Predicting Pull Request Completion Time: A Case Study on Large Scale Cloud Services

Chandra Maddila
Microsoft Research
Redmond, WA, USA
chmaddil@microsoft.com

Chetan Bansal
Microsoft Research
Redmond, WA, USA
chetanb@microsoft.com

Nachiappan Nagappan
Microsoft Research
Redmond, WA, USA
nachin@microsoft.com

ABSTRACT

Effort estimation models have been long studied in software engineering research. Effort estimation models help organizations and individuals plan and track progress of their software projects and individual tasks to help plan delivery milestones better. Towards this end, there is a large body of work that has been done on effort estimation for projects but little work on an individual check-in (Pull Request) level. In this paper we present a methodology that provides effort estimates on individual developer check-ins which is displayed to developers to help them track their work items. Given the cloud development infrastructure pervasive in companies, it has enabled us to deploy our Pull Request Lifetime prediction system to several thousand developers across multiple software families. We observe from our deployment that the pull request lifetime prediction system conservatively helps save 42.94% of the developer time by accelerating Pull Requests to completion.

CCS CONCEPTS

• **Software and its engineering → Software development methods**.

KEYWORDS

effort estimation, empirical studies, case studies, software metrics

ACM Reference Format:
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 European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '19), August 26–30, 2019, Tallinn, Estonia.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3338906.3340457

1 INTRODUCTION

Effort estimation models have been long studied in software engineering research. Effort estimation models help organizations and individuals plan and track progress of their software projects and individual tasks to help plan delivery milestones better. With the advent of development environments like GitHub [3], Pull Requests (PR) are becoming the norm for development. PRs are the equivalent of check-ins but involve the complete ownership from an individual or individual(s) including all testing and code review activities to be completed prior to their integration into the code branch. Bug fixes, new features, security fixes, content changes can all be part of a PR. In this paper we leverage the rich body of work in the effort estimation literature, defect prediction literature to build a prediction model to predict the completion time of a PR. We build a prediction model to estimate total effort required for completing a PR and integrate this estimate into the developer workflow to help with the code review process and improve developer productivity. This can be viewed as effort estimation in a local environment on a per PR basis. This potentially helps each individual developer plan their work better and identify and avoid delays which would eventually affect the entire project. Prior work at Microsoft [15] has also focused on effort estimation at the feature and project level, but not at the level of individual changes. We use several metrics from the defect prediction literature like code churn [19], reviewer information [16], ownership information [15] to build our PR lifetime prediction model. These predictions are first validated historically on systems with several thousands of developers across multiple product/service families at Microsoft. After validation, these predictions are pushed to the Cloud development environments where they automatically add a comment, should a PR be open beyond the predicted time. We have observed overwhelmingly positive response to this system with a 73% comment resolution rate and a 42.94% timing savings. We are in the process of turning on this system for several other product groups at Microsoft. In this paper, we make the following three contributions:

(1) We build and evaluate a prediction model for PR completion time using various in-product and process metrics.
(2) We deploy the ML model to actual developers to observe their behavior and do a quantitative and qualitative evaluation.
(3) We measure the efficacy of displaying the predictions to actual developers and observe a 42.94% improvement in Pull Request completion time.

The rest of the paper is organized as follows. In section 2 we discuss the implementation details of the data collection. Section 3 presents the empirical study with the data description, a correlation analysis, and the prediction model. Section 4 discusses the results of the deployment with qualitative and quantitative results. Section 5 presents the related work and we conclude with Section 6.

2 IMPLEMENTATION

In this section we discuss the implementation details for the PRLifetime Service. We start by discussing Azure DevOps on which it is deployed as an extension. Further, we describe the various components of the PRLifetime Service: 1) Data pipeline 2) Model Training

ESEC/FSE '19, August 26–30, 2019, Tallinn, Estonia
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-5572-8/19/08…$15.00
https://doi.org/10.1145/3338906.3340457

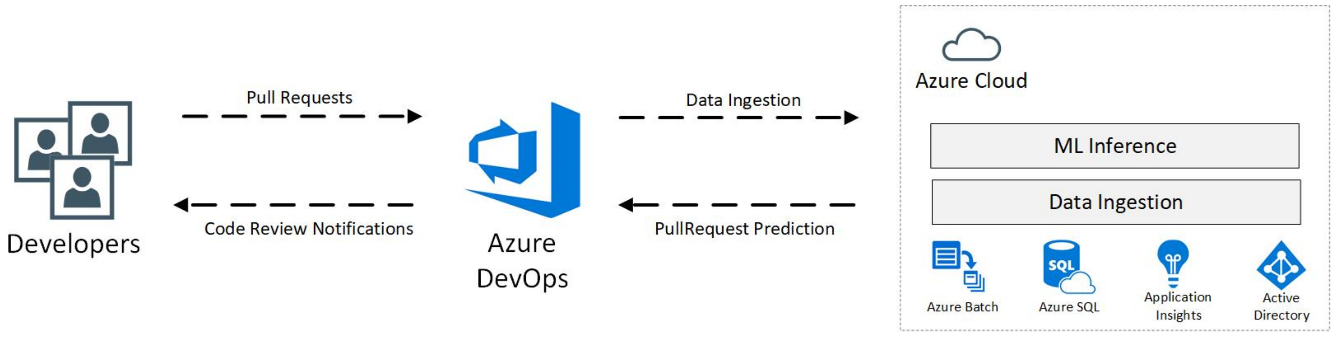


Figure 1: Architecture of the PRLifetime Service

3) Inference 4) Notification. Figure 1 shows a high level overview of the service architecture.

2.1 Azure DevOps

Azure DevOps (ADO) is a DevOps platform providing GIT based VCS (version control system) along with collaboration tools for developers such as Pull Request based code reviews, work item tracking and, also, planning tools for sprints / scrums. Lastly, it also enables CI / CD of code through CD pipelines. ADO is heavily used across Microsoft internally for product development. We have built and deployed PRLifetime Service as an extension of ADO because of the rich collaboration features. We also used ADO repository for developing the PRLifetime Service, allowing us to dogfood the feature within our own team as well. Below we talk about two main aspects of ADO which were crucial for the PRLifetime service:

(1) **Code Reviews** - ADO provides a rich set of features for doing code reviews for the code in the ADO repositories. To push any changes to the master branch of a repository, developers create Pull Requests (PR) and request their team members to review and approve the PRs. The reviewer can then either a) approve the PR or b) add inline comments on the PR and ask the PR author to revise the PR. The comments posted in the PR are visible to all the participants of the PR i.e. author and reviewers. As shown later in Figure 5, we leverage this comment functionality and add a comment whenever a PR has been pending for a significant time. We further discuss this in detail in Section 2.2.
(2) **APIs** - ADO exposes a rich set of REST APIs [2] to enable extensibility. These APIs not only allow us to ingest data from ADO for various entities such as Commits, Work Items, Pull Requests but also allows us to interact with the PR by adding comments, reviewers etc.

2.2 PRLifetime Service Implementation

The PRLifetime Service is implemented using C# and .NET 4.6. It's built on top of Microsoft Azure Cloud Services using Azure Batch [1] for compute, Azure SQL for data storage, Azure Active Directory for authentication and Application Insights for telemetry and alerting. We used ADO for collaborative development of the services and, also CI / CD for the codebase. To enable the service on an ADO repository, the ADO project admin grants access to the repository to a system account. Subsequently, the data pipelines in the PrLifetime Service start ingesting the data for training models and starts interacting with the PRs. The major components of the services are:

Backend Pipeline - We leverage Azure Batch for scaling the PRLifetime Service data pipelines which are responsible for data ingestion and, also decorating the PRs with comments. Azure Batch is a PaaS service for building high performance and parallel applications and abstracts out critical components such as the job scheduler, the compute resource provider, etc. Azure Batch allows us to scale horizontally as more and more repositories are onboarded. We create a job schedule for each of the onboarded repository and a job is triggered every 6 hours. As we onboard more repositories, our deployment auto scale the compute resources. A Batch job basically ingests/updates any new entities (PRs, Commits, etc.) using the ADO APIs that have been created or updated since the last run. Further, it also does inference and decoration of PRs.

Data Store - We use SQL Azure for storing the data and features used by the PRLifetime Service. We have a normalized schema for representing various entities like PRs, Commits, Users, etc. For each repository we onboard, we create a separate database. We use the Elastic Pool feature in SQL Azure to minimize the resource requirements. This allows us to be very efficient in terms of per repository cost.

Training data - We train our machine learning models using a subset of the PRs ingested. Based on our manual analysis, to reduce the noise, we exclude two sets of PRs:

(1) PRs with completion times less than 24 hours. We observed that if a PR is completed in few hours or within a day, most likely it is a change that needs to be rolled out immediately; like a security fix or some critical bug fix. In that case the PR is reviewed and merged on an accelerated schedule.
(2) PRs with completion time greater than 2 weeks. If a PR is sitting in the system for more than two weeks, as per our observation, they are mostly draft PRs or PRs that were abandoned due to a decision taken offline or PRs which are not that important but created and left around by the author before they head to a vacation or taking a time off.

PRs used for training = All the completed PRs - (PRs with less than 24 hours of lifetime + PRs with more than two weeks of lifetime)

Inference - For the ML predictions, we have trained and deployed the model in a Flask based python service in Azure. When the backend pipeline runs, it takes all the PRs that are active and have spent at least 24 hours in the system (this filter is due to the reasons mentioned in the model training section above). Then, we extract all the features in our feature space, for the current PR, and create a feature vector. Once the feature vectors are created for all the PRs for which we are planning to do inference, we make REST calls to the inference API we already deployed. Inference results for each PR (predicted Lifetime) is then saved in our cloud database.

Notification to the Developers During every run of our pipeline, we look at all the PRs that meet all of the following criteria:

(1) PRs that are active.
(2) PRs for which predicted lifetime is generated and saved in our database, during the previous run of the pipeline.
(3) PRs for which we have not already generated a notification earlier
(4) Age of the PR is at least (predicted lifetime + 24 hours).

We consider such PRs as a candidate for notification and a notification is sent to the PR author and reviewers of the PR to act on it (as shown in Figure 5). Time of notification (Tn) is defined as:

$$Tn = Tc + Tp + 24 \text{ hours}$$

where Tc is the PR creation date time and Tp is the predicted lifetime for a PR (in hours). After doing user studies and talking to various developers and managers, we decided to give a buffer time of 24 hours to account for variations in work schedules.

3 EMPIRICAL STUDY

We can broadly classify our empirical study into 5 main steps.

(1) Data Collection: Leveraging the rich history of prior work done in the Effort Estimation and Mining Software Repositories community regarding factors for Pull Request acceptance and defect prediction, we identify a set of metrics.
(2) Run the data collection algorithm on multiple repositories and collect the ground truth data.
(3) Use the data collected in Step 2 to build a Pull Request completion prediction model. Evaluate against training and test data to evaluate the efficacy of the model.
(4) Deploy the prediction model to the PRLifetime service and evaluate the user feedback on the decorations.
(5) Based on feedback from users, if needed, refine the prediction model.

3.1 Data Description

In this section we broadly describe the various metrics that are collected for building our Pull Request completion model. Some of the metrics are self-descriptive and for others we explain the metric.

3.2 Correlation Analysis

We performed a correlation analysis to understand what factors impact the completion time of PRs and what is the magnitude of impact. We collected 2,875 pull requests from 10 different repositories that are merged and completed. These repositories host the source code of various services ranging from a few hundred to a few thousand developers. The description of the measure and the various features used is presented in Table 1.

(1) **Outcome** The outcome measure is the PR completion time, i.e., the time interval between pull request creation and closing date, in hours. In case of re-opened pull requests, we only consider the date when they are first closed.
(2) **Features** We computed various PR related, author related, process related and churn related features whose correlation to PR lifetime are explained below in Table 2.

Here is the detailed description of some of the features:

Day of the week when the PR was created - Day of the week on which the PR is created. This helps us understand PRs created on which days are moving faster. We represent Sunday with 0 and Saturday with 6. A strong positive correlation with this metric indicates that PR created later in the week (like Friday or Saturday) are taking more time to complete.

Average age of PRs created by the PR author - This captures how quickly a specific Author's PRs were moving historically. The idea here is, if a developer is junior or new to a particular repository or project they tend to make more mistakes or they are subjected to more thorough reviews and testing which potentially delays their PRs. Over time they become better at completing their PRs and our model tries to capture that. Our correlation analysis shows a strong positive correlation that bolsters the argument that more time a developer's PRs took to complete historically, more time the current one takes.

Number of reviewers of the PR - Number of reviewers actively reviewing the PR. The intuition here is if more people are actively reviewing a PR and are engaged with it, more number of comments and questions are raised (probably more nitpicks too) and that could potentially delay the PR. A strong positive correlation here indicates more the number of reviewers more time it takes for it to be completed / merged.

Is a .csproj file being edited - If a .csproj file is being touched? We represent this with 1 if it is and 0 if it is not. A .csproj file in C# context is a project configuration file which tracks files in the current project/solution, external package dependencies and their versions, inter dependencies among different projects in a solution etc. Any modifications to these files indicate that major activity like adding or deleting files / modifying external dependencies or DLLs or modifying dependencies on other projects is happening. A very good positive correlation here indicates that any changes made to these files (represented by value 1) increases the PR's lifetime

Average age of PRs with similar paths changed - This indicates how PRs that touched same paths as the ones that are being modified in the current PR impacts PR's completion time. The intuition here is, there are always certain code paths which are known to be riskier and any changes to them usually takes very long time before they are merged. A positive correlation here indicates that more time it took for PRs, that are touching same paths/files as the current PR, in the past to complete, more time it takes to the current PR to be merged.

Is the PR a Bug fix? - This is 1 if the PR is doing a bug fix and 0 if it is not. The intuition here is, in large scale service and DevOps environments, bug fixes (especially critical bugs or showstoppers) will always be given higher priority and people just walk around and get all the needed sign-offs and try to push them as soon as possible. A weak negative correlation here indicates that possibly

Table 1: Feature description

Feature	Description	Type
Day of the week	The day of the week when the PR was created	Categorical
Avg age of PRs Of this developer	The average time for Pull Request completion by the developer who initiated this PR	Continuous
Number of reviewers	Total number of required reviewers on the current PR	Continuous
Is .csproj file being edited	Is .csproj file being modified?	Categorical (Binary)
Average age of PRs with similar paths changed	The average time for completion for the PRs which have the same project paths changed	Continuous
Number of distinct file types	Total number of distinct file types that are being modified	Continuous
PR description word count.	The word count of the textual description of the PR	Continuous
Setting or config. change	Is the PR modifying any config. files or settings	Categorical (Binary)
Number of active PRs at this time	To understand the overall load on the project with respect to open PRs	Continuous
Class churn	Churned LOC per class	Continuous
Method churn	Number of methods being churned	Continuous
New feature	Is this PR introducing a new feature?	Categorical (Binary)
LOC Changed	Number of lines changed	Continuous
Number of paths touched	Number of distinct paths that are being touched in the current change	Continuous
Conditional statements churn	Number of conditional statements being touched	Continuous
Loop churn	Number of loops being touched	Continuous
Class Member Churn	Number of classes being added/modified/deleted	Continuous
Is It Refactor	Is the PR doing any refactoring of existing code?	Categorical (Binary)
Reference churn	Number of references or dependencies (on other libraries / projects) being changed	Continuous
Number of files changed	Number of files that are being modified in PR	Continuous
Is it a merge change	Is the PR making any merge changes like forward or reverse integration (FIs / RIs)	Categorical (Binary)
Is it a deprecate change	Is the PR deprecating any old code?	Categorical (Binary)
PR Title Word Count	The word count of the textual title of the Pull Request.	Continuous
Is PR created during Business Hours?	Whether the PR is created during business hours or off hours?	Categorical (Binary)
Is it Bug Fix	Is the PR fixing bugs?	Categorical (Binary)
Age of the PR author in current team	Time spent by the developer in the current team.	Continuous
Age of the PR author in the repository	Time since the first activity in the repository by the PR author	Continuous
Age of the PR author in Microsoft	Time spent by the developer at Microsoft.	Continuous

PRs which are bug fixes (with value 1) takes lesser time to complete than the non bug fixes (with value 0).

Age of the PR 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. Intuitively, more time a developer spends in a team, less difficulty he/she will experience in pushing their change through. A negative correlation here indicates the possibility of the presence of such a relationship.

Age of the PR author in the repository - This helps capturing the familiarity of a developer with the repository in which he/she is making changes and the build, deployment processes of that repository. Though this sound similar to the "Age of the developer in current team", this varies a lot with the heterogeneous teams which work on multiple services (especially, micro services) and 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 more familiar a developer is with a specific repository, less time it takes for him/her to merge their changes made in that repository.

Age of the PR author in Microsoft - This helps capturing the seniority of a developer. Intuitively, senior people who has more experience tend to make less mistakes and doesn't experience push back on their changes. A negative correlation here is indicating that

Table 2: Correlation analysis

Feature	Correlation score
Day of the week when PR was created	0.163
Avg age of PRs by the PR author	0.159
Number of reviewers of the PR	0.131
Is csproj being edited	0.103
Average age of PRs With similar paths changed	0.089
Number of distinct file types	0.084
Pr description word count	0.072
Is settings or config changed in the PR	0.059
Number of active PRs at the time the PR was created	0.058
Class churn	0.055
Total churn	0.039
Method churn	0.037
Is the PR introducing a new feature	0.033
Loc changed	0.031
Number of paths touched	0.031
Conditional statements churn	0.029
Loop churn	0.028
Class member churn	0.021
Is the PR for refactoring	0.021
Reference churn	0.017
Number of files changed	0.016
Is it a merge change	0.008
Is it a deprecate change	-0.001
PR title word count	-0.001
Is the PR created during business hours	-0.019
Is it a Bug Fix change	-0.028
Age of the PR author in current team	-0.031
Age of the PR author in the repository	-0.046
Age of the PR author in Microsoft	-0.056

if someone has more experience, less time it takes them to merge their changes.

3.3 Prediction Model

We formulated the task of predicting the lifetime of a PR as a regression problem. 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 2 standard metrics: MAE (Mean Absolute Error), MRE (Mean Relative Error). These metrics are widely used for understanding the performance of regression tasks. We decided to go with gradient boosting algorithm as it has better accuracy with respect to both MAE and MRE. The comparative analysis of the three algorithms, evaluated against MAE and MRE is shown in Table 3.

Table 3: Comparison of different prediction models

Algorithm	MAE	MRE
Least squares	44.32	0.68
Bayesian Ridge	46.35	0.71
Gradient Boosting	32.59	0.58

For training and evaluation, we used Scikit-learn 0.20.0 package for Python 3.7.1. We used a standard evaluation technique called 10-fold cross-validation. The process of 10-fold cross-validation is explained below:

(1) Separating the data set into 10 partitions randomly
(2) Using one partition as the test data and the other nine partitions as the training data
(3) Repeating Step-2 with a different partition than the test data until all data have a prediction result
(4) Computing the evaluation results through comparison between the predicted values and the actual values of the data.

Finally, we decided to go with a gradient boosting model with our feature space containing 28 features as described in section 3.1. The details of the model are:

(1) Model time: Regression
(2) Algorithm used: Gradient boosting
(3) Toolkit: Python 3.7.1 - Scikit learn 0.20.0
(4) Feature space: 28 features
(5) Data points: 2,875 [PRs] from 10 repositories
(6) For the training set, Gradient Boosting MAE: 29.29, MRE: 0.52.

Our predictive model is deployed and runs as a Python flask cloud service and whenever a new PR is created and spent some time (24 hours), we generate a feature vector for that PR and infer its lifetime against the model that was pre-built. We discuss various evaluation metrics of our prediction model in the evaluation section.

4 RESULTS

We would like to answer two important questions here when we talk about results.

(1) How accurate is a prediction/estimate?
(2) Are we helpful in driving a PR towards a terminal state 'Completed' or 'Abandoned'? If so, by what margin are we expediting?

To answer question 1, we generated metrics and visualizations that explain how accurate our prediction model is (we will be taking about in section 4.2). And, to answer question 2, we generated metrics to capture quantitative feedback and interviewed the developers on whose PRs we made suggestions and gathered qualitative feedback. In the subsequent sections, we are going to discuss both our quantitative and qualitative analysis.

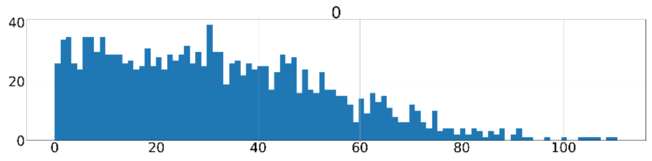


Figure 2: MAE distribution

4.1 Model Evaluation

We evaluated our prediction model against standard metrics: MAE and MRE. The Mean Absolute Error (MAE) is 32.60 hours and Mean Relative Error (MRE) is 0.58. To put these numbers in perspective, we have done an experiment by considering mean lifetime of our training data as the predicted lifetime of every PR in our testing data. Our random model's MAE is 36.43 hours and MRE is turned out to be 0.68. We are doing 11.8% better with MAE and 17.7% better with MRE compared to the random model. To understand if a MAE of 32.60 hours is reasonable, we should also understand few important metrics of the data on which we are training and doing inference against. The data set we are dealing with has a mean PR Lifetime of 107.63 hours and the earliest a PR is completed is in 24 hours (minimum) & a maximum lifetime of a PR is 336 hours. Basically, we have a prediction MAE of 32.60 for data points (PRs) ranging from 24 hours to 336 hours. Figure 2 shows the MAE distribution. A very interesting thing to note is, a significant portion of our MAE lies within the range of 50 hours and only a very little portion of our absolute error exceeds order of 107 hours (which is the mean of this distribution).

An important thing to note is, majority of our MRE (close to 90%) falls under a relative error of 1. Figure 3 shows the MRE distribution. As with any regression problems, one of the other important metrics to understand is tolerance. Based on the domain we are operating in and the users who are consuming our prediction results, tolerance levels may vary. For instance, in real estate or in finance, a tolerance of 50% is not acceptable. However, in effort estimation, going off by +/- 50% hours is reasonable since we are only alerting developers for acting on pending Pull Requests. In our Qualitative analysis section (4.3), we list anecdotes from users which supports this argument. The distribution of our prediction accuracy based on the tolerance level is listed in Table 4.

4.2 Quantitative Analysis

For every comment (as shown in Figure 3) we add in every PR, we calculate comment resolution percentage (CRP). CRP indicates how many comments made by us are honored and/or agreed on by the author of the PR.

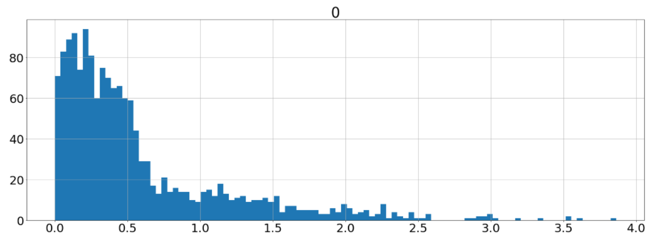


Figure 3: MRE distribution

Table 4: Tolerance level distribution

Tolerance level	% of predictions within tolerance level
+/- 0%	0.7
+/- 5%	7.17
+/- 10%	13.97
+/- 15%	21.33
+/- 20%	28.44
+/- 30%	41.06
+/- 50%	62.91
+/- 70%	74.44
+/- 90%	79.82
+/- 100%	81.29

CRP = No. of comments resolved / Total no. of comments made

We looked at the comments added by our system in all the 43 repositories in which it is enabled. We have made 1069 comments in last three months and our comment resolution percentage is 73.3. To descriptively put, 73 out of 100 comments made by us are resolved by the developers and indicate that developers agree with our recommendations. We realized that while CRP indicates a strong agreement by the author, we did another analysis (explained below) to understand if the PRs are getting completed faster because of the comments we are adding.

To assess the empirical efficacy of the comments, we compare the PR completion times for notified PRs and non-notified PRs (shown in Table 5). As described in Section 2.2, we notify PR authors for PRs which are 24 hours past the predicted PR completion time. In total, we notified the author of 913 PRs which are over 24 hours past the predicted PR time. We found that after decorating the PR completion time on average is 115.81 hours after the PR decoration. To compare against the scenario where we have no PR decoration, we look at 5769 older PRs. For these PRs we split the time as predicted+24 hours in one bucket and the time remaining after subtracting the predicted+24 hours. The Average time in this case is 165.54 hours. Comparing this with the 115.81, conservatively we see a quicker turnaround to resolving and closing PRs by 43%. Additionally, we observe that most PRs are completed within a week of decoration without comment. The distribution of the completion times of the PRs after the decoration is shown in Figure 4.

Table 5: Average PR completion time comparison

Category	Count	Time from decoration to completion (AVG)
Non-Notified PRs	1655	103.07
Notified PRs	1069	71.27

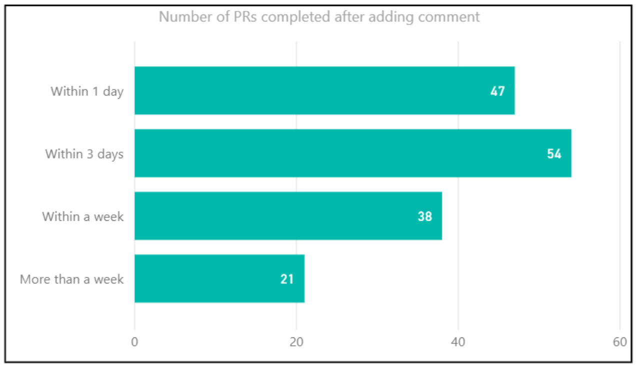


Figure 4: Distribution of PR completion times after decoration

4.3 Developer Feedback

In addition to quantitative analysis, we have reached out to developers from various product groups (17 people, including managers) on whose PRs we added a comment. We interviewed them and tried to understand what they think about our service. To paraphrase, our evaluation metrics are checking two things essentially: 1). If the comment we are adding in their PR is helping them drive the PR towards completion 2). If the number of hours we are predicting as the estimated lifetime of a PR are making sense. We decided not to make the interview anonymous and pinged people directly on Skype/Teams. We did not make an effort to make it anonymous because, the quality of responses is not going to be impacted as we are not asking any personal questions that make people frank and honest about their responses in an anonymous setting. In this interview, we asked two open-ended questions:

(1) If this comment has any value add? Do you think it helps you moving / acting on this PR towards completion or abandonment?
(2) Do you think the predicted PR completion time is reasonable?

4.3.1 Helpfulness in driving PRs towards completion / abandonment. We tried to understand how helpful our suggestions are and whether they are yielding intended benefits i.e., driving PRs towards a terminal state which is Completion or Abandonment. We received very positive feedback (comments/anecdotes from the developer) and observed that intended actions are happening on the PRs. On one of the PRs, a developer said *"I agree. Making few more changes and pushing this PR through! Thanks!"* Then we saw this developer acting on this PR by pinging the reviewers and driving this PR towards completion. And within 8 minutes this PR got completed. In another PR, the developer has acknowledged our nudge and completed it within a day. In another PR, the developer has done something very interesting. First, he replied to our comment saying *"The pipeline is failing and blocking this check in. Followed up with an ICM incident!"* then, soon enough (within a day) the PR was abandoned. Here, it is not just about completion, people are making progress on their PRs by pushing them towards a terminal state (completion/abandonment) and in turn maintaining the repository hygiene. So, we are clearly seeing that people are responding to the comments and most importantly, taking an action to push the PRs towards completion/abandonment thus expediting the deployment of changes and achieving repository hygiene. There are few situations where we received feedback that says the comment that we made and the attempt to capture developer's attention to drive their PRs towards completion is not very helpful. A comment from one of our interview respondents is *"The comment does not add any value to me personally because 1) I already know that the PR I've authored / am reviewing has been open for a long time, 2) I am currently working on doing better at looking at PRs earlier, so basically all the PRs I've reviewing have this, 3) I have the emails about PRs that come to my inbox highly filtered to cut down on noise, so I only see these comments when I'm on the PR already"* As the developer above has mentioned, sometimes a PR is set to wait due to few external factors like, dependency on other PRs or management decisions like feature planning and road-maps etc. In these cases, pinging the developer and re-iterating the fact that their PR is safe to complete (especially when they are staying on top of that PR) doesn't seems to be adding much value. As part of our future work, we are planning to come up with a way to systematically determine the external factors that are blocking a PR and nudge the right people but not the authors of the PR (especially if they do not have any action item pending on their side or can't do anything to move their PR forward).

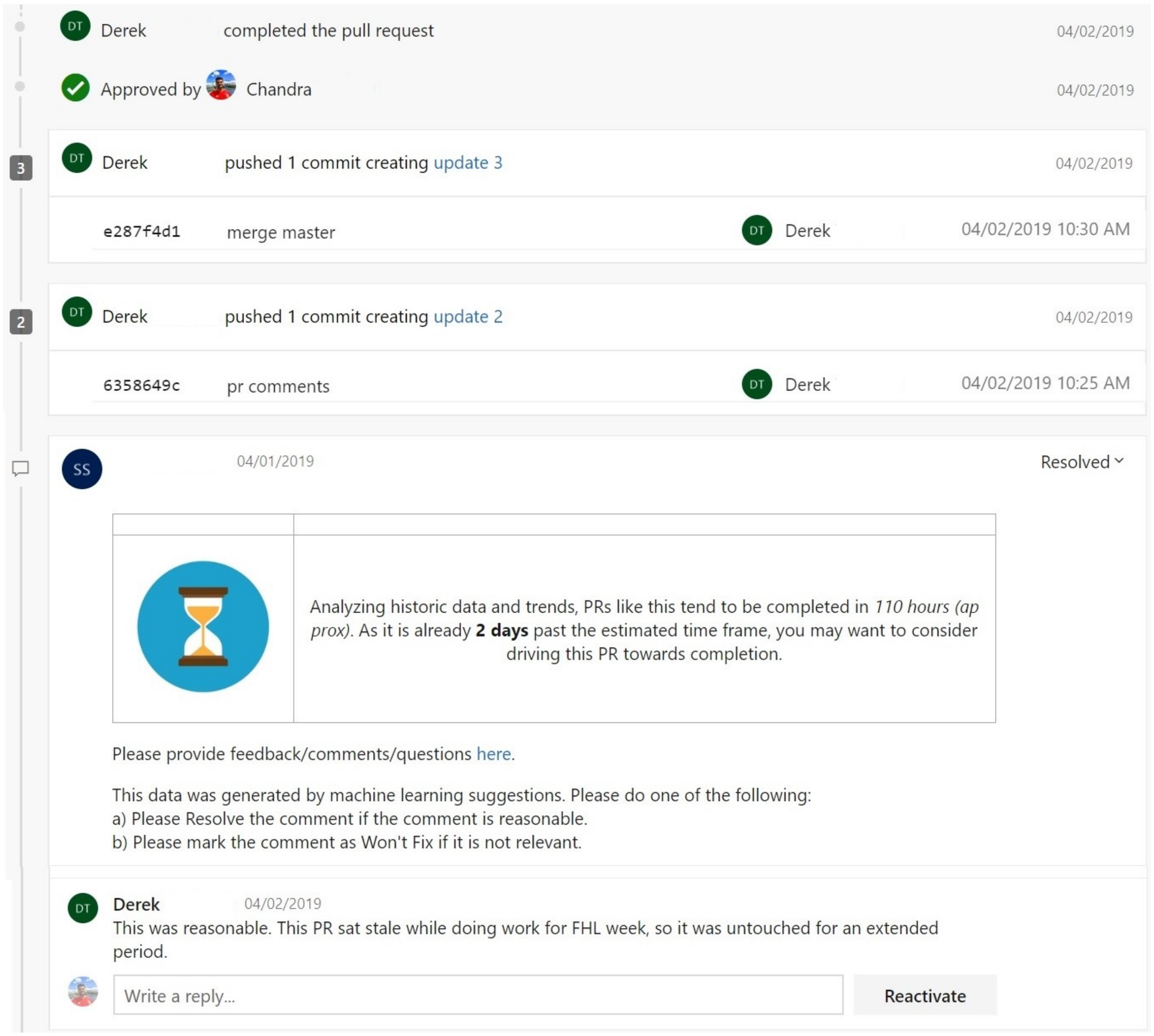
4.3.2 Accuracy of predictions for Pull Request completion time. We tried to understand how accurate our estimation is for PRs' completion time and we presented various metrics that bolster our accuracy claims in the quantitative analysis section. Here we are listing few anecdotes from the developer/author of the PR. *"Yes, I thought the approximation was pretty good"* Another developer, from a different PR and different repository says *"The approximation does sound about right. I went on-call which led to delay in check-in in this case. Normally, it would have been within about that range"*

5 RELATED WORK

Our related work is dived broadly into 3 sections. The first being on effort estimation, then on defect prediction and finally on PR acceptance

5.1 Effort Estimation

Software effort estimation is a field of software engineering research that has been studied a lot in the past four decades [5, 7, 9, 12, 17]. 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 [7], which he later updated to COCOMO 2.0 in 1995 [6]. This work was followed up by Briand et al. [8] which compared various effort estimation modelling techniques using the data set 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. [17] and Bettenburg et al. [5] looked at the variability present in the data and therefore built separate models for subsets of the data. Unlike, these past effort estimation studies, in our paper we


Derek completed the pull request 04/02/2019

Approved by Chandra 04/02/2019

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

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

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

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

SS 04/01/2019 Resolved

Analyzing historic data and trends, PRs like this tend to be completed in *110 hours (ap prox)*. 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.

Figure 5: A Pull Request with PRLifetime prediction in Azure DevOps

look into predicting how long it would take for a pull request to be accepted.

5.2 Predicting defects in software projects

In this line of study, researchers try to predict where software defects could be. Once that is known, the software developers could then focus on where their effort can be focused on. Structural object-oriented (OO) measurements, such as those in the CKOO metric suite [11], have been used to evaluate and predict fault-proneness [4, 8, 9]. These metrics can be a useful early internal indicator of externally visible product quality[4, 22, 23]. The CK metric suite consist of six metrics: weighted methods per class(WMC), coupling between objects (CBO), depth of inheritance (DIT),number of children (NOC), response for a class (RFC) and lack of cohesion among methods (LCOM). Vouk and Tai [26] showed that in-process metrics have strong correlation with field quality of industrial software products. They demonstrated the use of software metric estimators, such as the number of failures, failure intensity(indicated by failures per test case), and drivers such as change level, component usage, and effort in order to identify fault-prone and failure-prone components. Basili et al. [4] studied the fault-proneness in software programs using eight student projects. They observed that the WMC, CBO, DIT, NOC and RFC were correlated with defects while the LCOM was not correlated with defects. Tanget al. [23]

studied three real time systems for testing and maintenance defects. Higher WMC and RFC were found to be associated with fault-proneness. El Emam et al. [13] studied the effect of class size on fault-proneness by using a large telecommunications application. Class size was found to confound the effect of all the metrics on fault-proneness. Mockus et al. [18] predicted with significant accuracy the customer perceived quality using logistic regression for a commercial telecommunications system by utilizing external factors like hardware configurations, software platforms, amount of usage and deployment issues. Ostrand et al. [19] use information of file status such as new, changed, unchanged files along with other explanatory variables such as lines of code, age, prior faults etc. as predictors to predict the number of faults in a multiple release software system. More recently, code churn has been used to predict software binaries that may be fault prone [14]. In the paper, we are not looking at which files may have defects or faults, but he rather how long it will take for a defect fix will take to be accepted as part of a pull request.

5.3 Predicting Pull Request acceptance

More recently there has been interest in predicting pull request acceptance. Soares et al. [21], and Tsay et al. [25] looked at a variety of factors to see which one had on impact on pull request acceptance. More specifically, Terrell et al. [24] and Rastogi et al. [20] looked at gender or geographical location impacts a pull request acceptance. The work closest to our work is by Yu et al. [27], 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 which 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.

6 CONCLUSION

In this paper we have presented our PR completion time prediction system that has been deployed at Microsoft across several product families impacting several thousand developers. The deployment feedback has been overwhelmingly positive with a 73% comment resolution rate and a conservative savings of 44% of the developer time. We also present the set of features we use for building our PR completion time prediction model and its evaluation test results. We plan to scale the system to more product groups across Microsoft, measuring the telemetry and feedback from the system to continuously improve it and take into account user input. We also plan to work with GitHub to put this as a feature external to Microsoft to enable the broader software engineering community benefit from this.

ACKNOWLEDGEMENTS

We would like to thank Rahul Kumar, Tom Zimmermann, B. Ashok and Prof. Mei Nagappan for all their help with this work.

REFERENCES

[1] Azure Batch. https://azure.microsoft.com/en-us/services/batch/.

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

[3] GitHub. https://github.com/about.

[4] V. R. Basili, L. C. Briand, and W. L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Trans. Softw. Eng.*, 22(10):751–761, Oct. 1996.

[5] N. Bettenburg, M. Nagappan, and A. E. Hassan. Towards improving statistical modeling of software engineering data: Think locally, act globally! *Empirical Softw. Engg.*, 20(2):294–335, Apr. 2015.

[6] B. Boehm, B. Clark, E. Horowitz, J. Westland, R. Madachy, and R. Selby. Cost models for future software life cycle processes: Cocomo 2.0. *Annals of Software Engineering*, 1:57–94, 12 1995.

[7] B. W. Boehm. Software engineering economics. *IEEE Trans. Softw. Eng.*, 10(1):4–21, Jan. 1984.

[8] L. C. Briand, K. El Emam, D. Surmann, I. Wieczorek, and K. D. Maxwell. An assessment and comparison of common software cost estimation modeling techniques. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, pages 313–322, New York, NY, USA, 1999. ACM.

[9] L. C. Briand, J. Wüst, J. W. Daly, and D. V. Porter. Exploring the relationship between design measures and software quality in object-oriented systems. *J. Syst. Softw.*, 51(3):245–273, May 2000.

[10] L. C. Briand, J. Wust, S. V. Ikonomovski, and H. Lounis. Investigating quality factors in object-oriented designs: an industrial case study. In *Proceedings of the 1999 International Conference on Software Engineering (IEEE Cat. No.99CB37002)*, pages 345–354, May 1999.

[11] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, 20(6):476–493, June 1994.

[12] S. Chulani, B. Boehm, and B. Steece. Bayesian analysis of empirical software engineering cost models. *IEEE Trans. Softw. Eng.*, 25(4):573–583, July 1999.

[13] K. El Emam, S. Benlarbi, N. Goel, and S. N. Rai. The confounding effect of class size on the validity of object-oriented metrics. *IEEE Trans. Softw. Eng.*, 27(7):630–650, July 2001.

[14] L. Layman, G. Kudrjavets, and N. Nagappan. Iterative identification of fault-prone binaries using in-process metrics. In *ESEM*, 2008.

[15] L. Layman, N. Nagappan, S. Guckenheimer, J. Beehler, and A. Begel. Mining software effort data: preliminary analysis of visual studio team system data. pages 43–46, 01 2008.

[16] L. MacLeod, M. Greiler, M. Storey, C. Bird, and J. Czerwonka. Code reviewing in the trenches: Challenges and best practices. *IEEE Software*, 35(4):34–42, July 2018.

[17] T. Menzies, A. Butcher, D. Cok, A. Marcus, L. Layman, F. Shull, B. Turhan, and T. Zimmermann. Local versus global lessons for defect prediction and effort estimation. *IEEE Transactions on Software Engineering*, 39(6):822–834, June 2013.

[18] A. Mockus, P. Zhang, and P. L. Li. Predictors of customer perceived software quality. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 225–233, New York, NY, USA, 2005. ACM.

[19] T. J. Ostrand, E. J. Weyuker, and R. M. Bell. Where the bugs are. In *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '04, pages 86–96, New York, NY, USA, 2004. ACM.

[20] A. Rastogi, N. Nagappan, G. Gousios, and A. van der Hoek. 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, pages 22:1–22:8, New York, NY, USA, 2018. ACM.

[21] D. M. Soares, M. L. de Lima Júnior, L. Murta, and A. Plastino. Acceptance factors of pull requests in open-source projects. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, SAC '15, pages 1541–1546, New York, NY, USA, 2015. ACM.

[22] R. Subramanyam and M. S. Krishnan. Empirical analysis of ck metrics for object-oriented design complexity: implications for software defects. *IEEE Transactions on Software Engineering*, 29(4):297–310, April 2003.

[23] M.-H. Tang, M.-H. Kao, and M.-H. Chen. An empirical study on object-oriented metrics. In *Proceedings of the 6th International Symposium on Software Metrics*, METRICS '99, pages 242–, Washington, DC, USA, 1999. IEEE Computer Society.

[24] J. Terrell, A. Kofink, J. Middleton, C. Rainear, E. Murphy-Hill, C. Parnin, and J. Stallings. Gender differences and bias in open source: pull request acceptance of women versus men. *PeerJ Computer Science*, 3:e111, May 2017.

[25] J. Tsay, L. Dabbish, and J. Herbsleb. Influence of social and technical factors for evaluating contribution in github. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 356–366, New York, NY, USA, 2014. ACM.

[26] M. A. Vouk and K. C. Tai. Some issues in multi-phase software reliability modeling. In *Proceedings of the 1993 Conference of the Centre for Advanced Studies on Collaborative Research: Software Engineering - Volume 1*, CASCON '93, pages 513–523. IBM Press, 1993.

[27] Y. Yu, H. Wang, V. Filkov, P. Devanbu, and B. Vasilescu. Wait for it: Determinants of pull request evaluation latency on github. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, MSR '15, pages 367–371, Piscataway, NJ, USA, 2015. IEEE Press.