine if pull request's author has any pending action items, we check if the state of the pull request is set to "Waiting on Author." We use Azure DevOps's `GetPullRequestReviewersAsync` API to get the votes of all the reviewers.

Check for existence of unresolved comments: The existence of unresolved comments determines whether the blocker of a pull request is the author or the reviewer. We use Azure DevOps's `GetThreadsAsync` API to get all the threads. We then check for the existence of threads with statuses "Active" or "Pending." The presence of threads with any of these two statues indicates that there are unresolved comments.



Author1 completed the pull request Apr 27

Reviewer1 approved the pull request Apr 27

Enhanced Reminder Service
Apr 27

Resolved

Hello @Reviewer1. This PR is blocked on your approval. Please review it ASAP.

Write a reply...

Reactivate

Reviewer1 was added as a reviewer. Apr 23

Author1 created the pull request Apr 23

Fig. 5. Nudge notification, with @Reviewer1 tagged in the reminder.

Enumerate the list of change blockers: We first use the `GetPullRequestReviewersAsync` API offered by Azure DevOps to query the list of reviewers on a given pull request. We then use the `GetThreadsAsync` API to determine the list of all the reviewers who commented on the pull request at least once and whose comments are resolved by the author of the pull request. We prepare two lists (reviewers who commented and all reviewers) and choose one of them to use based on the state of the pull request. If there are no reviews on a pull request, then we send notifications to the reviewers in the "all reviewers" list. If there has been a review activity (reviewers placed comments on the pull request), then we prioritize notifications to the reviewers in the list of reviewers who commented.

7.5 Nudge Notification

Figure 5 shows the screenshot of the Nudge notification. Note that the dependent actor (in this case the reviewer but not the author) is being "@-mentioned" in the notification. This triggers a separate email to the reviewer of this pull request asking them to unblock the pull request. As we can notice, the pull request was created and had been waiting for the reviewer's approval for four days. After the Nudge service tagged the reviewer and pushed them to act on the pull request, the reviewer approved it and the pull request got completed on the same day.

8 EVALUATION

In this section, we describe (1) the experiments we conducted to assess the value of a pull request level effort estimation system, (2) the value of a system like Nudge that leverages the effort estimation models to notify developers about their overdue pull requests, (3) the impact Nudge has on large development teams and organizations, and (4) the scale at which a system like Nudge can operate. This is reflected in the following research questions:

RQ1. What is the accuracy of effort estimation models in predicting the lifetime of pull requests?
RQ2. What is the impact of a service like Nudge on completion times of pull requests?
RQ3. What are developers' perceptions about the usefulness of the Nudge service?

RQ4. Can the deployment of Nudge be scaled to thousands of repositories without sacrificing gains in pull request processing time and user perception?

8.1 Data Collection and Methodology

We obtained data from the large-scale deployment of the Nudge service for 9 months on 147 repositories in Microsoft. The data include telemetry from the Nudge service using only lifetime prediction as a mechanism (which we will refer to as Nudge-LT), as well as from the Nudge service extended with activity and actor identification (which we refer to as Nudge-FULL or just Nudge). The repositories are owned by various product and service teams and are of different sizes, geographies, and products. Nudge has made notifications on 8,500 pull requests during the 9-month time window under study. We discuss the results from Nudge-LT first and subsequently analyze the effect of the additional heuristics of Nudge-FULL.

For RQ1, we collect historical data from pull requests that are merged. This gives us the start and end timestamps to help us calculate the lifetime for each pull request and construct a ground truth dataset. We collected 2,875 pull requests from 10 different repositories that have been merged and completed. These repositories host the source code of various services. Their number of contributing developers ranges from a few hundred to a few thousand. This dataset is independent of the data used to train the model (as explained in Section 5.2). As for the correlation analysis (Section 5.1), we omit any pull requests whose age is less than 24 hours (short-lived pull requests) and more than 336 hours (long-lived pull requests).

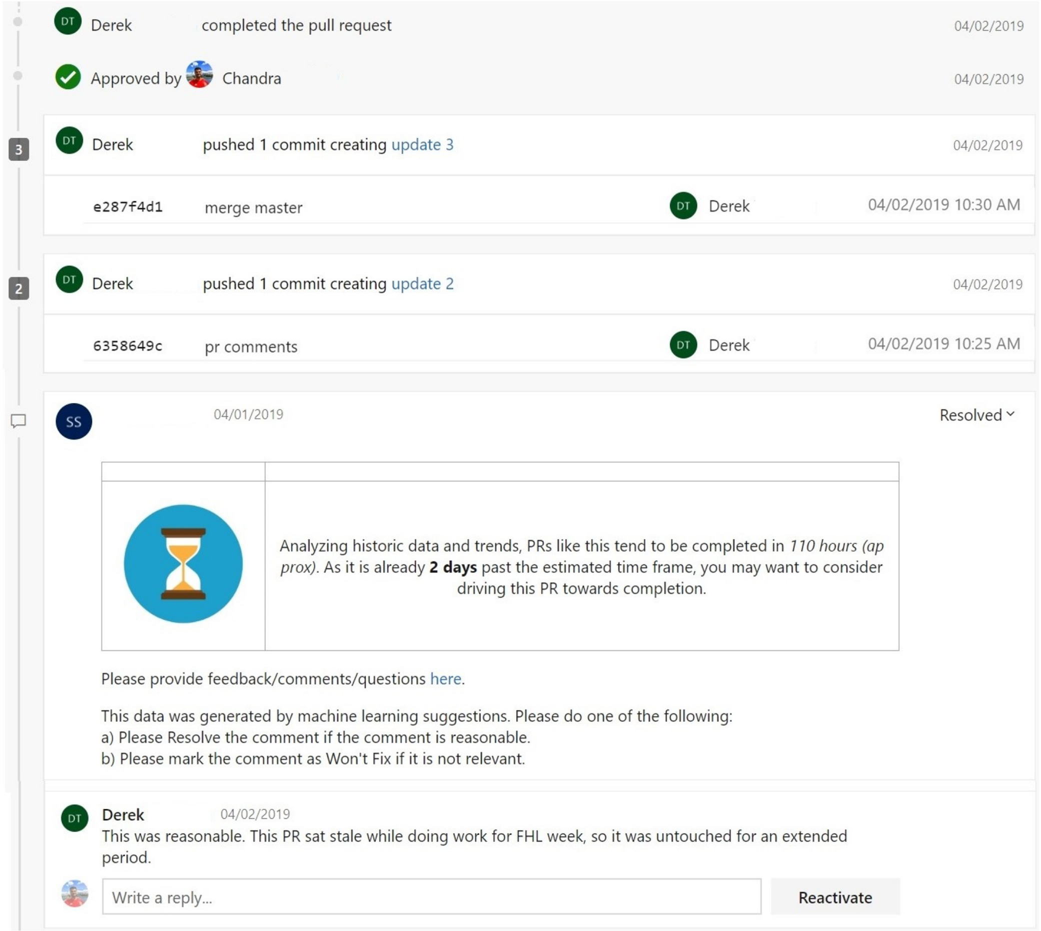
For RQ1, we also use repositories on which we operationalized Nudge to obtain feedback from developers on the estimations. We randomly select pull requests for which we are about to send a Nudge notification and add more details in the notification comment. These are details like Nudge model's predicted lifetime for a given pull request, how long the pull request has been open past the estimated lifetime by the Nudge model. Figure 6 shows the details about the predicted lifetime of a sample pull request, as predicted by the Nudge model, and the reason for sending the notification at a given point in time.

For RQ2, we collect data from the 147 repositories on which we operationalized Nudge. We collect data on how the lifetime of pull requests is varying between pull requests that received a Nudge notification and pull requests that did not. We also collect data about the time it takes for the author of the pull request to either complete or abandon the pull request after a Nudge notification is sent.

For RQ3, we collect data through our automated pipeline that actively tracks every single inline reply that is posted by the developers in response to a Nudge notification and whether they positively or negatively resolved a comment. We do this for all 8,500 pull requests on which we made notifications.

For the 147 repositories on which we deploy Nudge, notifications are sent when a pull request meets the criteria needed to be nudged, as imposed by the Nudge model and algorithm. All developers who receive a Nudge notification are given equal opportunity to provide feedback in the pull request, either to positively or negatively resolve the comment or to provide anecdotal feedback by replying inline to the Nudge notification. Note that the repositories on which Nudge has been operationalized are organizationally away from the developers of the Nudge service. The notifications did not reveal the names or identities of the developers of the Nudge service to avoid response bias [19].

For RQ4, we took advantage of the fact that our initial experiments convinced Microsoft management to deploy Nudge in production. This enabled us to monitor Nudge in production at Microsoft during the period January 2021 until December 2021. During this period, Nudge was deployed on 8,000 different systems at Microsoft. Nudge sent 210,000 notifications authored by 40,000 unique



Derek completed the pull request 04/02/2019

Approved by Chandra 04/02/2019

Derek pushed 1 commit creating update 3 04/02/2019

e287f4d1 merge master Derek 04/02/2019 10:30 AM

Derek pushed 1 commit creating update 2 04/02/2019

6358649c pr comments Derek 04/02/2019 10:25 AM

04/01/2019

Analyzing historic data and trends, PRs like this tend to be completed in *110 hours (approx)*. As it is already **2 days** past the estimated time frame, you may want to consider driving this PR towards completion.

Please provide feedback/comments/questions here.

This data was generated by machine learning suggestions. Please do one of the following:
a) Please Resolve the comment if the comment is reasonable.
b) Please mark the comment as Won't Fix if it is not relevant.

Derek 04/02/2019
This was reasonable. This PR sat stale while doing work for FHL week, so it was untouched for an extended period.

Fig. 6. A pull request with a lifetime prediction notification in Azure DevOps.

developers. We collect pull request completion time after notification, as well as the positive/negative resolutions of pull request recommendations made by Nudge.

8.2 RQ1: What Is the Accuracy of Effort Estimation Models in Predicting the Lifetime of Pull Requests?

To answer this research question, we collect metrics that explain how accurate our prediction model is. We also list the anecdotes we received from the developers about the accuracy of the prediction model.

Model Evaluation. We evaluated our prediction model against standard metrics: MAE and MMRE. For the pull request level effort estimation model, the MAE is 32.60 hours (Figure 7 shows the distribution of MAE) and MMRE is 0.58 (Figure 8 shows the distribution of MMRE).

To put these numbers in perspective, we have conducted an experiment by considering the mean lifetime of our training data as the predicted lifetime of every pull request in our testing data. Our constant model's MAE is 36.43 hours and MMRE is 0.68. This means our trained model is 11.8% better in terms of MAE and 17.7% better in terms of MMRE compared to the constant model.

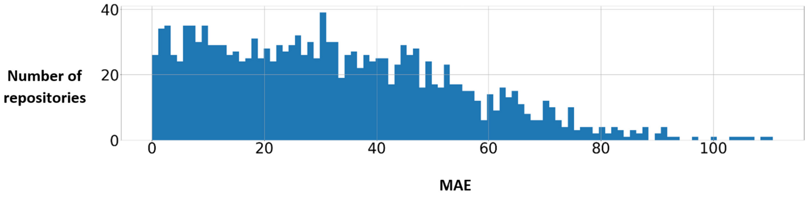


Fig. 7. MAE distribution.

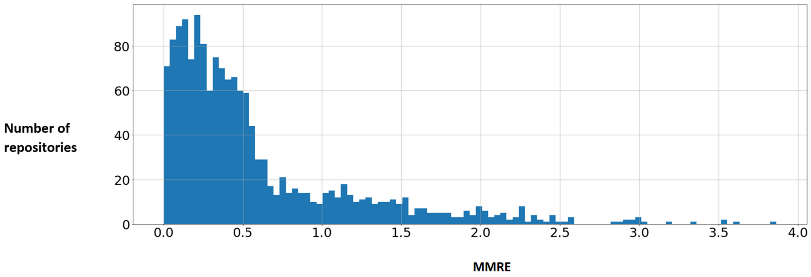


Fig. 8. MMRE distribution.

The MAE of 32.60 hours corresponds to around 1.3 days. The average duration is 107.63 hours or a little over 4 days. For our purposes, for warning developers when they are late, we consider an average deviation of around a day to be acceptable.

User Feedback about Model's Prediction Accuracy. We received positive feedback from the developers of the randomly selected pull requests for which we added more details about the model prediction as illustrated in Figure 6. One of the developers said:

> *This was reasonable. This pull request sat stale while doing work for FHL, so it was untouched for an extended period.*

Here, the developer is acknowledging that the model's prediction (110 hours) is reasonable and, noticeably, provides an explanation for why the pull request is taking long to wait. As we see in Figure 6, the developer ended up completing the pull request within a few hours after the notification was sent. Similarly, another developer said,

> *I totally agree with the model saying this pull request should take not more than 120 hours to complete. The code change is slightly complex and the estimation seems reasonable.*

In this case, the developer is positive about the fact that the model is predicting the lifetime by taking into account the complexity of the change and giving enough breathing room for the developers to act on it before nudging them. Another developer passed feedback by acknowledging the fact that the model adapts to the changes happening inside the pull request by comparing two of her pull requests,

> *I see the estimation is 176 hours on this pull request, and it was 64 hours on another pull request of mine where I was editing a lesser number of files and not pushing critical code changes. I do not know if your model is taking these facts into account. But, it seems, like, . . . interesting!*

This anecdote supports the fact that the model adapts to the pull request in question and the users starts to notice that the model is doing a reasonable job in adapting to the change in context.

Table 4. Comparison of Average Pull Request Lifetime (Hours)

Service	Avg PR lifetime	Number of PRs
None	197.2	3,856
Nudge-LT	112.6	4,117
Nudge-FULL	77.7	4,383

8.3 RQ2: What Is the Impact of Nudge Service on Completion Times of Pull Requests?

To measure the impact of Nudge, we use two metrics to assess whether the Nudge service is helping developers and yielding the intended benefit:

(1) Average pull request lifetime: This is the average of the time difference (in hours) between pull request creation and closing date. A service like Nudge is expected to introduce positive effects like reduction in pull request lifetime by notifying the change blockers about making progress and closing the pull requests.
(2) Distribution of the number of pull requests that are completed within a day, in 3 days, within a week, and after a week since Nudge sent a notification. This captures to what extent developers are actually reacting to the Nudge notifications, and, if so, how quickly they are reacting.

While measuring and comparing the metrics above, we make sure to nullify the effects of other variables such as the month of the year (changes move faster in some months and slower during some), typical code velocity in a given repository (some repositories naturally experience higher development velocity because of the nature and critically of the service), team or organization culture (some teams typically are more agile and ship things faster), and so on. Therefore, if we compare pull requests from two different repositories or from two different time periods, then we cannot confidently say whether an increase or decrease in average lifetime is due to the presence or absence of the Nudge service or due to other factors explained above. To remedy this, we set up a randomized trial (A/B, or in fact A/B/C testing) by randomly selecting one of the three configurations listed below for each pull request:

None: Turn the Nudge service off for a set of randomly selected pull requests.
Nudge-LT: Turn on the basic version of the Nudge service with just lifetime prediction but without user identification and activity detection.
Nudge-FULL: Turn on the user identification and activity detection features along with the effort estimation model in the Nudge service.

Table 4 displays the average pull request lifetime for each of these configurations. We see a clear decrease in an average lifetime for the pull requests for which Nudge notifications are sent. The average lifetime of the pull requests on which Nudge notifications are sent is 112.6 hours, which is a 42.9% decrease compared to the set of pull requests on which we did not send the notification (where the average lifetime is 197.2 hours). Actor identification and activity detection further brought the average lifetime down to 77.7 hours, which is a reduction of 60.62% in average pull request lifetime.

In Figure 9, we plot the distribution of pull requests that are completed within a working day, 3 days, a week, or more than a week after Nudge sent the notification. Only 1,570 pull requests of 8,500 pull requests (18.47%) have taken more than a week to close. 81.53% of the pull requests are closed within a week. An important observation to make is that 2,300 pull requests, i.e., 27.05% of the pull requests on which Nudge sent the notification were completed within a day. This distribution indicates that the majority of the pull requests on which Nudge sends notifications are completed relatively quickly.

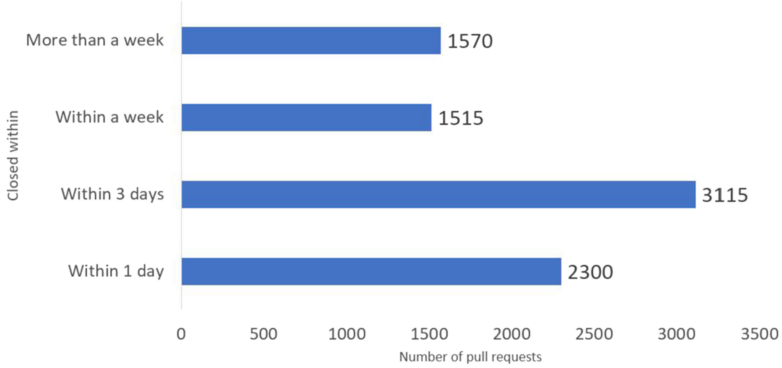


Fig. 9. Distribution of completed pull requests after sending a notification.

Table 5. The Difference in Percentage of Positively Resolved Notification

Service type	# Positive responses	# Negative responses	# Total responses	# No responses	# Total PRs
Nudge-LT	1829	2062	3891	226	4117
Nudge-FULL	3199	882	4081	302	4383

8.4 RQ3: What Are Developers' Perceptions about the Usefulness of the Nudge Service?

To understand whether users are favorable toward the Nudge system, we pursue a mixed-methods approach. To that end, we rely on two sources of information:

- For every Nudge notification that is sent, the developers have an option to perform one of the following three actions: positively resolve the notification (by marking it as “resolved”), negatively resolve the notification (by marking it as “won’t fix”), and provide no response.
- Second, Nudge users can enter an inline reply within a Nudge notification to explain their (dis)satisfaction.

We again distinguish between Nudge-LT and Nudge-FULL.

8.4.1 Notification Resolution. Table 5 shows the number of positive and negative reactions to notifications, both for Nudge-LT and Nudge-FULL. For the vast majority (93–97%) the developers actually provided an explicit verdict.

For Nudge-LT, the majority of verdicts (2062/3891, 53%) were negative. This suggests that nudges based on lifetime predictions alone are not considered sufficiently helpful.

For Nudge-FULL, by contrast, the vast majority of verdicts (3199/4018, 80%) were positive. When also including non-responses in the total, the percentage of positive resolutions remains high, at 73% (3199/4383). This makes it clear that the activity detection and actor identification of Nudge-FULL clearly contribute to the positive perception of Nudge.

Note also that positive feedback of 73% is substantial if we look at it in isolation. Various studies have shown that users tend to provide explicit negative feedback when they do not like or agree with a recommendation while not so explicit about positive feedback [27, 39]. Seventy-three percent of the developers who received Nudge notifications explicitly resolving the notifications positively indicates a clear positive sentiment that the developers exhibit toward the Nudge service.

8.4.2 Nudge-LT User Feedback. We tried to understand how helpful our suggestions are and whether they are yielding intended benefits, i.e., driving pull requests toward a terminal state that is completion or abandonment. We received positive feedback (comments from developers) and observed that intended actions are taking place on the pull requests. To provide a glimpse, we list some of the quotes that we received from the developers that are appropriate to discuss in the context of this article. On one of the pull requests, a developer said,

> *I agree. Making a few more changes and pushing this pull request through! Thanks for the notification!*

We then saw this developer acting on this pull request by pinging the reviewers and driving this pull request toward the completion within 8 minutes.

In another pull request, the developer first replied to the Nudge notification saying,

> *The pipeline is failing and blocking this check-in. Followed up with an ICM incident and completed the pull request!*

Then, within a day, the pull request was abandoned. Thus, Nudge is not just about merging approved pull requests quicker but also about pushing pull requests to a terminal state, including abandonment, and in this way maintaining repository hygiene.

For Nudge-LT, we also received feedback that says the notification is not useful, because it is blocked by a reviewer. For example,

> *The comment does not add any value to me personally because I already know that the pull request I've authored has been open for a long time. It is not me who is blocking this but the reviewer.*

Similarly, For Nudge-LT we see comments about why notification is considered not useful in cases where the author interacted with the pull request recently by resolving a comment or pushing a new commit, which we nevertheless ended up nudging, because the lifetime of the pull request was long. One such comment comes from a developer who says,

> *I just resolved the comments on this pull request yesterday. I know about this one being pending for a while. This is not helpful!*

Both cases were in fact addressed by the actor and activity detection mechanisms of Nudge-FULL.

8.4.3 Nudge-FULL user Feedback. Consistent with the many positive notification resolutions (Section 8.4.1), many users were positive about the actor identification and activity detection enhancements. While there were some differences on how long the service should hold itself back before sending a notification when an activity is seen (24 hours vs. 48 hours), we generally received agreement about the usefulness of these features. When asked about determining change blockers and "@ mentioning" them in the notification thus eliminating an extra hop, users stated,

> *Yes it'll be nice for the tool to ping the reviewers instead of having the person do it.*

> *Yes I think that's handy to notify specific people. I often see someone "waiting" on a PR for changes, but then forget to revisit and follow up after changes have been pushed.*

Another user indicated that the algorithm was very accurate in determining the change blocker for a pull request that he was working on,

> *Change blocker was perfectly identified and notified for pull request 731796. You did my job!*

While the deployment of the activity detection and actor identification modules reduced the negative feedback significantly, there remain cases where the developers expressed their dislike toward the Nudge notifications. For example,

> *This pull request is awaiting on another pull request due to a module-level dependency. Thanks for the reminder though!*
>
> *I know what I am doing. This is not helpful.*
>
> *I went on a vacation. I would have liked it if you knew that and did not nudge me.*

Suggestions on how to address this feedback are discussed in Section 9.

8.5 RQ4: Nudge at Scale

To assess the impact of scaling up to thousands or repositories, we report Microsoft's experiences with deploying Nudge in production. The initial deployment of the Nudge service on 147 source code repositories and the observed efficiency gains and positive user feedback convinced Microsoft management to deploy Nudge beyond the original repositories. Thus, we trained and deployed the "Nudge-FULL" configuration for 8,000 repositories. From January 2021 to December 2021, the Nudge service sent notifications on 210,000 pull requests authored by 40,000 unique developers. This deployment corresponded to an increase by a factor of 50 of in the number of repositories compared to the initial experiment. This increase was easily handled by Nudge, thanks to the fact that scalability was a design consideration right from the start.

We could not perform A/B testing as on the deployment on 147 repositories of the Nudge service due to administrative and logistical reasons. However, we were able to collect two important metrics from the large-scale deployment: (1) the positive resolution percentage and (2) the distribution of pull request that are completed within a working day, 3 days, and a week.

We found that 71.5% of the 210,000 Nudge notifications were resolved positively. This is close to the 73% positive resolution percentage from the Nudge service deployment on 147 repositories. Similarly to the small-scale deployment, 16.35% of the pull requests took more than a week to close (formerly 18.47%), and 83.65% of the pull requests were closed within a week (formerly 81.53%). These numbers indicate that the findings from RQ1–RQ3 continue to hold true when deployed at the scale of thousands or repositories.

9 DISCUSSION

In this article, we presented Nudge, a service for improving software development velocity by accelerating pull request completion. Nudge leverages machine learning-based effort estimation, activity detection, and actor identification to provide precise notifications for overdue pull requests. Our experiments on 8,500 pull requests in 147 repositories over a span of 18 months demonstrate a reduction in completion time by over 60% (from 197 hours on average to 77 hours) and 73% of the developers reacted positively to being *nudged*—numbers that continued to be valid when we scaled up Nudge to thousands of repositories. In this section, we reflect on these contributions, assess their limitations, consider design alternatives, and explore future implications of our findings.

9.1 Explicit Completion Times

In our current implementation, pull request completion time is an attribute internal to Nudge, that is not shared with the pull request authors. An alternative design would be to let the author use the predictor to get an estimate of how long it would take to close this pull request, which they then can use to set a deadline for the pull request completion. We did not pursue this route, because doing this might adversely impact the pull requests: The prediction might become a self-fulfilling prophecy causing unnecessary delay [17, 33]. Also, the pull request process will become unnecessarily complicated, since the author and reviewers might engage in a back-and-forth discussion to decide the deadline.

9.2 Interruptions

Nudge uses the existing functionality in Azure DevOps to remind the actors by adding comments to the pull request. These comments would result in email notifications that can be addressed asynchronously. This lightweight workflow is no different from other notifications that are sent when a reviewer is added to the pull request or they add a comment to the PR. Therefore, given the asynchronous nature and also based on the survey results, we do not believe that Nudge causes significant interruption for the reviewers. Also, recall that Nudge sends only one Nudge notification per pull request to minimize repeated interruptions.

Nudge does not reduce the total effort needed to complete a pull request. Instead, it warns developers that others are waiting for them, suggesting them to prioritize the work on a given pull request. The cost of this for the nudged developer is that some other work (ongoing coding activities, opening a new pull request, responding to another pull request) is delayed, while the nudged pull request is moved forward. With Nudge, developers can take an informed decision whether to work on the pull request in question sooner rather than later. In this way, they not just optimize their own queue of tasks locally, but can take a bigger picture into account, reducing the number of developers who are waiting for them to take action.

9.3 Code Review Quality

In our work on Nudge, we have focused on the calendar time duration of code reviews, since it is deterministic and observable. Furthermore, in an industrial context, such speed of code reviews is important because of time-bound product release lifecycles. In case of bugs and incidents, faster code reviews can help with faster resolution of bugs and quicker service restoration.

In this article we have assumed that the total amount of effort in a pull request is not affected by Nudge: Tasks are moved earlier in time, but the nature of these tasks remains the same. In line with that, we argue that the *quality* of the reviews and code changes in nudged pull requests is not affected by Nudge. Nevertheless, it could be the case that developers feel pressure based on nudges received, and hence rush their work, and deliver lower quality. However, it could also be that developers are able to deliver *better* work, since handling of the pull request takes place in a more confined time span, requiring fewer context switches, or context switches that are closer in time together. We leave a rigorous investigation of the effect of nudging reviewers and developers on pull request quality as future work.

9.4 Simplifying Lifetime Prediction

An alternative to our learned lifetime prediction model is to work with a simple *constant* model. We explored this, as stated in Section 8.2, by taking the mean of the pull request lifetime as the estimated lifetime for all pull requests in the population. While simpler, such a constant approach suffers from the following problems:

(1) Nudge has been designed to be operationalized on tens of thousands of repositories, with different characteristics, processes/practices, and ever-changing dynamics. Thus, even a "constant" model is likely to require different settings across repositories and periodic re-calibration.
(2) The constant model will underestimate complex pull requests yet overestimate simple ones. This may undermine the confidence in Nudge's notifications
(3) We conducted informal, small-scale user studies by showing the users the notifications and simulating the timing of the notifications of constant and actual Nudge-LT models.

> Developers are inclined toward a model that adapts to changing workloads (dynamic), customized by user profiles or history, and that considers the size or complexity of the pull requests.

9.5 Addressing Nudge Limitations

Twenty percent of the Nudge notifications (882/4081) received an explicit *won't fix* mark from the developers. We recognize the following reasons, together with a potential way to address them.

First, a pull request may be blocked by the progress on another pull request. Presently, we do not take such inter-pull-request dependencies into account. A possible next step is to scan pull requests for other pull requests mentioned in their discussions and to consider such dependencies when nudging, putting, e.g., more emphasis on blocking pull requests, and postponing nudging blocked pull requests until they are unblocked.

Second, while we have some level of detection to understand if a user is away, it is limited to detecting weekends and popular public holidays only, at this point. Future work includes incorporating an algorithm that looks at other data sources to detect and predict when a user will be away and account for that in the Nudge notifications.

Last, the Nudge system, at this point, does not "learn" based on user feedback. If a user passes negative feedback, then Nudge does not use that information to pass that back to the model and adjust the parameters. Accounting for the user feedback, structuring it so that Nudge could leverage it and determining the opportune moment to send the Nudge notifications are possible ways to further enrich Nudge.

9.6 Threats to Validity

9.6.1 Internal Validity. Our qualitative analysis was conducted by reaching out to the developers via Microsoft Teams. None of the interviewers knew the people that were reached out or worked with them before. We purposefully avoided deploying Nudge on repositories that are under the same organization as any of the researchers involved in this work. As Microsoft is a large company and most of the users of the Nudge service are organizationally distant from the people involved in building Nudge, the risk of response bias is minimal. However, there remains a chance that respondents may be positive about the system, because they want to make the developers of Nudge, who are from the same company happy. Last, for the error estimation of the machine learning models, we have used a single run of the 10-fold cross-validation. Using repeated cross-validation can result in a more accurate estimation of the performance of machine learning models.

9.6.2 External Validity. Depending on data availability and API usage policies, the Nudge model can be operationalized on other popular git-based source control systems like GitHub, GitLab, BitBucket, and so on. However, the coefficients or the factors that impact the completion time of the pull requests, change blockers, and so on, may vary in those systems. Careful analysis of large samples of open source data has to be performed before the Nudge model is deployed on systems like GitHub. Some of the implementation details such as the heuristics used for identifying non-human actors will need to be adapted depending on the context. Similarly, in the current implementation, we remove the 48 hours period corresponding to the weekend while computing pull request completion time, yet this may not be applicable to open source projects.

The empirical analysis, design and deployment, evaluation, and feedback collection have been conducted specifically in the context of Microsoft. Given that Microsoft is one of the world's largest concentration of developers and developers at Microsoft use a very diverse set of tools, frameworks, and programming languages, our research, and the Nudge system will have broader applicability. However, at this point, the results are not verified in the context of other organizations or the open source community.

10 CONCLUSION

Pull request is a key part of the collaborative software development process. In this article, we presented Nudge, a service for improving software development velocity by accelerating pull request completion. Nudge leverages machine learning-based effort estimation, activity detection, and actor identification to provide precise notifications for overdue pull requests. To make the notifications actionable, Nudge infers the actor, the pull request author or its reviewer(s), who is delaying the pull request completion.

We have conducted a large-scale deployment of Nudge at Microsoft where it has been used to *nudge* over 8,500 pull requests, over a span of 18 months, in 147 repositories. We have also conducted a qualitative and quantitative user study to assess the efficacy of the Nudge algorithm. Our findings include that 73% of the notifications by Nudge have been positively acknowledged by the users. Further, we have observed a significant reduction in completion time, by over 60% on average, for pull requests that were *nudged*.

We further scaled out Nudge to 8,500 repositories at Microsoft and presented results from the large-scale deployment. We observe that Nudge service was able to retain a good positive resolution percentage (71.5%) similar to the deployment on 147 repositories (73%). We also observe that 83.65% of the nudged pull requests were completed within a week similar to the deployment on 147 repositories (81.53%).

At the time of writing, the results reported in this article have been the reason for Microsoft to explore adopting Nudge to a wider set of repositories. Though culturally very different from Microsoft systems, we also believe Nudge-like functionality could be beneficial to repositories of many open source systems. From a research perspective, we see future research in the areas of measuring the impact of shorter or longer reviewing cycles on reviewing quality, refining the pull request lifetime prediction models, taking inter-repository dependencies into account when nudging, and estimating reviewer availability to make nudges as meaningful as possible.

ACKNOWLEDGMENTS

We thank Rahul Kumar, Tom Zimmermann, B. Ashok, Suhas Shanbhogue, and Mei Nagappan for all their help with this work, and the anonymous reviewers for their valuable feedback.

REFERENCES

[1] Azure DevOps REST API. Retrieved 2020 from https://docs.microsoft.com/en-us/rest/api/azure/devops/?view=azure-devops-rest-5.0.

[2] GitHub. Retrieved 2020 from https://flow.microsoft.com/en-us/.

[3] Accessed 2020. GitHub. Retrieved 2020 from https://flow.microsoft.com/en-us/blog/sending-pull-request-review-reminders-using-ms-flows/.

[4] GitHub. Retrieved 2020 from https://www.openml.org/a/estimation-procedures/9.

[5] GitHub Marketplace. Retrieved 2020 from https://github.com/marketplace.

[6] Long-Running Branches Considered Harmful. Retrieved from https://blog.newrelic.com/culture/long-running-branches-considered-harmful/.

[7] 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 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 937–945.

[8] Iman Attarzadeh, Amin Mehranzadeh, and Ali Barati. 2012. Proposing an enhanced artificial neural network prediction model to improve the accuracy in software effort estimation. In *Proceedings of the 4th International Conference on Computational Intelligence, Communication Systems and Networks*. IEEE, 167–172.

[9] Chetan Bansal, Sundararajan Renganathan, Ashima Asudani, Olivier Midy, and Mathru Janakiraman. 2020. DeCaf: diagnosing and triaging performance issues in large-scale cloud services. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice*. ACM, 201–210.

[10] Olga Baysal, Oleksii Kononenko, Reid Holmes, and Michael W. Godfrey. 2013. The influence of non-technical factors on code review. In *Proceedings of the 20th working conference on reverse engineering (WCRE)*. IEEE, 122–131.

[11] Nicolas Bettenburg, Meiyappan Nagappan, and Ahmed E. Hassan. 2015. Towards improving statistical modeling of software engineering data: Think locally, act globally! *Emp. Softw. Eng.* 20, 2 (April 2015), 294–335. https://doi.org/10.1007/s10664-013-9292-6

[12] Ranjita Bhagwan, Rahul Kumar, Chandra Sekhar Maddila, and Adithya Abraham Philip. 2018. Orca: Differential bug localization in large-scale services. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*. USENIX, 493–509.

[13] Barry Boehm, Brad Clark, Ellis Horowitz, J. Westland, Raymond Madachy, and Richard Selby. 1995. Cost models for future software life cycle processes: COCOMO 2.0. *Ann. Softw. Eng.* 1 (12 1995), 57–94. https://doi.org/10.1007/BF02249046

[14] Barry W. Boehm. 1984. Software engineering economics. *IEEE Trans. Softw. Eng.* 10, 1 (January 1984), 4–21. https://doi.org/10.1109/TSE.1984.5010193

[15] Lionel C. Briand, Khaled El Emam, Dagmar Surmann, Isabella Wieczorek, and Katrina D. Maxwell. 1999. An assessment and comparison of common software cost estimation modeling techniques. In *Proceedings of the 21st International Conference on Software Engineering (ICSE'99)*. ACM, New York, NY, 313–322. https://doi.org/10.1145/302405.302647

[16] Lionel C. Briand, Jürgen Wüst, John W. Daly, and D. Victor Porter. 2000. Exploring the relationship between design measures and software quality in object-oriented systems. *J. Syst. Softw.* 51, 3 (May 2000), 245–273. https://doi.org/10.1016/S0164-1212(99)00102-8

[17] Gul Calikli, Berna A. Uzundag, and Ayse Bener. 2010. Confirmation bias in software development and testing: An analysis of the effects of company size, experience and reasoning skills. In *Proceedings Workshop on Psychology of Programming Interest Group (PPIG'10)*, Rebecca Yates and Fabian Fagerholm (Eds.).

[18] Sunita Chulani, Barry Boehm, and Bert Steece. 1999. Bayesian analysis of empirical software engineering cost models. *IEEE Trans. Softw. Eng.* 25, 4 (July 1999), 573–583. https://doi.org/10.1109/32.799958

[19] Nicola Dell, Vidya Vaidyanathan, Indrani Medhi, Edward Cutrell, and William Thies. 2012. "Yours is better!": Participant response bias in HCI. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI'12)*. Association for Computing Machinery, New York, NY, 1321–1330. https://doi.org/10.1145/2207676.2208589

[20] Tapajit Dey, Sara Mousavi, Eduardo Ponce, Tanner Fry, Bogdan Vasilescu, Anna Filippova, and Audris Mockus. 2020. Detecting and characterizing bots that commit code. In *Proceedings of the 17th International Conference on Mining Software Repositories (MSR'20)*. ACM, 209–219.

[21] Klissiomara Dias, Paulo Borba, and Marcos Barreto. 2020. Understanding predictive factors for merge conflicts. *Inf. Softw. Technol.* 121 (2020), 106256.

[22] Alberto Faro, Daniela Giordano, and Mario Venticinque. 2021. Internetworked wrist sensing devices for Pervasive and M-Connected Eldercare. In *Proceedings of the IEEE 3rd Global Conference on Life Sciences and Technologies (LifeTech'21)*. 454–456. https://doi.org/10.1109/LifeTech52111.2021.9391828

[23] Rahul Kumar, Chetan Bansal, Chandra Maddila, Nitin Sharma, Shawn Martelock, and Ravi Bhargava. 2019. Building sankie: An AI platform for devops. In *Proceedings of the IEEE/ACM 1st International Workshop on Bots in Software Engineering (BotSE'19)*. IEEE, 48–53.

[24] S. V. Aswin Kumer, P. Kanakaraja, A. Punya Teja, T. Harini Sree, and T. Tejaswni. 2021. Smart home automation using IFTTT and google assistant. *Mater. Today: Proc.* 46 (2021), 4070–4076. https://doi.org/10.1016/j.matpr.2021.02.610

[25] Lucas Layman, Nachiappan Nagappan, Sam Guckenheimer, Jeff Beehler, and Andrew Begel. 2008. Mining Software Effort Data: Preliminary Analysis of Visual Studio Team System Data. In *Proceedings of the 2008 International Working Conference on Mining Software Repositories (Leipzig, Germany) (MSR'08)*. Association for Computing Machinery, New York, NY, USA, 43–46. https://doi.org/10.1145/1370750.1370762

[26] Carlene Lebeuf, Alexey Zagalsky, Matthieu Foucault, and Margaret-Anne Storey. 2019. Defining and classifying software bots: a faceted taxonomy. In *Proceedings of the IEEE/ACM 1st International Workshop on Bots in Software Engineering (BotSE'19)*. IEEE, 1–6.

[27] Dugang Liu, Chen Lin, Zhilin Zhang, Yanghua Xiao, and Hanghang Tong. 2019. Spiral of silence in recommender systems. In *Proceedings of the 12th ACM International Conference on Web Search and Data Mining (WSDM'19)*. Association for Computing Machinery, New York, NY, 222–230. https://doi.org/10.1145/3289600.3291003

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

[29] Chandra Maddila, Chetan Bansal, and Nachiappan Nagappan. 2019. Predicting pull request completion time: A case study on large scale cloud services. In *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'19)*. Association for Computing Machinery, New York, NY, 874–882. https://doi.org/10.1145/3338906.3340457

[30] Sonu Mehta, Ranjita Bhagwan, Rahul Kumar, Chetan Bansal, Chandra Maddila, B. Ashok, Sumit Asthana, Christian Bird, and Aditya Kumar. 2020. Rex: Preventing bugs and misconfiguration in large services using correlated change

analysis. In *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI'20).* 435–448.

[31] Varun G. Menon, Sunil Jacob, Saira Joseph, Paramjit Sehdev, Mohammad R. Khosravi, and Fadi Al-Turjman. 2020. An IoT-enabled intelligent automobile system for smart cities. *IEEE IoT J.* (2020), 100213. https://doi.org/10.1016/j.iot.2020.100213

[32] T. Menzies, A. Butcher, D. Cok, A. Marcus, L. Layman, F. Shull, B. Turhan, and T. Zimmermann. 2013. Local versus global lessons for defect prediction and effort estimation. *IEEE Trans. Softw. Eng.* 39, 6 (June 2013), 822–834. https://doi.org/10.1109/TSE.2012.83

[33] Raymond Nickerson. 1998. Confirmation bias: A ubiquitous phenomenon in many guises. *Rev. Gen. Psychol.* 2 (6 1998), 175–220. https://doi.org/10.1037/1089-2680.2.2.175

[34] Thomas J. Ostrand, Elaine J. Weyuker, and Robert M. Bell. 2004. Where the Bugs Are. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'04).* ACM, New York, NY, 86–96. https://doi.org/10.1145/1007512.1007524

[35] Steven Ovadia. 2014. Automate the internet with "if this then that" (IFTTT). *Behav. Soc. Sci. Libr.* 33, 4 (2014), 208–211. https://doi.org/10.1080/01639269.2014.964593

[36] Ayushi Rastogi, Nachiappan Nagappan, Georgios Gousios, and André van der Hoek. 2018. Relationship between geographical location and evaluation of developer contributions in Github. In *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM'18).* ACM, New York, NY, Article 22, 8 pages. https://doi.org/10.1145/3239235.3240504

[37] Luyao Ren, Shurui Zhou, Christian Kästner, and Andrzej Wąsowski. 2019. Identifying redundancies in fork-based development. In *Proceedings of the IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER'19).* IEEE, 230–241.

[38] Daricélio Moreira Soares, Manoel Limeira de Lima Júnior, Leonardo Murta, and Alexandre Plastino. 2015. Acceptance factors of pull requests in open-source projects. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing (SAC'15).* ACM, New York, NY, 1541–1546. https://doi.org/10.1145/2695664.2695856

[39] Harald Steck. 2011. Item Popularity and Recommendation Accuracy. In *Proceedings of the 5th ACM Conference on Recommender Systems (RecSys'11).* Association for Computing Machinery, New York, NY, 125–132. https://doi.org/10.1145/2043932.2043957

[40] Margaret-Anne Storey, Alexander Serebrenik, Carolyn Penstein Rosé, Thomas Zimmermann, and James D. Herbsleb. 2020. BOTse: Bots in software engineering (Dagstuhl Seminar 19471). In *Dagstuhl Reports*, Vol. 9. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

[41] Josh Terrell, Andrew Kofink, Justin Middleton, Clarissa Rainear, Emerson Murphy-Hill, Chris Parnin, and Jon Stallings. 2017. Gender differences and bias in open source: Pull request acceptance of women versus men. *PeerJ. Comput. Sci.* 3 (May 2017), e111. https://doi.org/10.7717/peerj-cs.111

[42] Jason Tsay, Laura Dabbish, and James Herbsleb. 2014. Influence of Social and Technical Factors for Evaluating Contribution in GitHub. In *Proceedings of the 36th International Conference on Software Engineering (ICSE'14).* ACM, New York, NY, 356–366. https://doi.org/10.1145/2568225.2568315

[43] Erik Van Der Veen, Georgios Gousios, and Andy Zaidman. 2015. Automatically prioritizing pull requests. In *Proceedings of the IEEE/ACM 12th Working Conference on Mining Software Repositories.* IEEE, 357–361.

[44] Qingye Wang, Bowen Xu, Xin Xia, Ting Wang, and Shanping Li. 2019. Duplicate pull request detection: When time matters. In *Proceedings of the 11th Asia-Pacific Symposium on Internetware.* 1–10.

[45] Song Wang, Chetan Bansal, and Nachiappan Nagappan. 2020. Large-scale intent analysis for identifying large-review-effort code changes. *Inf. Softw. Technol.* (2020), 106408.

[46] Song Wang, Chetan Bansal, Nachiappan Nagappan, and Adithya Abraham Philip. 2019. Leveraging change intents for characterizing and identifying large-review-effort changes. In *Proceedings of the 15th International Conference on Predictive Models and Data Analytics in Software Engineering.* 46–55.

[47] Marvin Wyrich and Justus Bogner. 2019. Towards an autonomous bot for automatic source code refactoring. In *Proceedings of the IEEE/ACM 1st International Workshop on Bots in Software Engineering (BotSE'19).* IEEE, 24–28.

[48] Yue Yu, Huaimin Wang, Vladimir Filkov, Premkumar Devanbu, and Bogdan Vasilescu. 2015. Wait for It: Determinants of Pull Request Evaluation Latency on GitHub. In *Proceedings of the 12th Working Conference on Mining Software Repositories (MSR'15).* IEEE Press, Piscataway, NJ, 367–371. http://dl.acm.org/citation.cfm?id=2820518.2820564.

[49] 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? *Inf. Softw. Technol.* 74 (2016), 204–218.

Received 16 March 2021; revised 20 February 2022; accepted 20 May 2022