TU Delft

Delft University of Technology

Recommender Systems for DevOps

Maddila, C.S.

DOI
10.4233/uuid:769d3d81-8a84-4f59-80a6-2d237aa878a4

Publication date
2022

Document Version
Final published version

Citation (APA)
Maddila, C. S. (2022). *Recommender Systems for DevOps*. [Dissertation (TU Delft), Delft University of Technology]. https://doi.org/10.4233/uuid:769d3d81-8a84-4f59-80a6-2d237aa878a4

Important note
To cite this publication, please use the final published version (if applicable).
Please check the document version above.

Copyright
Other than for strictly personal use, it is not permitted to download, forward or distribute the text or part of it, without the consent of the author(s) and/or copyright holder(s), unless the work is under an open content license such as Creative Commons.

Takedown policy
Please contact us and provide details if you believe this document breaches copyrights.
We will remove access to the work immediately and investigate your claim.

This work is downloaded from Delft University of Technology.
For technical reasons the number of authors shown on this cover page is limited to a maximum of 10.

RECOMMENDER SYSTEMS FOR DEVOPS

RECOMMENDER SYSTEMS FOR DEVOPS

Dissertation

for the purpose of obtaining the degree of doctor
at Delft University of Technology
by the authority of the Rector Magnificus Prof.dr.ir. T. H. J. J. van der Hagen
chair of the Board for Doctorates
to be defended publicly on
Tuesday 20 December 2022 at 15.00 o'clock

by

Chandra Sekhar MADDILA

Master of Technology in Software Systems
Birla Institute of Technology and Sciences, Pilani, India
born in Jarajapupeta, Andhra Pradesh, India

This dissertation has been approved by the promotors.

Composition of the doctoral committee:

Rector Magnificus	Chairperson
Prof. dr. A. van Deursen	Delft University of Technology, Promotor
Prof. dr. N. Nagappan	Meta Platforms Inc. and IIIT-Delhi, Promotor
Dr. ir. G. Gousios	Delft University of Technology, Promotor
Independent members:	
Prof. dr. ir. D. Spinellis	Delft University of Technology and Athens University of Economics and Business
Prof. dr. M. A. Storey	University of Victoria
Prof. dr. A. Hanjalic	Delft University of Technology
Prof. dr. A. Zaidman	Delft University of Technology
Prof. dr. ir. F. A. Kuipers	Delft University of Technology, Reserve member

Keywords: DevOps, Recommender Systems, Artificial Intelligence, Machine Learning

Printed by: ProefschriftMaken, www.proefschriftmaken.nl

The author set this thesis in LATEX using the Libertinus and Inconsolata fonts. The thesis cover is derived from a template by Prof. Dr. K. P. Hart

An electronic version of this dissertation is available at
http://repository.tudelft.nl/.

Sometimes it is the people no one can imagine anything of who do the things no one can imagine.

Alan Turing

Contents

Acknowledgements xi

Summary xiii

1 Introduction 1
1.1 Background and Motivation . . . 2
1.1.1 DevOps . . . 2
1.1.2 Recommenders in DevOps . . . 3
1.2 Problem Statement . . . 4
1.2.1 Challenges and opportunities in DevOps . . . 4
1.2.2 Research Goals and Questions . . . 5
1.3 Solution Direction . . . 7
1.3.1 DevOps Recommenders in Action . . . 7
1.3.2 DevOps Recommender Evaluation . . . 9
1.3.3 A Conceptual Framework for DevOps Recommender Systems . . . 9
1.4 Research Methodology . . . 12
1.4.1 Technical action research . . . 12
1.4.2 Recommendation Effectiveness . . . 12
1.4.3 Open Science . . . 14
1.5 Origin of Chapters . . . 14

2 ORCA: Differential Bug Localization in Large-Scale Services 17
2.1 Introduction . . . 17
2.2 Related Work . . . 20
2.3 Overview . . . 22
2.3.1 System Development Lifecycle . . . 22
2.3.2 Post-Deployment Bugs . . . 23
2.3.3 Orca Overview . . . 26
2.3.4 Orca Scope . . . 27
2.4 Design . . . 27
2.4.1 Differential Code Analysis . . . 28
2.4.2 Build Provenance Graph . . . 29
2.4.3 Algorithm . . . 31

2.5 Implementation . . . 33
2.5.1 Data Loaders . . . 33
2.5.2 Background Analyses . . . 34
2.5.3 API Implementation . . . 34
2.5.4 Usage . . . 35
2.6 Evaluation . . . 36
2.6.1 Result Quality . . . 36
2.6.2 Reduction of OCE Workload . . . 38
2.6.3 Performance . . . 38
2.7 Conclusion and Future Work . . . 39

3 Nudge: Accelerating Overdue Pull Requests Towards Completion 41
3.1 Introduction . . . 42
3.2 Related Work . . . 45
3.3 Background: A Pull Request's Life Cycle . . . 46
3.4 Nudge System Design . . . 49
3.4.1 Design Overview . . . 49
3.4.2 Key Design Considerations . . . 50
3.5 Pull Request Lifetime Prediction . . . 52
3.5.1 Correlation Analysis . . . 53
3.5.2 Prediction Model . . . 55
3.6 pull request status determination . . . 56
3.6.1 Activity detection . . . 57
3.6.2 Actor Identification . . . 58
3.7 Implementation . . . 60
3.7.1 Nudge Service Architecture . . . 60
3.7.2 Azure DevOps . . . 61
3.7.3 Activity detection . . . 62
3.7.4 Actor identification . . . 63
3.7.5 Nudge Notification . . . 63
3.8 Evaluation . . . 63
3.8.1 Data collection and methodology . . . 64
3.8.2 RQ1: What is the accuracy of effort estimation models in predicting the lifetime of pull requests? . . . 67
3.8.3 RQ2: What is the impact of Nudge service on completion times of pull requests? . . . 68
3.8.4 RQ3: What are developers' perceptions about the usefulness of the Nudge service?. . . . 70
3.8.5 RQ4: Nudge at Scale . . . 73

3.9 Discussion . . . 73
3.9.1 Explicit Completion Times . . . 74
3.9.2 Interruptions . . . 74
3.9.3 Code review quality . . . 74
3.9.4 Simplifying Lifetime Prediction . . . 75
3.9.5 Addressing Nudge Limitations . . . 75
3.9.6 Threats to validity . . . 76
3.10 Conclusion . . . 76

4 ConE: A Concurrent Edit Detection Tool for Large Scale Software Development 79
4.1 Introduction . . . 80
4.2 Related Work . . . 83
4.3 Concurrent versus non-concurrent edits in practice . . . 85
4.3.1 Data Collection . . . 87
4.3.2 RQ1: Concurrent versus non-concurrent bug inducing edits . . . 88
4.3.3 RQ2: Edits in files versus bug fixes in files . . . 88
4.4 System Design . . . 91
4.4.1 Core Concepts . . . 91
4.4.2 The ConE Algorithm . . . 94
4.4.3 Default Thresholds and Parameter Tuning . . . 95
4.5 Implementation and deployment . . . 97
4.5.1 Core Components and Implementation . . . 97
4.5.2 ConE Deployment . . . 98
4.5.3 Notification Mechanism . . . 100
4.5.4 Scale . . . 100
4.6 Evaluation: Developers perceptions about ConE's usefulness . . . 101
4.6.1 Comment resolution percentage . . . 101
4.6.2 Extent of interaction . . . 102
4.6.3 User Interviews . . . 103
4.6.4 Representative Quotes . . . 104
4.6.5 Factors Affecting ConE Appreciation . . . 106
4.7 Discussion . . . 107
4.7.1 Outlook . . . 107
4.7.2 Threats to validity . . . 108
4.8 Conclusion and future work . . . 108

5 Nalanda: A Socio-Technical Graph Platform for Building Software Analytics Tools at Enterprise Scale 111
5.1 Introduction . . . 112
5.2 Building The Nalanda Graph . . . 113
5.2.1 Nalanda's Graph Schema . . . 114
5.2.2 Data Collection and Graph Construction . . . 115

5.2.3 Data consistency and self-healing . . . 117
5.2.4 Scale . . . 117
5.3 Indexing Nalanda for Information Retrieval . . . 118
5.3.1 The Nalanda Artifact Index . . . 118
5.3.2 The Nalanda Expert Index . . . 119
5.3.3 Scale . . . 120
5.4 Nalanda Applications (I): The MyNalanda Portal . . . 120
5.4.1 MyNalanda Motivation . . . 120
5.4.2 The MyNalanda Homepage . . . 120
5.4.3 MyNalanda Usage . . . 122
5.4.4 MyNalanda Evaluation: Perceived Usefulness . . . 122
5.5 Nalanda Applications (II): Artifact and Expert Recommender . . . 125
5.5.1 The Nalanda Ranking Algorithm . . . 126
5.5.2 Implementation . . . 126
5.5.3 Quantitative Evaluation . . . 128
5.5.4 User Perception . . . 129
5.6 Discussion . . . 130
5.6.1 Outlook . . . 130
5.6.2 Threats and Limitations . . . 131
5.7 Related work . . . 132
5.8 Conclusion . . . 133

6 Conclusion **135**
6.1 Contributions . . . 135
6.2 Reflection on the Research Questions . . . 137
6.2.1 Research Question 1: What are the modalities of recommenders that need to be built to support modern DevOps? . . . 137
6.2.2 Research Question 2: How to assess the efficacy of smart recommenders and their impact on DevOps processes? . . . 138
6.2.3 Research Question 3: How to build a general-purpose and scalable data framework for developing smart DevOps recommenders? . . . 140
6.3 Concluding Remarks . . . 141

Curriculum Vitæ **159**

List of Publications **161**

Acknowledgements

I am deeply grateful to my supervisor, prof. dr. Arie van Deursen, for his guidance, support, and encouragement throughout my PhD journey. His expertise and insights have been invaluable in shaping the direction of my research and in helping me to grow as a researcher. I am grateful for the endless hours of discussion and feedback, and for always pushing me to think critically and deeply about my work. Working with Arie has been a life changing experience and had a profound impact on the way I conduct research, my thought process, and critical thinking.

I would like to express my gratitude to Dr. Nachi Nagappan and Dr. Georgios Gousios who are my co-supervisors. I could not have asked for better co-supervisors than Nachi and Georgios, who greatly influenced my work, helped me strike the delicate balance between rigor and relevance in my research.

I would like to thank the members of my thesis committee for their valuable feedback and insights. Their expertise in Recommender systems, Empirical Software Engineering, and Programming Languages has been instrumental in the development of this thesis, and I am grateful for their guidance and support.

I'd like to thank my collaborators and coauthors from Microsoft Research, Microsoft, TU Delft, and University of Victoria: B. Ashok (BASH), Nachi Nagappan, Nicole Forsgren, Tom Zimmermann, Christian Bird, Chetan Bansal, Denae Ford Robinson, Kim Herzig, Margaret-Anne Storey, Georgios Gusios, and Arie van Deursen.

I feel humbled and honored to be associated with you, mentored by you, and collaborated with you. You all left lasting impact on how I approach problems, conduct research, and create great impact while being humble.

SUMMARY

The software development life cycle (SDLC) for a developer has increased in complexity and scale. With the advent of DevOps processes, the gap between development and operations teams reduced significantly. Developers are now expected to perform different roles from coding to operational support in the new model of software development. This shift demands the evolution and improvement of software development practices and deliver products at a faster pace than organizations using traditional software development and infrastructure management processes. As a consequence, the demand for more intelligent and context sensitive *DevOps* tools and services that help developers increase their efficiency is increasing. A lot of research went into developing recommenders for DevOps, by leveraging the advancements made by the recommender system community. However, a lot of existing tools still work in 'silos' and does not take into account a holistic view of DevOps processes and the data generated at phase of the DevOps lifecycle while making recommendations.

By contrast, in this thesis, we propose a unified framework to develop recommenders for DevOps: perform data collection, building the models, deploying them, and evaluating the effectiveness of such recommenders in large-scale cloud development environments quickly and efficiently. We study the effect of such recommenders on the DevOps processes by performing empirical research and mixed method approaches (qualitative and quantitative analyses) on each of the deployed recommenders to better understand the productivity gains and the impact created by them.

Our results show that developers benefit greatly from smart recommenders such as Nudge, ConE, Orca, and MyNalanda. We also show, through rigorous experiments, technical action research methods, and empirical analyses that these recommenders provide as much as 65% gains in terms of change progression and 73% accuracy for root causing the service incidents automatically. We also conduct large scale surveys and interviews to support our empirical analysis and quantitative results. Our unified data framework and the platform we developed for building these recommenders is generic enough and encourages reusability of vital functions of such recommenders systems, such as data collection, model training, inference, deployment, and evaluation.

1

Introduction

The software development life cycle (SDLC) for a developer has increased in complexity and scale, which induced a fundamental shift amongst software developers and engineers in the past few years. DevOps, which is the combination of practices, and tools that increases an organization's ability to deliver applications and services at high velocity, accelerated this shift. This shift demands the evolution and improvement of software development practices and delivery of products at a faster pace than organizations using traditional software development and infrastructure management processes. Due to greater availability of computing, storage, better tooling, and the necessity to react, developers are constantly looking to increase their velocity and throughput of developing and deploying changes. Consequently, there is a great need for more intelligent and context-sensitive DevOps tools and services that help developers increase their efficiency while developing and debugging. There is a need for building recommender systems, which aim at supporting developers in navigating large information spaces and getting instant suggestions that might be helpful to solve a particular development task. Given the vast amounts of heterogeneous data available from the SDLC, such intelligent tools and services (recommenders) can now be built and deployed at a large scale to help developers achieve their goals and be more productive.

In this thesis, we develop tools, which we refer to as 'recommenders', to help mitigate various problems that exist in the DevOps lifecycle. We perform empirical research and mixed method approaches (qualitative and quantitative analyses) on each of the deployed recommenders to better understand the productivity gains and the impact created by them. We propose a unified data framework to develop recommenders for DevOps: performing data collection, building the models, deploying them, and evaluating the effectiveness of such recommenders in large-scale cloud development environments quickly and efficiently. We also discuss the architecture and implementation

details of the large-scale data platform, which provides the necessary infrastructure (back-end systems and front-end bots) to develop and scale these recommenders.

1.1. Background and Motivation

1.1.1. DevOps

DevOps is defined as a set of practices and tools that automate and integrate the processes between software development and operations teams (DEVelopment + OPerationS) (Ebert et al., 2016). It emphasizes large-scale systems possibly deployed in the cloud, continuous integration and deployment, team empowerment, cross-team communication, and collaboration through rigorous review processes, and technology automation.

In a traditional software development environment, developers seek to push out software faster. Operations, on the other hand, know that moving fast could pose a risk of destabilizing the system. DevOps was created to resolve this dilemma by integrating all the stake holders associated with software development and deployment into a coherent and automated workflow with a shared focus, which is rapid delivery of high-quality software that meets all user requirements while maintaining the integrity and stability of the entire system (Forsgren et al., 2018).

Under a DevOps model, development and operations teams are no longer 'siloed.' Sometimes, these two teams merge into a single team where the engineers work across the entire software development life cycle — from development and test to deployment and operations. DevOps teams use tools to automate and accelerate processes, which helps to increase the speed of deployment, reliability, and efficiency (Bass et al., 2015).

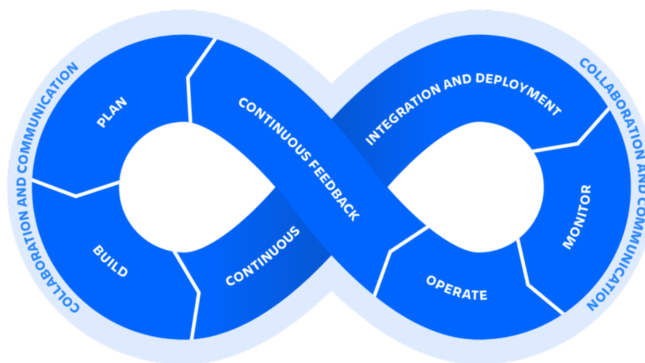


Figure 1.1: DevOps Lifecycle

Due to the continuous nature of DevOps, the DevOps life cycle exhibits a pattern of an infinite loop, as opposed to a traditional software development lifecycle

such as a waterfall model. Figure 1.1 shows the phases of DevOps lifecycle[1]: plan, build, continuous integration and deployment (CI / CD), monitor, operate, and continuous feedback. These phases are related to each other. Despite appearing to flow sequentially, the loop symbolizes the need for constant collaboration and iterative improvement throughout the entire lifecycle.

1.1.2. Recommenders in DevOps

Recommendation systems aim to help people find information and make decisions where they lack experience or can't consider all the data at hand (M. P. Robillard et al., 2014). These systems combine many computer science research areas, such as Information Retrieval (IR), Machine Learning (ML), and Big data systems to proactively tailor suggestions that meet users' particular information needs and preferences. Web-based applications are the primary focus area for recommender systems. Amazon.com's item recommendations, Facebook's friend recommendations, and movie recommendations by Netflix are some of the popular recommender systems used in practice today.

The challenges software developers face today in navigating large-scale information and data sources in software development environments have similarities to those of web-based use cases. For example, an on-call engineer (OCE) working on resolving an incident in a large-scale cloud service will need to mine an incessant amount of logs, disparate data sources, and dependency graphs to find out the commit that has caused the incident.

With the advent of DevOps and large-scale cloud services, there is a substantial amount of data generated in the process of developing those services and operating them. Consequently, there is a greater need today to build systems that will help developers by *generating knowledge from information or data*. Such systems help make the DevOps processes efficient and help organizations deliver software at a faster pace while not losing on reliability.

The software engineering research community has been working on understanding the aspects of Recommender Systems for Software Engineering (RSSE). Projects like CROSSMINER (Rocco et al., 2021) are an important first step toward understanding the aspects of RSSEs such as requirement elicitation, development, and evaluation of IDE-based RSSEs. Tools such as CodeBroker (Ye & Fischer, 2005) to help to surface reuse opportunities, Expertise Browser (Mockus & Herbsleb, 2002) for locating experts, Strathcone (Holmes et al., 2006) to decide what examples to use, and ParseWeb (Thummalapenta & Xie, 2007) to help decide what call sequences to make. These are built by bringing the concepts of RSSEs to practice. Beyond providing support while authoring code, tools that can recommend replacement methods for adapting code to a new library version (Dagenais & Robillard, 2011) and help during debugging by finding code and people related to a bug fix (Ankolekar et al., 2006).

Robillard et al argue that recommender systems aim at helping developers find the

[1] Image courtesy: https://www.atlassian.com/devops

information they should know about and evaluate alternative decisions, RSSEs span a wide spectrum of software engineering tasks and practically unbounded amounts of development data (M. P. Robillard et al., 2014). They lay out a foundation for the design dimensions of the recommender systems for software engineering comprising: the nature of the context, the recommendation engine, output modes, and cross-dimensional features. Our work is greatly influenced by the foundations laid by them.

1.2. Problem Statement

1.2.1. Challenges and opportunities in DevOps

Modern software development life cycles, which are inspired by the DevOps model, are extremely rapid. Rapid cycles generate more code changes. Many code changes generate lots of data about these code changes, their development, and their effect on production. This fundamental shift can be attributed partly to the nature of products that developers work on, i.e., large-scale cloud services such as Google, Amazon, Facebook, etc. Partly, this shift can also be attributed to the advanced development tools and frameworks available today that facilitate rapid prototyping and deployment, such as cloud platforms, continuous integration, continuous deployment systems, etc. There is significantly more data generated by these systems, during their development, testing, and deployment. In this thesis, we seek to explore how to leverage that data for the benefit of developers and software engineering processes.

For example, given a set of changes that have been authored and reviewed by engineers in a single repository, one can easily determine the notion of an *expert* or *owner* for a given area of the source code. These concepts can then be leveraged to create a bot that automatically suggests reviewers and experts that should be included in future changes to specific areas of the code (Zanjani et al., 2016). Similarly, we depend on numerous large-scale services in software services. These services are complex and extremely dynamic as developers continuously commit code and introduce new features, fixes, and, consequently, new bugs. Hundreds of commits may enter deployment simultaneously. Therefore, one of the most time-critical, yet complex tasks towards mitigating service disruption is to localize the bug to the right commit. These are examples of some of the *recommenders* that can be built by leveraging the data generated during the DevOps process.

Researchers have been developing recommender systems to help software developers and the DevOps processes (Lebeuf et al., 2018; Shull et al., 2007). Most of these recommenders are developed and deployed in silos. They are limited by their ability to share the data, modeling or training and deployment infrastructure, and their ability to transfer the heuristics learned from one system to another. This limits the ability to *quickly build and deploy* the recommenders, collect feedback and telemetry, and iterate over them in large-scale software development environments.

The challenges that come with scaling these recommenders are not well studied due to the lack of deployment and evaluation under large-scale, realistic industrial

conditions. Partly, this is due to the fact that a lot of these recommenders are expensive to build and maintain. Furthermore, the operational costs prevent them from scaling to thousands of repositories in a fiscally responsible manner. Therefore, we see a great opportunity to propose, build, and evaluate recommenders in large-scale industrial settings. In that process, we define the frameworks and systems that help standardize the process of building, scaling, and democratizing the process of building such recommenders, which we discuss thoroughly in the rest of the thesis.

1.2.2. Research Goals and Questions

The goal of this thesis is threefold. First, we seek to provide new recommender systems to support modern DevOps activities. Second, we target a method and framework for evaluating these recommenders. Third, we aim at offering a generic model for building such recommenders.

Recommender systems to support modern DevOps

Building new recommenders to help the DevOps processes and various stakeholders involved in the DevOps lifecycle operate efficiently is one of the primary goals of the thesis.

To arrive at such recommenders, we need to develop a deeper understanding of the problems faced by DevOps practitioners in large-scale cloud development environments. We analyze the large amounts of data generated as part of the DevOps processes. Furthermore, we conduct systematic user studies and interviews with various stakeholders to understand the problems associated with DevOps in large-scale cloud environments. Some problems that repeatedly surfaced were related to root causing service disruptions, delays associated with change progression, information overload, and satisfying the information needs of developers.

Another important observation we make is that because these recommenders need to be smart (context-aware, leveraging past data to make a future prediction), they are extremely data-hungry. Furthermore, these recommenders need a holistic view of the data generated across the DevOps lifecycle to make better predictions and judgments. For example, a recommender system for automated bug (incident) localization needs to have data from the code authoring side (commits, pull requests) and deployment side (operational logs, stack traces, etc.) In a way, the systems need to replicate the behaviors or capabilities of DevOps engineers, i.e., developing a holistic understanding of the system.

After developing a deeper understanding of the problem space, challenges, and opportunities, we developed various recommenders listed below. Figure 1.2 shows the data platforms we construct off the data collected from the DevOps lifecycle, and how our solutions fit into the same DevOps lifecycle.

- ORCA (Chapter 2): Helps with automated root causing of incidents by finding the code change that introduced the bug.

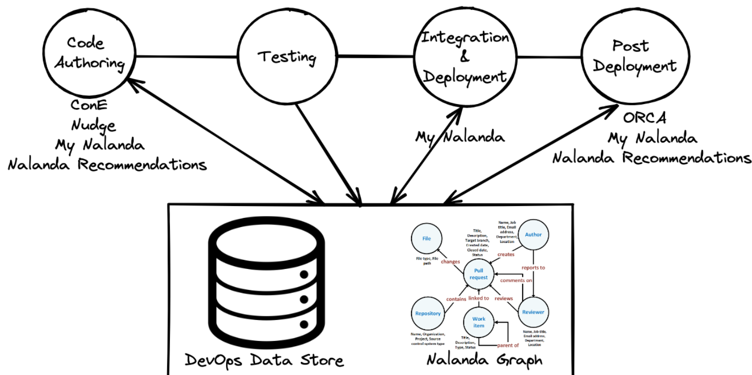


Figure 1.2: DevOps lifecycle, data platforms, and the solutions

- Nudge (Chapter 3): Helps with accelerating the pull request completion time by sending timely and context-aware reminders.
- ConE (Chapter 4): Facilitates early intervention for identifying potential merge conflicts and notifying the developers involved.
- Mynalanda (Chapter 5): Helps developers dealing with information overload and information discovery by providing a news-feed of their activity.
- Nalanda Recommendations (Chapter 5): Provides recommendations about the related items (pull requests, work items), and subject matter experts (SMEs) that will help in accomplishing a task.

Evaluating Recommender systems

Next to deploying our recommenders, we focus on understanding the effectiveness of the smart recommenders on the software development and DevOps processes. This includes proposing the right set of metrics to measure the efficacy of the recommender. The process of coming up with the metrics that track the improvements is not trivial. We start by understanding the problem domain and conducting interviews with key stakeholders, such as engineering managers, program managers, and the developers themselves. We perform this exercise in an iterative fashion until it is clear that the metrics that we are tracking are adding business value and helping us measure the intended research outcomes.

Furthermore, we also work on understanding the user perception of the usefulness and accuracy of the recommendations by employing a mixed-methods approach, which includes both qualitative and quantitative studies. While metrics reveal one side of the story, understanding the experience of the developers and the human factors of employing smart recommenders in their software development workflow requires rich data obtained from methods such as surveys, focus groups, and interviews.

To that end, we conduct extensive studies about all of our smart recommenders and pass the feedback from the field back to the design process.

A generic model for building recommenders

Lastly, we investigate whether building a scalable data platform and operationalizing smart recommenders at scale is plausible. We study various aspects of scalable distributed system designs, and the processes involved in deploying smart recommenders in a non-intrusive fashion. Thus, we can explore optimal paths toward building such recommenders. Currently, there exist tools or solutions that solve specific problems (as explained in Section 1.1.2). Yet unfortunately, these tools cannot leverage the data ingestion, training, inference, and recommendation infrastructure common to such tools. To address this, we conduct a deep dive and come up with a design that is generic enough to build multiple smart recommenders using the same infrastructure. This not only speeds up the development of recommenders but also yields a better return on investment by amortizing the fixed costs over multiple tools.

Research Questions

To summarize, we aim to address three higher-level research questions.

RQ 1 What are the modalities of recommenders that need to be built to support modern DevOps?

RQ 2 How to assess the efficacy of smart recommenders and their impact on DevOps processes?

RQ 3 How to build a general-purpose and scalable data framework for developing smart DevOps recommenders?

1.3. Solution Direction

1.3.1. DevOps Recommenders in Action

Developing the right set of recommenders that can have a real impact on DevOps processes requires a deeper understanding of the problems faced by DevOps practitioners in large-scale cloud development environments.

We observe that root causing service disruptions or incidents is a major pain for the developers. In large-scale services there exist so many complex service dependencies. These services are complex and extremely dynamic as developers continuously commit code and introduce new features, fixes, and, consequently, new bugs. Hundreds of commits may enter deployment simultaneously. Therefore one of the most time-critical, yet complex tasks towards mitigating service disruption is to localize the bug to the right commit. To that end, we created ORCA (Online Root Cause Analysis). We explain details about ORCA in Chapter 2.

Next, we shifted our focus to the left side of the DevOps lifecycle. We observed that Pull requests and code reviews are a key part of the DevOps processes 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. 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, 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, the review process could be slowed down significantly. Hence, if the pull request's author and reviewers do not actively engage, the pull requests can remain open for a long time, slowing down the coding process and possibly causing side effects such as merge conflicts. To address this, we created Nudge, which is a large-scale machine learning service that helps make progress on stale pull requests. We explain details about Nudge in Chapter 3.

As we dig deeper, we started to realize that there exist a plethora of problems in the code authoring side of the DevOps world. As the *shift left* movement is growing in prominence (Colavita, 2016), we focused more on developing recommenders to strengthen the left side of the DevOps lifecycle.

Modern, complex software systems are being continuously extended and adjusted. The developers responsible for this may come from different teams or organizations and may be distributed over the world. This may make it difficult to keep track of what other developers are doing, which may result in multiple developers concurrently editing the same code areas. This, in turn, may lead to hard-to-merge changes or even merge conflicts, logical bugs that are difficult to detect, duplication of work, and wasted developer productivity. To address this, we built CONE (CONcurrent Edit detection) service. We explain details about CONE in Chapter 4.

While developing the recommenders for helping developers, we found that a lot of them are 'reactive' i.e., they will start helping the developers after the coding activity is completed. We shifted further left to understand the problems associated with code authoring in the first place. Software development is information-dense knowledge work that requires collaboration with other developers and awareness of artifacts such as work items, pull requests, and files. With the speed of development increasing, information overload is a challenge for people developing and maintaining these systems. Finding information and people is difficult for software engineers, especially when they work in large software systems or have just recently joined a project.

To address the challenges associated with satisfying the information needs of software developers and information overload, we built Nalanda. Nalanda is a large socio-technical graph that indexes artifacts of software development (files, pull requests, tasks, etc), relationships that exist between these artifacts, and the associations that exist between artifacts and software developers. We use this knowledge to make context-sensitive recommendations about related tasks, pull requests that solved similar problems, subject matter experts (SMEs). Additionally, we also show the software development activity of developers, their colleagues, and teammates in one place in

form of a news-feed-based service. Details about Nalanda are discussed in Chapter 5.

The above observations we made while working closely with various stakeholders in the DevOps lifecycle form an important basis for the thesis. These specific recommenders explained above serve to answer all of our Research Questions and Research Question 1 (RQ1) in their specific context.

1.3.2. DevOps Recommender Evaluation

Gauging the impact of a recommender is important, yet challenging. In order to understand the impact of a recommender, a ground truth needs to be established, which is not always available or clearly defined. For example, when we make a recommendation to make progress on a stale pull request, it is almost impossible to programmatically determine that someone completed that pull request because of our nudge.

We follow a two-pronged approach for determining the impact of recommenders on software development practices, which forms the basis for our RQ2.

1. *Implicit feedback.* We measure the impact of a recommender by looking at the data. For recommenders like Nudge, it is the number of pull request completed within X days of nudging where X is a parameter, such as 1 day, 3 days, 7 days, etc. For recommenders like ORCA, we look at the actual root causes of incidents in the last X months and check how many of them were caught by ORCA. Similarly, for Nalanda recommendations, we track whether people are interacting with the recommendations by gathering data about click-through logs.

2. *Explicit feedback.* We gather explicit feedback through mechanisms such as providing a thumbs-up/down button for each recommendation, providing a provision to enter verbose feedback, etc. Additionally, for recommenders like Nudge, ConE, and Nalanda, we also conducted large-scale user studies, surveys, and interviews.

1.3.3. A Conceptual Framework for DevOps Recommender Systems

In order to address the challenges and opportunities discussed in the previous section, we propose a conceptual framework for DevOps Recommender Systems, inspired by the Sankie recommender infrastructure discussed by Kumar et al (Kumar et al., 2019a).

In contrast to the siloed approaches employed so far, we propose a *unified* perspective on leveraging the data generated by the software development life cycle (SDLC). This is a novel and powerful idea. It also enables us to envision impactful scenarios such as online and automated root causing (Chapter 2) by joining the data generated in the coding phase of SDLC with the data generated in the deployment and post-deployment phases. To that end, we propose a unified AIOps (AI-assisted DevOps)

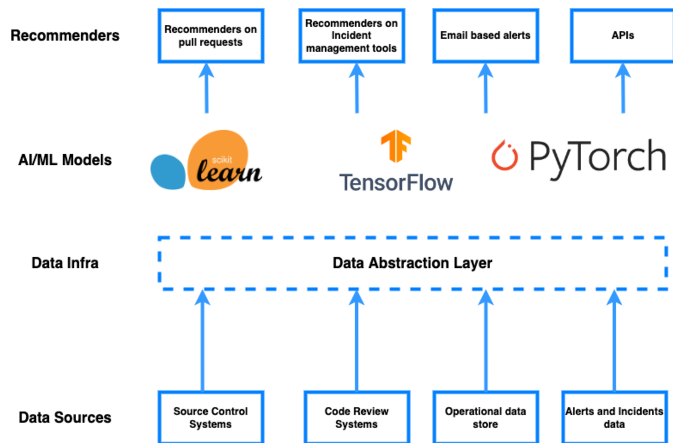


Figure 1.3: Conceptual framework overview

framework for developing smart recommenders, and methods to measure the impact of such recommenders on software development practices.

Building large-scale data ingestion and recommendation platform is a non-trivial task, requiring various components to work in sync. Figure 1.3 shows an overview of the data model and the recommendation system, which forms the basis for our Research Question 3 (RQ3). Some of the major components of our framework are listed below.

1. **Data Abstraction Layer** — To make the data platform extensible, a data abstraction layer is utilized to encode various DevOps concepts like pull requests, code reviews, builds, alerts, etc. collected from data sources like source code repositories, code review systems, operational data stores, and incident management systems.

2. **AI / ML Models** — We need to train models and learn heuristics from the data exposed through the data abstraction layer. For that, we leverage machine learning frameworks, such as Scikit learn (Pedregosa et al., 2011), Tensorflow (Martín Abadi et al., 2015), and PyTorch (Paszke et al., 2019).

3. **Recommenders** — The data platform provides recommendations and exposes rules learned through the training process using a set of self-service APIs. These APIs can further be integrated into pull requests, incident management tools, and email-based alerts.

Design Principles

We need to keep certain design principles in mind when designing such a large-scale data framework and recommendation system for DevOps. Guiding principles for the framework include.

1. **Extensibility** — The data platform ingests data from various heterogeneous data sources like source control, build pipeline, cloud services etc and implementations like Azure DevOps, GitHub etc. This makes it critical to have a shared and generalized schema to avoid custom and redundant implementations.

2. **Scalability** — The framework needs to be scalable both in terms of data processing and, also, onboarding more repositories. For this, we leverage the auto-scale and elastic capabilities provided by cloud deployment for which we leverage Microsoft Azure. Auto-scale allows us to scale the compute nodes up and down automatically based on the incoming workload, reducing manual effort and also the COGS (cost of goods served). Using elastic pooling in SQL Azure allows us to share resources and costs between multiple databases without compromising on performance and latency.

3. **Real time processing** - In the DevOps world, there are several time-critical events like code reviews, code check-ins, deployments, etc. In order to make AI-assisted recommendations for these scenarios, data platform bots should have the ability to process events in real-time (or near real-time) and make recommendations.

A *recommender* is a component that provides intelligent and actionable recommendations to engineers and developers. A recommender has three major functions as explained below.

1. Training: the training aspect of a recommender in the data platform provides the basic function of gathering data from the database, massaging it, applying the algorithm of choice, and then persisting the results to the database in a structured manner.

2. Recommending: Given an item such as a change or a pull request, this function is responsible for generating recommendations that are applicable to the given item. For example, based on the historic data of change progression, the recommender we will discuss in Chapter 3 believes that it is time to *nudge* the change blocker to make progress on the pull requests they are blocking. This specific recommendation will be generated and passed as a comment on the pull request.

3. Evaluating: Any recommender should support a fundamental concept of *evaluating* the recommender with respect to a set of ground truth data points that

are either acquired from the data or human encoded. Using the example of a recommender we will discuss in Chapter 3, if we nudge a change blocker (the author or the reviewer of a pull request), they may or may not respond. Even when they respond, it might not be because of the Nudge comment. Therefore, it is important to understand the impact of the recommendation by devising a strategy. Some strategies we employed are mixed-method approaches and randomized A/B testing for evaluating the efficacy of such recommenders.

1.4. Research Methodology

1.4.1. Technical action research

In technical action research (Wieringa, 2014), a newly designed artifact is tested in the field by using it to help a client. Technical action research is like a regular case study (Runeson & Höst, 2009; Yin, 2008), called 'single mechanism experimentation' by Wieringa, but with the additional goal of actually helping a client in the field.

In technical action research, we follow the 'design cycle', i.e. the creation, design, and validation of artifacts based on requirements and stakeholder goals. Then, we follow the empirical cycle to investigate artifacts in context and present the different elements of research problem analysis, research setup, and data analysis. This way we get to combine action research and design science by starting with a concrete problem in an organization, then applying an artifact to improve the problem, and finally reflecting on lessons learned. The aim of these combinations is to reduce the tension between relevance and rigor. Essentially, the aim of this way of using action research in design science is to bridge the gap between the idealizations made when designing the artifact and the concrete conditions of practice that occur in real-world problems.

Given our ability to *deploy* our solutions among developers, in production, we are able to rapidly prototype our solutions and employ technical action research methods to collect feedback. We then use that feedback to keep improving the efficacy and accuracy of our recommenders in an iterative fashion.

1.4.2. Recommendation Effectiveness

Empirical Software Engineering is a field of software engineering focused on gathering data, through measurements and experiments, to build theories about processes involved in software engineering. We use a mixed-method approach to answer our research questions, employing surveys, controlled experiments (A/B tests), interviews, and mining software repositories. In this section we explain the main characteristics of each research method we used:

- **Mining Software Repositories (MSR)**. MSR is a field of software engineering research in which researchers extract data from software repositories. We mine Version Control Systems (VCS), code review systems, Continuous Integration and Continuous Deployment pipelines (CI / CD), and so on, to extract informa-

tion about different aspects of the code and how it evolves over time. We then use that information to perform empirical analyses and A/B tests, which help us in understanding the impact of our recommenders on software development practices.

We also define key metrics that help us in tracking the effectiveness of our recommenders. Some of the metrics we employ are pull request completion time, time to mitigate incidents, the number of merge conflicts prevented, developer satisfaction, etc.

- **Interviews and Surveys**. While researchers can understand what happens in a software system, understanding why that happens is often not possible by only observing the data. To obtain richer data to answer this kind of question we use interviews: we can learn why something happened by talking to developers, we can ask their opinions on certain behavior, and understand the decision-making process behind a change. Furthermore, even though interviews are an important source of information, the results might not be generalizable: hence, we also employ surveys to challenge and expand our results.

  In Chapter 5, we conduct a large-scale survey with 2,000 plus participants. We intend to understand the usefulness of a news feed-based solution for aggregating the software development activity across repositories that a developer works in. We also leverage the survey to understand the user perceptions about various parts of the MyNalanda solution.

- **Randomised Control Trial**. In this dissertation, we devise and run a randomised control trial. The randomised control trial (RCT) is a trial in which subjects are randomly assigned to one of two groups: one (the experimental group) receiving the intervention that is being tested, and the other (the comparison group or control) receiving an alternative (conventional) treatment (Kendall, 2003). The two groups are then followed up to see if there are any differences between them in outcome. The results and subsequent analysis of the trial are used to assess the effectiveness of the intervention, which is the extent to which a treatment, procedure, or service does patients more good than harm. RCTs are the most stringent way of determining whether a cause-effect relation exists between the intervention and the outcome.

  In Chapter 3 we study the effect of one of the recommenders named Nudge: in this case, the variable being tested is the time it takes to complete pull requests. We test whether the Nudge notifications yielded the intended benefits of accelerating the pull request completion time.

We employ a series of mixed-methods studies that combine several of the above techniques to answer one research question. Particularly for validating the accuracy of survey answers, we triangulate answers in questionnaires corresponding to data extracted from the Azure DevOps telemetry. For recommenders like Nudge (Chapter

Table 1.1: Research methods used for each study

Study	Chpt.	Quant. Analysis	Survey	Interviews
ORCA	2	yes	no	yes
Nudge	3	yes	no	yes
ConE	4	yes	no	yes
Nalanda (MyNalanda)	5	yes	yes	yes
Nalanda (Recommendations)	5	yes	yes	yes

3) we also conduct a randomized trial with the A/B tests. Table 1.1 gives an overview of which research methods we employed for each of the individual studies.

1.4.3. Open Science

We chose the technical action research approach to prioritize realism and relevance. This helped us conduct methodical research, operationalize the recommenders, collect feedback, and repeat the loop. This also helped us demonstrate the impact and benefits of our research to various stakeholders and eventually scale the recommenders to thousands of developers.

As a consequence of our deliberate choice to go for a high level of realism and relevance, that can only be obtained through (technical) action research, we had to compromise on some aspects of open science. The consequences are that:

1. The tools produced are tightly connected to the Microsoft infrastructure, and cannot be used in isolation.
2. All data collected is company sensitive and proprietary and cannot be made publicly available.

Note, however, that the algorithms, methods, and design choices for our tool are described in substantial detail, enabling replication in either other organizations or in the open-source domain.

1.5. Origin of Chapters

Table 1.2 illustrates the connection between the chapters in this thesis and the research questions. In the following, we detail each chapter.

All chapters, except Chapter 5 of this thesis have been published in peer-reviewed journals and conferences. As a result, each chapter is self-contained with its background, related work, and results section. Unless otherwise specified, the first author is the main contributor to each work. The first three authors contributed equally to the ORCA work (Chapter 2). The origin of each chapter is explained below:

- **Chapter 2** We published the paper 'Orca: Differential Bug Localization in Large-Scale Services.' at USENIX OSDI 2018, with Ranjita Bhagwan, Rahul

Table 1.2: Relation between research questions and chapters

RQ	Description	Chapters
RQ1	What are the modalities of recommenders that need to be built to support modern DevOps?	1, 3, 4
RQ2	How to assess the efficacy of smart recommenders and their impact on DevOps processes?	2, 5
RQ3	How to build a general-purpose and scalable data framework for developing smart DevOps recommenders?	2, 3, 4, 5

Kumar, Chandra Maddila, Adithya Abraham Philip as co-authors. This paper has received *Jay Lepreau best paper award* . The first three authors contributed equally to this work.

- **Chapter 3** We published the paper ‘Predicting pull request completion time: a case study on large scale cloud services.’ at the European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE) 2019 industry track, with Chandra Maddila, Chetan Bansal, Nachiappan Nagappan as co-authors.

  A subsequent publication, by making substantial improvements to the Nudge service, with the title ‘Nudge: Accelerating Overdue Pull Requests Towards Completion’, was accepted at a journal ACM Transactions on Software Engineering and Methodology (TOSEM) in 2022, with Chandra Maddila, Sai Surya Upadrasta, Chetan Bansal, Nachiappan Nagappan, Georgios Gousios, Arie van Deursen as co-authors.

- **Chapter 4** We published ‘ConE: A Concurrent Edit Detection Tool for Large Scale Software Development’ as a journal-first paper at ACM Transactions on Software Engineering and Methodology (TOSEM) in 2021, with Chandra Maddila, Nachiappan Nagappan, Christian Bird, Georgios Gousios, and Arie van Deursen as co-authors.

  The same paper was also presented at ESEC/FSE 2022 as a journal first paper.

- **Chapter 5** A paper with the title ‘Nalanda: A Socio-Technical Graph Platform for Building Software Analytics Tools at Enterprise Scale’ is under review at ESEC/FSE 2022 industry track, with Chandra Maddila, Suhas Shanbhogue, Apoorva Agrawal, Thomas Zimmermann, Chetan Bansal, Nicole Forsgren, Divyanshu Agrawal, and Arie van Deursen as co-authors.

2

ORCA: DIFFERENTIAL BUG LOCALIZATION IN LARGE-SCALE SERVICES

Today, we depend on numerous large-scale services for basic operations such as email. These services are complex and extremely dynamic as developers continously commit code and introduce new features, fixes and, consequently, new bugs. Hundreds of commits may enter deployment simultaneously. Therefore one of the most time-critical, yet complex tasks towards mitigating service disruption is to localize the bug to the right commit.

This chapter presents the concept of differential bug localization *that uses a combination of differential code analysis and software provenance tracking to effectively pin-point buggy commits. We have built Orca, a customized code search-engine that implements differential bug localization. Orca is actively being used by the On-Call Engineers (OCEs) of a large enterprise email and collaboration service to localize bugs to the appropriate buggy commits. Our evaluation shows that Orca correctly localizes 79% of bugs for which it has been used. We also show that it causes a 4x reduction in the work done by the OCE.*

2.1. INTRODUCTION

Orion[1] is a large enterprise email and collaboration service that supports several millions of users, runs across hundreds of thousands of machines, and serves millions of requests per second. Thousands of developers contribute code to it at the rate of

[1]Name changed for anonymity

hundreds of commits per day. Dozens of new builds are deployed every week. Software bugs are bound to be common in such a complex and dynamic environment. It is critical to detect and promptly localize such bugs since service disruptions lead to customer dissatisfaction and significantly lower revenues (Infonetics, 2016).

When a service disruption happens because of a software bug, the first-step towards mitigating its effect is to localize the responsible bug to the right commit. We call this *commit-level bug localization*. This is a non-trivial task since the intense pace of development demands that multiple commits be aggregated into a single deployment. In addition, commit-level bug localization needs to happen as quickly as possible so that buggy commits can be reverted promptly thereby restoring service health. About half of all Orion's service disruptions are caused by software bugs.

Unfortunately, bug localization in large services such as Orion is a cumbersome, time-consuming, and error-prone task. The *On-Call Engineers (OCEs)* are the first points-of-contact when a disruption occurs, and they are responsible for bug localization. Though knowledgeable, on-call engineers can hardly be expected to have complete and in-depth understanding of all recent commits. Moreover, bugs that emerge after deployment are complex and often non-deterministic. And yet, very few tools exist to enable OCEs to perform this critical task.

Our goal is to build a tool that will help OCEs correctly and swiftly localize a bug to the buggy commit. Over a period of eight months, we studied post-deployment bugs, their symptoms, the buggy commits that caused them, and the current approach to bug localization that Orion's OCEs follow. We made four key observations.

1) *Bug localization is time-critical, bug fixing is not.* When a bug disrupts a service, the OCE's task is to keep the service disruption time to a minimum. She finds the buggy commit as fast as possible and reverts it rather than wait for the concerned developer to fix the bug. The reason is that, depending on the complexity of the bug, the developer may take a long time to fix it. Therefore to keep disruption to a minimum, it is better to revert the buggy commit first and introduce the fix at a later time. Thus, fast commit-level bug localization is critical.

2) *Rich monitoring infrastructure exists but is insufficient because of uncaptured dependencies.* Since service disruptions are a major concern, developers have created thousands of active *probes* that periodically monitor service-components or API calls and raise *alerts* if they fail. Despite this, bug localization is a challenge because a probe to a component may fail not because of any change to the component itself, but because of a change to a dependent component. For instance, a server-side probe failed with an exception `Type RecipientId not supported` because a developer made a commit to client-side code that added support for the datatype `RecipientId` without adding support on the server-side. To make matters worse, as the service evolves fast, new dependencies emerge at a rapid rate and no tool can completely capture all of them.

3) *Text-based similarity exists between symptom and cause.* We have found that when a probe detects a bug, a text-based similarity often exists between the unhealthy

probe name or exception text that it generates and the source-code change that caused the bug. In the example mentioned in the previous paragraph, the term `recipient` occurs in both symptom (the exception text) and cause (added support to the datatype on client). We also see this similarity in some customer complaints as well which predominantly use natural language. For instance, a customer recently complained that "*Email ID suggestions for people I know is not working.*". The cause for this was an incorrect change to a function named `PeopleSuggest`.

4) *Bugs may appear much after the buggy commit is made.* We observed that while the symptoms of a bug appear in a current build, the cause may be a commit deployed in a much older build. These are particularly challenging for the OCE to localize because they have to investigate, in the worst-case, all commits in the current build before moving on to investigate an older build. This can significantly lengthen service disruptions.

Given our goal and requirements, we design a novel search technique that we call *differential bug localization*. Using descriptions of the bug as a query, we detect changes to the abstract syntax tree in the source-code and search only these changes for text-based similarity. We call this *differential code analysis*. To find offending commits in older builds, we introduce a construct called the *build provenance graph* that captures dependencies between builds. We designed Orca, a custom code search-engine that leverages differential bug localization to provide a ranked list of "suspect" commits.

The Orion service has integrated Orca into its alerting and monitoring processes. This chapter describes Orca and makes the following contributions:

- We provide a study of post-deployment bugs found in the Orion service and their characteristics (Section 2.3).
- We introduce *differential bug localization*, which uses two constructs: differential code analysis of the abstract syntax tree, and the build provenance graph (Section 2.4).
- We have designed Orca, a tool that Orion's OCEs are actively using to localize bugs (Section 2.5).
- We provide an evaluation of Orca for bugs found in the Orion service (Section 2.6).

To the best of our knowledge, ours is the first such study of a bug localization tool deployed on a large-scale enterprise service. We have evaluated Orca on 33 post-deployment bugs found in Orion since October 2017. We show that Orca correctly localizes 26 out of 33 bugs for a recall of 79%. In 25 of the 26 cases, the correct commit was ranked in the top 5 records shown by our UI (Section 2.6). We also show that Orca causes a 4x reduction in the work done by the OCE.

We have designed Orca for usability and ease-of-adoption. While this chapter concentrates on Orion's deployment of the tool, Orca has been deployed on five other

services within our enterprise. Our techniques are generic and extend well to other large services. To make it easy for OCEs to use the tool interactively, we have optimized its performance through multiple caching and preprocessing techniques. Our user-interface provides results with an average run-time of 5.9 seconds per query.

2.2. Related Work

The Programming Languages, Software Engineering and Systems communities have extensively studied bug detection, bug localization and debugging. While Orca takes inspiration from some of this prior work, it targets a fundamentally different application space, i.e. large-scale service deployments. Also, Orca is meant to be used by on-call engineers, not developers.

A bug localization tool for such services needs to be *fast*: the query response time should be at most a few seconds since OCEs will use the UI interactively. It should be *general*: the techniques should support code in different languages and should not need an OCE or developer to provide specification. It should be *non-intrusive*: the service's existing coding and deployment practises should not require change. Finally, it should be *adaptive:* it should work in an extremely dynamic and changing environment. We now describe prior work in the area and why they do not satisfy some or all of these requirements.

IR-based Bug localization techniques (Ashok et al., 2009; Lam et al., 2015; Rao & Kak, 2011; Rath et al., 2018; S. Wang & Lo, 2016; Youm et al., 2015; Zhou et al., 2012) use a given bug-report to search, based on textual similarity, for similar bug reports in the past. For each match, they localize the bug at the *file-level*, not at commit-level, to files that have been changed to address similar bug-reports. (S. Wang & Lo, 2016) presents a structured and detailed study of the various techniques that are used for information-retrieval based bug localization. They use similar search techniques over five major concepts: *version history*, *bug reports*, *stack traces*, and *reporter information*.

While these techniques are fast, general and non-intrusive, they assume a stationary system, i.e. they assume that there is an inherent similarity between the current version of the system and previous versions. This is fundamentally not true in service deployments. For instance, Orion experiences a change in module dependencies at the rate of 1% to 3% every month, and some source-code files consistently change more than once a day!

Moreover, such studies are usually conducted for software products. Comparatively, the work presented in this chapter is different because it focuses on (a) dynamic deployments of software services, and (b) structured (bugs) and *unstructured* queries for localization of the manifested issue. Finally Orca is deployed and being used in real-time on a large service deployment. To our knowledge, existing bug localization techniques have not seen such deployment.

Dynamic Analysis techniques are the most commonly used and widely studied approaches for detecting bugs and issues in software. Testing and automated fuzz

testing (D. L. Bird & Munoz, 1983; Godefroid et al., 2008) techniques provide an effective method to automatically generate test cases that produce random inputs for the underlying software under test. Although these techniques are useful, they are complementary to our approach. Testing is never complete and does not provide guarantees about the correctness of the software under test. As a result, in spite of comprehensive testing, bugs still emerge regularly in service deployments, as we have noticed with Orion.

To find post-deployment bugs, statistical techniques (Bowring et al., 2002; Chilimbi et al., 2009; Liblit et al., 2005) were employed to automatically isolate bugs in programs by tracking and analyzing *successful* and *failed* runs of the program. While this technique holds promise, it is intrusive: it requires fine-grained instrumentation and a large number of program traces from the service deployment. Given the stringent performance needs and dynamism in services, we do not have the luxury of utilizing such techniques.

Delta Debugging techniques (Cleve & Zeller, 2005; Zeller, 2002) help automate the problem of debugging by providing the debugger or programmer information about the state of the program in passing and failing runs. The possible search space of the root cause is systematically pruned by using information from the various runs and by creating new executions. The ideas in delta debugging rely heavily on *program slicing* (Agrawal & Horgan, 1990; Weiser, 1981). Given its requirements, delta debugging tends to require a large number of tests and the data from instrumented programs. Hence, it is intrusive.

Static analysis entails analyzing software without executing the programs in question. The analysis may be performed at the level of the source-code itself, or at the level of the object code that is generated for execution (byte code or binary). Analysis is performed by automated tools that tend to be rooted in some formal method such as *model checking* (Clarke et al., 1999), *symbolic execution* (McMillan, 1993), and *abstract interpretation* (Cousot & Cousot, 1977). Although such techniques have shown significant promise in the past, performing such analysis on a large scale for services has proven to be very slow and intractable. Performing program analysis and verification at smaller scales for individual components is the current limit of such techniques.

Tools such as Semmle (De Moor et al., 2007) provides a unified framework that implements various program analysis techniques and correctness checks. In our experience, Semmle has proven to be somewhat useful for simple situations, but it lacks generality.

Differential static analysis techniques such as SymDiff (Lahiri et al., 2010) are immediately relevant to the problem discussed in this chapter. But differential analysis techniques are usually *property driven*; two versions of the program are analyzed with respect to a specific correctness property. For example, the analysis may be performed for asserting differences in the new version w.r.t. null pointer dereferences. We believe this approach too lacks generality since it is not feasible to enumerate all such

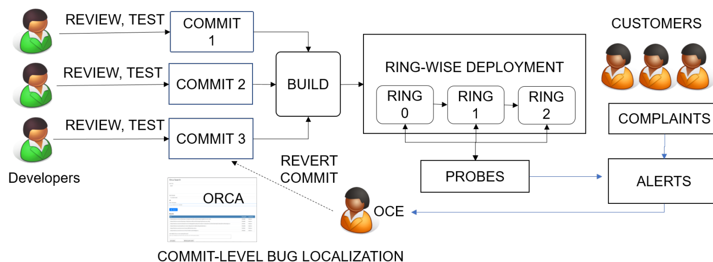


Figure 2.1: System Development Lifecycle of the Orion Service.

properties of code in large dynamic services with multiple dependent components. Yet, we do draw inspiration from this work to build Orca's AST-based differential analysis,

In addition to previous work (Yuan et al., 2012; Zhao et al., 2017) on bug localization and debugging, previous-work has targetted improving log-based debugging by making logs richer and more targetted towards diagnosability. We believe such work is complementary to our approach and Orca can gain significantly with such techniques that enhance logs.

2.3. Overview

In this Section, we first describe the system development lifecycle of the Orion enterprise email service and the role of the OCE. We next describe the characteristics of post-deployment bugs in Orion and motivate the approaches we adopt in Orca. We provide an overview of Orca and its goals, and finally state Orca's scope.

2.3.1. System Development Lifecycle

Figure 2.1 shows Orion's system development lifecycle (SDLC) and Table 2.1 holds a summary definition of each term we use for the reader's convenience. Multiple developers *commit* code, where a commit varies in complexity from a small tweak to a single file to changes to hundreds, even thousands of files. These commits are *reviewed* by one or more reviewers that the developer chooses. After multiple iterations with the reviewers, the commits are tested using unit, integration, and functional tests. Periodically, the administrator creates a new *build* by combining a set of commits. A build is a unit of deployment of the service and may contain just one, or hundreds of commits.

Builds are deployed in stages onto *rings*. A ring is a pre-determined set of machines on which the same build runs. The build is first deployed onto the smallest ring, or "Ring 0", consisting of a few thousand machines. When it is considered safe, the build progresses through multiple rings such as Ring 1 and Ring 2 until it finally is deployed world-wide. The idea of this staged deployment is to find bugs early in the life-cycle of deployment.

Term	Definition
Commit	Set of file changes made by one developer.
Review	Recommendations made by one or more developers for a commit.
Build	Unit of deployment for the service consisting of one or more commits.
Ring	Set of machines onto which a build is deployed.
Probe	Periodic checks to functions/APIs to ensure they are working as expected.
Alert	An email- or web-based notification that warns the OCE of a problem.

Table 2.1: Terms and their definitions

Once the build is deployed to a ring, several tools monitor it. A tool may use passive or active monitoring techniques, either analyzing logs or sending periodic *probes* to a component. It uses anomaly detection techniques to raise an *alert* that the OCE receives.

If only the machines running a specific build raise alerts, the OCE concludes that the build is buggy and she begins bug localization. Roughly half of all alerts in Orion are caused by bugs. Consequently, bugs are a significant reason for service disruption. If the alerts are not confined to a particular build, the bug is likely due to other reasons such as faulty networks or hardware misconfigurations. Root-causing infrastructure issues is not our focus as several tools already exist for this purpose (M. Yu et al., 2018; Q. Zhang et al., 2011).

To localize the bug to a commit, the OCE picks the commit that she feels is most likely to cause an issue and contacts the developer who created it. If the developer responds in the affirmative that her commit may indeed have caused the bug, the commit is immediately reverted and the service is restored to a healthy state. Note that the developer does not necessarily debug or fix the bug before responding to the OCE. If the developer says that their commit is not responsible for the bug, then the OCE picks the next most likely commit, and repeats the process until the service becomes healthy. Note that compared to a novice, an experienced OCE with domain-knowledge may pick the correct commits more promptly and therefore restore the service much faster. Orca removes this dependency on experience and domain-knowledge by codifying it in its search algorithms.

2.3.2. Post-Deployment Bugs

Over a period of eight months, we analyzed various post-deployment bugs and the buggy source-code that caused them. Table 2.2 outlines a few characteristic issues. The table shows the type of alert, it provides an overview of the symptom, and a description of the root-cause. It also shows the number of commits (and the number

of files) that an OCE has to consider while performing bug attribution which, in some cases is more than 200.

No.	Type	Symptom	Cause	Term Similarity	Commits
1	Customer complaint	"People Suggestion" feature, that suggests potential recipients for an email, was not working for a subset of users.	A "people ranking" algorithm was incorrectly modified.	A variable used the keyword `suggest` in the modified function.	33
2	Probe	An email synchronization problem was detected.	A buggy commit to the synchronization component caused requests coming only from web-based clients to fail.	The probe contained the name of the synchronization component, which was also in the directory path of the modified files.	201
3	Probe	A worker process for a specific component started crashing repeatedly.	Incorrect configuration changes to the component's environment caused this. The commit was to a previous build but bug showed later only after a large number of users hit it.	The component name matched in the class name of modified code.	201
4	Customer complaint	Authentication process for some applications that used REST started crashing.	A library that these applications depended upon was modified but was not tested for all applications.	Keyword `auth` was in the pathname of the change.	46

5	Probe	Threads were getting blocked in processing, large delays were noticed in REST calls made by a web service.	Http client code for the service had been modified to make some synchronous calls asynchronous.	The component name matched a modified user-agent string in the http client.	18
6	Probe	No. of exceptions generated anomalously high while migrating mailboxes	Caused by a code-change to a mailbox migration components.	Keyword `migrat` matched in a function name in added code.	12
7	Probe	Number of exceptions in the log file for a component **C** became abnormally high.	Support for a new datatype was added in a component that made an API call to component **C**, but **C** does not support that datatype.	The exception text for **C** contained the datatype.	70
8	Probe	CPU Usage on a set of machines was anomalously high.	Reads and writes to a dictionary were not thread-safe. Multiple threads were reading from a dictionary while it was being modified, causing a CPU blowout.	No keyword matched.	89
9	Probe	POP and IMAP services started failing.	Dependencies were broken when a code commit changed a library that the POP and IMAP services used.	Keywords `Pop` and `Imap` matched in code changes.	110
10	Customer complaint	A client signing in via OAuth does not display calendar.	Client-side implementation was incompatible with the server-side commit.	Keyword `OAuth` matched the symptom and the server-side change.	39

Table 2.2: Examples of post-deployment bugs

In general, we have found that bugs fall predominantly into one of the following

categories:

- **Bugs specific to certain environments.** Often, a component starts failing because files implementing that component have a bug (Bugs 2, 5). Usually, the failures happen only for a specific type of client such as web-based clients, or in a specific region such as Japan. Tests do not catch the bug since not all configurations, clients and environments are tested.

- **Bugs due to uncaptured dependencies.** Dependencies can be of various types. In Bug 10, a server-side implementation is modified without appropriately modifying the client-side code. This happens because developers often overlook such dependencies as no compile-time tool captures them completely. Another example of an uncaptured dependency is Bug 4. A commit modifies a certain library, but unbeknownst to the developer, another component depends on certain features in the older version of the library and stops working correctly.

- **Bugs that introduce performance overheads.** Several probes track performance issues. For instance, in Bug 8, a code addition that was not thread-safe caused CPU overload that slowed down the service. The bug emerges only when a large number of users use the service. Hence it is not caught in testing.

- **Bugs in the user-interface.** A UI Feature starts misbehaving, so a customer complains. An example of this is Bug 1.

2.3.3. ORCA OVERVIEW

Studying these bugs and observing the OCEs gave us valuable insights. We state these insights, and describe how Orca 's design is influenced by them.

Often, the same meaningful terms occur both in the symptom and the cause. Table 2.2 captures this under the “Term Similarity" column. Some matched terms are proper nouns such as the component name (Bug 3) or global datatypes (Bug 7). They can also be commonly used terms such as protocols (`Imap` in Bug 9) or the function performed (`suggest` in Bug 1 and `migrat` in Bug 6). Given the term similarity between symptom and cause, we designed Orca as a custom-designed search engine, using the symptom as the query-text, and giving us a ranked list of commits as results. Orca searches for the symptomatic terms in names of the modified code by performing differential code analysis on the abstract syntax tree. We describe this procedure in Section 2.4.1.

Testing and anomaly detection algorithms do not always find a bug immediately. A bug may start surfacing in a new build despite being introduced through a commit to a much older build. Bug 3 in Table 2.2 is an example. We introduce a *build provenance graph* to allow Orca to expand its search to older builds from which the current build has been derived. We describe this in Section 2.4.2.

Builds may have hundreds of commits, so manually attributing bugs can be long-drawn task. For instance, Bug 2 appears in a build that had 201 commits. Bug 3 appears in a build with 160 commits but the root-cause was in the previous build which had 41 commits. The OCE is faced with the uphill task of analyzing, in these cases, up to 200 commits before discovering the buggy commit. OCEs often work at odd hours and are constantly pressed for time. We therefore used various techniques to make Orca fast, interactive and accurate. We have built an Orca UI and leverage caching and parallelism to ensure an interactive experience for the OCE. We describe our optimizations in Section 2.5.

There are thousands of probes in the system, and probe failures and exceptions are continuously logged. Therefore there is rich data on what symptoms, or potential queries to Orca look like. This allows us to track frequency of terms that appear in the queries and use the *Inverse Query Frequency (IQF)* rather than the Inverse Document Frequency (IDF) in our search rankings. We explain this further in in Section 2.4.3.

2.3.4. Orca Scope

In this section, we elaborate what Orca does *not* aim to solve. This is primarily because existing techniques already address these issues.

Orca does not target issues caused by faults in the infrastructure. Several techniques (Isard, 2007; M. Yu et al., 2018) exist to do this.

Orca does not solve the anomaly detection problem directly as several techniques already exist for this (Attariyan et al., 2012; Chow et al., 2014). But we do recognize that anomaly detection algorithms are imperfect. Orca's build provenance graph helps find bugs even when the anomaly detection algorithm detects a bug well after a commit introduces the bug.

Orca does not handle bugs where the query does not have any context-specific information. Notable examples are performance issues, where the symptoms are, simply, out-of-memory exceptions or CPU-above-threshold exceptions. There exist other techniques (Chow et al., 2014; Zhao et al., 2014) in the literature which can debug such issues and therefore, can be used in combination with Orca.

2.4. Design

In this Section, we describe Orca's differential bug localization constructs in more detail. The input query to Orca is a symptom of the bug. This could be the name of the anomalous probe, an exception message, a stack-trace, or the words of a customer-complaint. The idea is to search for this query through *changes* in code or configurations that could have caused this bug. Thus the "documents" that the tool searches are properties that changed with a commit, such as names of files that were added, removed or modified, commit and review comments, modified code and modified configuration parameters. Orca's output is a ranked list of commits, with the most likely buggy commit displayed first.

We first describe two novel constructs that we use for search-space pruning and

search-space expansion respectively: differential code analysis, and the build provenance graph. Finally we provide a detailed description of the search algorithm.

2.4.1. Differential Code Analysis

Orca can search the entire code file to perform bug localization. But this approach can find false matches and drop Orca's precision, especially since we have found that every commit changes, on average, only about 20 lines of source-code per-file (including file additions), whereas the entire file can consist of hundreds of lines of code.

We therefore employ differential code analysis within Orca. Differential Analysis has been studied in the past, albeit mostly at the semantic level (Lahiri et al., 2013). The goal in prior research has been to identify relevant pieces of the code change that can potentially cause different behavior in the new version relative to the old version, but with reference to a specified *property*, such as null dereference, memory consistency, etc. Such techniques rely on compuationally expensive techniques such as differential symbolic execution (Person et al., 2008) and regression verification (Godlin & Strichman, 2009).

Our approach leans more toward the syntactic than the semantic side. There are two reasons for this: (a) it is difficult to determine the full set of properties to capture all bugs in large-scale services, and (b) we would like to avoid the performance overhead of traditional static analyzers for differential analysis. We use the abstract syntax trees (AST) of the old and new version of the program to discover relevant parts of the source that have been *added*, *removed*, and *modified*.

Our analysis finds differences in entities of type class, method, reference, condition, and loop. We formalize the notion of *diffing* two ASTs, A_i and A_j, as creating a "difference set" D in the following way. Say e_i is the old version of an entity, and e_j the new version. Say t is the type of the entity. Then,

$$D = \begin{cases} d_{added} = \forall e \in A_j \mid type(e) = t \wedge e \notin A_i \\ d_{removed} = \forall e \in A_i \mid type(e) = t \wedge e \notin A_j \\ d_{changed} = \forall e_j \in A_j \mid type(e_j) = t \wedge e_j \in A_i \\ \qquad\qquad\qquad \wedge e_j \neq e_i \wedge type(e_i) = t \\ d_{diff} = e_i \Delta e_j \mid e_j \in A_j \wedge type(e_j) = t \wedge \\ \qquad\qquad\qquad e_i \in A_i \wedge e_j \neq e_i \end{cases}$$

Thus, the difference set D captures entities that have been *added*, *removed*, or *changed*. For all entities in D, we also capture the textual differences between the two versions of the entity, d_{diff}, using the following heuristic. We check if the old version of the entity e_i is also found in the new AST A_j. Our heuristic checks for a "similarly" named entity or declaration as e_i. If we find such an entity, we treat it as a newer version of the entity e_i, i.e. e_j. We run differential code analysis for all changed files in a commit. To find term similarity at the file-level, Orca searches

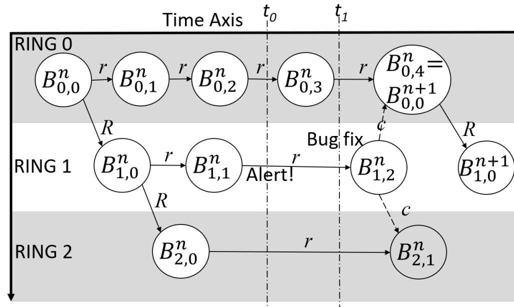


Figure 2.2: Example fragment of a build provenance graph.

through the difference set of each file.

We would like to point out that our syntactic approach to differential code analysis is not sound: we may detect changes even though there are none. For instance, consider the case where a function name changes completely, but the body remains unchanged. Our algorithm will treat this as a completely new function. While this could cause a precision drop in Orca since we are including more textual differences than actually exist, the algorithm does ensure that all changes are captured.

2.4.2. Build Provenance Graph

In Section 2.3.2, we have shown that a buggy commit to an older build may show symptoms only in a subsequent build. This could be because an inaccurate anomaly detection algorithm detects an anomaly too late. It may also be that a subtle bug manifests only in certain environments or only when a large number of users hit the service.

To accommodate this scenario, we expand our search to include previous builds that the symptomatic build is “derived" from. We maintain a *build provenance graph (BPG)* that captures dependencies between various builds along the axes of time and ring ID. Figure 2.2 shows an example fragment of a build provenance graph.

Construction

We now describe how we construct the build provenance graph. The BPG captures how builds are created and promoted in the different rings. Every build is represented as $B^i_{r,v}$, where i is a the build identifier, r is the ring identifier, and v is the version of the build within a ring.

Say a build spends a certain amount of time in Ring 0. If it is stable and shows no unhealthy behavior for a while, it is considered for promotion to the next ring. Build $B^n_{0,0}$ is one such build. It forks off build $B^n_{1,0}$ which runs in the next ring, Ring 1. Since $B^n_{1,0}$ is directly derived from $B^n_{0,0}$, any bugs that emerge in $B^n_{1,0}$ could potentially be due to commits originally made to $B^n_{0,0}$. Thus, we introduce an *inter-ring* edge between the two builds. Similarly, once a build is considered stable in Ring

1, it is forked off to Ring 2. This introduces the inter-ring edge between $B^n_{1,0}$ and $B^n_{2,0}$. We specify an inter-ring edge as:

$$B^i_{r,v} \xrightarrow{\text{R}} B^i_{r+1,0},$$

For a given build identifier i, only one inter-ring edge can exist between two consecutive rings.

Meanwhile, developers make fresh commits $within$ each ring too, thereby creating the next build versions within the same ring. So in the figure, $B^n_{0,1}$ is derived from $B^n_{0,0}$, $B^n_{1,1}$ from $B^n_{1,0}$, and so on. Several such intra-ring edges may exist in all rings though most are in Ring 0 which is the most dynamic and experimental ring.

We represent the intra-ring edges as:

$$B^i_{r,v} \xrightarrow{\text{r}} B^i_{r,v+1}$$

At any given time, a vertical line drawn through the graph yields the *active build* within each ring at that time. For instance, at time t_1, the active builds are $B^n_{0,3}$, $B^n_{1,1}$, and $B^n_{2,0}$. For ease of explanation, we assume that at any given time there is only one active build in every ring though in reality there could be many.

We now explain a third edge-type called the *back-port* edge, shown in Figure 2.2 by dotted arrows. A critical bug that goes undetected may, with time, propagate to a large number of builds across all rings. Say at time t_0, such a bug causes an alert in build $B^n_{1,1}$. Say at time t_1, using the process described in Section 2.4.2, we localize the bug to a commit made to the earlier build $B^n_{0,0}$. The bug is fixed through a new commit c to the code of $B^n_{1,1}$, and this generates $B^n_{1,2}$, with an intra-ring edge between them.

We can see that since the bug originated in $B^n_{0,0}$, it also exists in the active builds within Ring 0 and Ring 2 that have been derived from it, i.e. $B^n_{0,3}$ and $B^n_{2,0}$. Consequently we apply the commit c, or we *back-port* it, to $B^n_{0,3}$ and $B^n_{2,0}$. This creates new builds $B^n_{0,4}$ in Ring 0 and $B^n_{2,1}$ in Ring 2. Thus we add two back-port edges with the label c from $B^n_{1,2}$ to $B^n_{0,4}$, and $B^n_{1,2}$ to $B^n_{2,1}$.

We now state this mathematically. Let c_j be a commit to $B^i_{r,v}$, and let $BP(c_j)$ be the set of all builds that c_j is back-ported to. Then,

$$B^i_{r,v} \xrightarrow{c_j} B', B' \in BP(c_j)$$

Finally, we describe how the identifier i changes in the build provenance graph. All builds across all rings that have the identifier i are derived originally from $B^i_{0,0}$. Thus $B^i_{0,0}$ is called an *origin* build. With time, several new commits are applied in Ring 0. To ensure that these commits are fully deployed across all rings, a subsequent build from Ring 0 is chosen to be the next origin build. In the figure, this is $B^n_{0,4}$, which we rename as $B^{n+1}_{0,0}$, or the *next* origin build. All subsequent builds are now derived from this new origin build.

It can be seen that, barring backport edges, every node in the build provenance graph has only one incoming edge. This can be either an inter-ring or an intra-ring edge.

Traversal

We now describe how Orca uses the build provenance graph to expand its search-space and find potential buggy commits in older builds. Given a symptomatic build $B^i_{p,q}$, the purpose of the traversal is to find a list of candidate commits for the search.

We observe that Ring 0 is the most experimental of all rings. Builds in Ring 0 see a large number of significant commits. Consequently, our intuition is that, to localize a bug that appears in $B^i_{p,q}$, we should search all builds back to the the origin build $B^i_{0,0}$, which is in Ring 0. Thus using inter-ring and intra-ring edges, we back-track from $B^i_{p,q}$ to the origin build $B^i_{0,0}$. In addition, we also include all back-ported commits to every build on the same path. Since every build has only one incoming inter-ring or intra-ring edge, there is only one such path from $B^i_{p,q}$ to $B^i_{0,0}$. The candidate list of commits to search will include all commits made to the builds on this path, and the back-ported commits on the same path.

We now specify this mathematically. Let the candidate list of commits be Γ. Let P be all the builds in the path from $B^i_{0,0}$ to $B^i_{p,q}$. Then,

$$\Gamma = \bigcup C(B^i_{r,v}) \bigcup c_j$$

where $C(B^i_{r,v})$ is the set of all commits made to $B^i_{r,v}$, and $\exists B$ such that $B \xrightarrow{c_j} B^i_{r,v}$, $B^i_{r,v} \in P$.

We now explain this through an example with Figure 2.2. Say an alert is raised in $B^n_{2,1}$. Backtracking from $B^n_{2,1}$ to $B^n_{0,0}$ yields the set of commits that are of interest to us $\{C(B^n_{2,1}), C(B^n_{2,0}), C(B^n_{1,0}), C(B^n_{0,0}\})$. To this, we add c, which is a backported commit from $B^n_{1,2}$ to $B^n_{2,1}$, thereby giving us the final list of commits to search in. That is,

$$\Gamma = \{C(B^n_{2,1}), C(B^n_{2,0}), C(B^n_{1,0}), C(B^n_{0,0}), c\}$$

2.4.3. Algorithm

In this Section, we describe Orca's search algorithm which uses differential code analysis and the build provenance graph. The Orca search algorithm consists of four steps: 1) Query pre-processing where we perform tokenization, stemming and stop-word removal, 2) build graph traversal described in Section 2.4.2, 3) token-matching in code changes using Differential Code Analysis and 4) ranking and visualization of results. Our system runs differential code analysis and constructs the build provenance graph in the background periodically so that these tasks do not slow down the query response time.

Step 1: The search queries are symptoms of the problem, consisting of probe names, exception texts, log messages, etc. We first tokenize the terms in these symptoms by using a custom-built code tokenizer. This tokenizer uses heuristics that we

have built specifically for code and log messages, such as splitting large complex strings along Camel-cased or Pascal-cased fragments. We also create n-gram based tokens since we have found that bi-grams, like `ImapTransfer`, `mailboxSync`, capture important information.

Next, we filter out irrelevant words also called stop-words (Luhn, 1959) from these symptoms. Previous work has shown that logs have a lot of inherent structure (Xu et al., 2009). For instance, all exception names have the suffix `Exception` and almost all log messages have a timestamp. Unlike conventional search-engines, even before we built Orca, we had access to about 8 million alerts consisting of probe names, exceptions and log messages from Orion's log store. We therefore perform stop-word removal on these to weed out commonly used or irrelevant terms such as `Exception` or timestamps. This step gives us a list of relevant "tokens" in the symptom. For each token t, we also maintain an *Inverse Query Frequency (IQF)* value (Yang et al., 2008) that we call t_{IQF}, obtained by analyzing Orion 's logs. t_{IQF} is calculated as (No. of queries/No. of queries in which token t appears). A high value of t_{IQF} implies that the token t is more important.

Step 2: We traverse the build provenance graph to find all builds related to the symptomatic build. From each build we discover, we enumerate all the commits that created the build. This leads us to the next step, which is matching tokens to files for each commit.

Step 3: Within a given commit C, for each file f and token t in the symptom, i.e for each tuple $T =< f, t >$, we search for the token in difference set of the file, D_f. We use $TF-IQF$ (Yang et al., 2008) as a "relevance" score, R_T^C, for each tuple. R_T^C is calculated as $n * t_{IQF}$ where n is the number of times the token t appears in difference set. This relevance score captures that the tuple $< f, t >$ is more relevant if the token t is very infrequent (i.e. t_{IQF} is very high), or if it appears many times in f.

We repeat this step for every token and file in the commit. At this point, we have file-level relevance values. Note though that we perform commit-level bug localization. Thus, we now aggregate the relevance values across all files and tokens to get one relevance value for the commit C, that we call R^C. So,

$$R^C = \underset{T}{\Theta} R_T^C \tag{2.1}$$

where Θ is an aggregation function such as such as `Max`, `Avg` or `Sum`. We show in Section 2.6 that the MAX function provides the best results in our deployment.

Step 4: Finally, Orca returns a list of commits in decreasing-order of their relevance. Along with the commit ID, we also display one representative file-token pair which has the highest value of R_T^C. According to the OCEs we interviewed, showing the token with the highest relevance value helps them interpret the results better and increases their confidence in Orca's results. This is the reason we do not use standard aggregation techniques such as cosine-similarity (Manning et al., 2008) to aggregate all token-matches.

EXAMPLE

We shall use the following example to illustrate how the search algorithm works. Say commits C_1 and C_2 create a build B. Say a probe called `LdapAuthProbe`, that monitors the LDAP authentication service, starts throwing exceptions of type `AuthFailedException`. Let us also say that the bug was caused by commit C_2 that erroneously modified a function named `LdapRequestHandler` in a class named `LdapService`, declared and defined in a file `LdapService.cs`. Say C_1 modified an `Imap` protocol implementation in a file `Imap.cs`.

The query to Orca is "`LdapAuthProbe AuthFailedException`. First, we tokenize and stem the query, and remove stop-words `Failed` and `Exception`. This yields the tokens `Ldap`, `Auth` and `Probe`. The word `Probe` occurs very frequently across all symptoms and therefore receives a very low IQF score, whereas `Ldap`, being a specific protocol, gets the highest IDF score. `Auth`, being somewhat more frequent than `Ldap`, receives a slightly lower IQF score.

We leave out build graph traversal for the sake of simplicity. Therefore, our list of candidate commits are only C_1 and C_2. In our example, the token `Ldap` will match both the class name, `LdapService`, and the function, `LdapRequestHandler`. Therefore the value of relevance for this is $2*$ the IDF value of token `Ldap`. C_2 will also find a match with token `Auth`. C_1, however, does not match any of these tokens. Our ranking algorithm will therefore choose C_2 over C_1 and, as the highest-ranked result, it will show the filename `LdapService.cs` and token `Ldap` rather than `Auth`.

2.5. IMPLEMENTATION

We have implemented Orca in a combination of C# (using .NET Framework v4.5) and SQL. Currently, the implementation is approximately 50,000 lines of code. In this Section, we briefly describe our implementation of Orca, and the various user interfaces we expose for OCEs.

2.5.1. DATA LOADERS

Since Orca requires information about various different parts of the system – source-code, builds, deployment information and alerts – a significant part of our implementation are data loaders for these different types of data. Figure 2.3 shows an architectural overview of the implementation. At the heart of Orca is a standard SQL database. This database is populated by data loaders at a predefined frequency. We now describe the different data loaders we use.

Source Data We implemented loaders for various source-control systems such as GIT and others internal to our organization. These loaders ingest source-code, code-versions and histories. The differential code analysis algorithm uses data from this loader.

Builds Data about builds resides in multiple big-data logs. We have build loaders

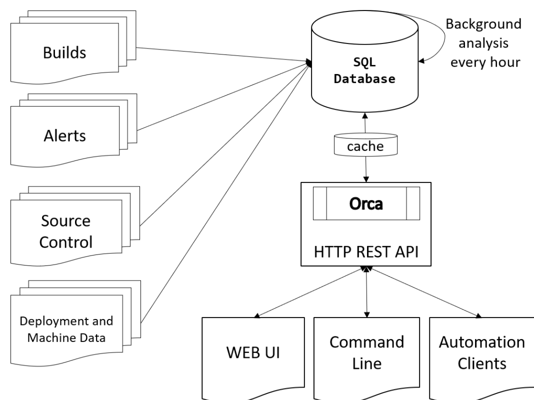


Figure 2.3: Implementation of Orca.

that interface with several big-data logging systems to load build-specific information into our SQL database. We use this loader to construct the build provenance graph.

Deployment and Machine Data We load logs created by continuously monitoring the state of all machines within all rings. The state includes information about the current status of the machine (healthy/unhealthy), along with the information on the build version running on the machine.

Alerts are loaded from existing databases. Today, Orca supports multiple data sources for loading alert information.

2.5.2. Background Analyses

As our data loaders periodically load new data into the SQL database, we periodically initiate differential code analysis, build provenance graph construction and IQF calculation used in Section 2.4.3. If needed, the frequency of an analysis can be changed to make it more/less frequent. For example, the IQF calculation runs once a week as opposed to the other two processes which are run once every hour. Finally, it should be noted that all analyses that have been developed and deployed within this system are agnostic of the data source. The SQL database schema is normalized; thus, providing the same interface to all analyzers irrespective of the data source.

2.5.3. API Implementation

We now describe the Orca API and its implementation. Each Orca request is processed in real-time by the core Orca engine, and results are returned in JSON format. Clients decide on the relevant parts of the return result and how to display them. We make use of a Redis Cache (Carlson, 2013) for improving our lookup times associ-

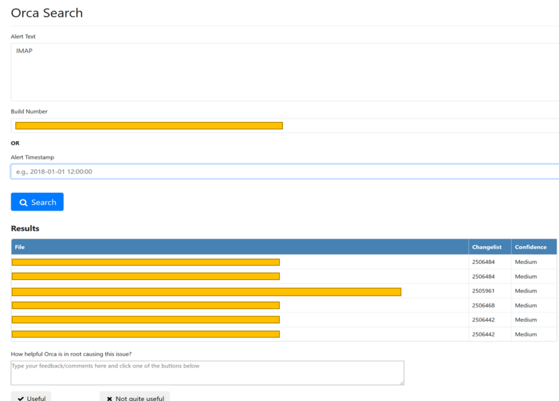


Figure 2.4: Web based UI of ORCA for Orion.

ated with data that is static. This includes data about the builds, files, source-code, difference sets, and the build provenance graph. Internally, the Orca system records all requests made and any feedback provided, which we use for learning and improvement.

All our services, servers, databases, and caches are implemented and operated using Azure as both a Platform as a Service (PaaS) and Infrastructure as a Service (IaaS) provider.

2.5.4. Usage

Orca can be used in two ways: OCEs can use it to make *ad-hoc queries* interactively, or an alerting infrastructure can query Orca to get a list of suspect commits for an alert and include it in the alert itself.

For the OCEs, we have built a web-based UI that allows them to enter the details of their query and view the results. Figure 2.4 shows a screenshot of the UI with sensitive information removed. We have also built a PowerShell® *cmdlet* with which the OCE can interact through a command-line interface.

To integrate Orca with alerting infrastructure, we have built an API that the alerting system can query directly. Currently, this mechanism is being used by multiple groups that are generating alerts within the Orion group. The web-based UI and *cmdlet* tend to be used by OCEs when new information such as log text or exception text has been discovered after the original alerts were generated.

Today, Orca has been deployed on multiple code-bases for six large-scale services within our enterprise. The combination of data loaders used in each code-base is slightly different and unique, but fully operational and functional.

2.6. Evaluation

Since its deployment with Orion in October 2017, the Orca API has been used to debug 558 issues within the Orion service. Unfortunately, there is no central location where OCEs retrospectively log information about buggy commits. Thus to quantitatively evaluate Orca's accuracy, we interviewed multiple OCEs and manually analyzed source-code, bug-reports and email-threads. We collated detailed information for 33 of these bugs. In this Section, we provide results with this set of 33 bugs. These bugs vary greatly in characteristics. While some were inadvertently introduced by a single line-change, others were caused by complex dependencies between components.

First, we evaluate Orca's result quality, i.e. how often it attributes a bug correctly to the right commit. Next, we evaluate how much effort an OCE saves by using Orca. Third, we evaluate the performance of Orca and the savings we get by using a Redis-based cache.

	No. of results = 5				**No. of results = 10**				**No. of results = 20**			
	DCA		DCA+BPG		DCA		DCA+BPG		DCA		DCA+BPG	
Agg.fn	R	MRR	R	MRR	R	MRR	R	MRR	R	MRR	R	MRR
MAX	0.70	0.44	0.76	0.46	**0.70**	**0.44**	**0.79**	**0.44**	0.70	0.46	0.79	0.45
SUM	0.55	0.43	0.61	0.46	0.70	0.45	0.73	0.48	0.73	0.45	0.79	0.48
AVG	0.49	0.33	0.36	0.27	0.61	0.34	0.61	0.34	0.64	0.34	0.67	0.31

Table 2.3: We evaluate differential code analysis alone (DCA) and with build provenance graph (DCA+BPG). Numbers in parentheses are the number of bugs correctly localized. We evaluate aggregating by MAX, SUM, and AVG.

2.6.1. Result Quality

To measure the quality of Orca's results, we interviewed several OCEs about how they would quantify result quality. Based on these interviews we determined that it is important that we find the buggy commit in as many cases as possible. This is captured by the *Recall (R)*, i.e. the fraction of bugs where we found the buggy commit in *any* position in our results. The OCEs also told us that they linearly scan the list of commits that Orca provides, hence the closer the correct commit is to the top, the more time they save. To capture this, we use the *Mean Reciprocal Rank (MRR)* (Manning et al., 2008). MRR is the most suitable metric since we assume there is only one buggy commit that causes the symptom. MRR is calculated as $\frac{1}{n}\Sigma_{i=1}^{n} 1/r_i$, where n is the number of queries, r_i is the rank of the buggy commit for query i. If Orca is unable to find the correct commit, we assume r_i is infinity, i.e. we add 0 to the sum total.

Table 2.3 shows the results for various combinations of parameters and features. We have run Orca with differential code analysis, both with and without the build provenance graph. We have varied the number of results we show in the Orca UI, and

we have evaluated Orca for different aggregation functions (Θ in Equation 2.1). We present results with differential code analysis only under "DCA" and the combination of differential code analysis and the build provenance graph as "DCA+BPG".

To evaluate Orca, we ask three questions:

- **How much value does the build provenance graph add?** We wanted to understand in what number of cases the build graph was actually necessary.
- **How should we aggregate the file-level relevances to commit-level?** Equation 2.1 in Section 2.4.3 described how we need to aggregate file-level relevance value into one value at the commit-level. We evaluate three different aggregation functions: MAX, SUM and AVG.
- **How many results should we show in the Orca UI?** When asked, the OCEs mentioned that they would not want to see beyond 10 results for each query. Hence, we wanted to evaluate what was the trade off between the number of results shown and the recall and MRR.

We now answer each of these questions in order.

The build provenance graph adds 13% to the recall. Observe the data in bold in Table 2.3. DCA alone localizes 23 bugs for a recall of 0.7. Adding the build provenance graph helps us localize 3 more bugs correctly thereby increasing our recall to 0.79. While at first glance, this may appear to be a small increase, OCEs find it significantly more difficult to attribute bugs that occur in older builds, and therefore the value of finding these 3 bugs eases the OCE's workload considerably. We show this quantitatively in Section 2.6.2.

The MAX and SUM aggregation functions perform better than AVG. While the buggy code-changes match some very high-relevance tokens, several lower relevance tokens match these code-changes too. Hence taking the average value across all matches dilutes the high-value token matches, therefore reducing both recall and MRR. Such a dilution does not happen if we use MAX or SUM. We choose MAX in our implementation.

Showing 10 results seems a good trade-off between result quality and UI succinctness. We evaluated results while setting the number of results shown as 5, 10 and 20. We find that with 10 results we achieve close to our best recall and MRR values. While showing 20 results gives us slightly better MRR with MAX as the aggregation function, we decided to keep our results concise and limit the number shown to 10.

With 10 results and using MAX, we obtained a recall of 0.79, i.e. we found the root-cause in 26 out of 33 cases. The MRR was 0.44. We also studied the matched terms for these 26 bugs and found that term-similarity serves as a good proxy to capture different types of bugs. Table 2.2 showed that matched terms fall roughly into four categories: they either match a component name, a function that the component performs such as `migrat` or `suggest`, datatypes, or protocol names such as `Imap`. We found in our case that of the 26 correctly localized cases, in 13 cases the token was

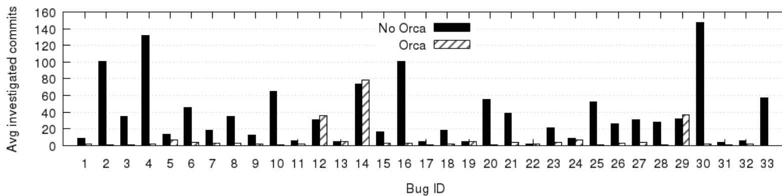


Figure 2.5: For all 33 bugs, the expected number of commits that the OCE investigates without Orca and with Orca.

a component name, in 9 cases it captured the function being performed, in 2 cases it matched a datatype, and in the remaining 2 cases, the match was on a protocol name. Therefore term-similarity is quite versatile: it helps us catch a variety of bugs.

Of the 7 cases that we could not localize, in 1 case the issue was related to performance. In 2 cases, the problem was because a configuration setting changed that triggered the use of code that was committed much earlier than our build provenance graph covered. In future work, we therefore plan to include all configuration settings in our differential analysis. In the remaining 4 cases, term similarity just did not capture the high-level semantics of the bug, and static or dynamic analysis may be required.

2.6.2. Reduction of OCE Workload

We now investigate the effect of Orca on reducing the OCE's workload. Given a bug, an OCE will investigate an average of $c/2$ commits to localize it, where c is the number of commits in the buggy build. If the OCE uses Orca, they need investigate at most r commits, where r is the rank of the correct commit in the results that the UI shows. If Orca does not find the correct commit, then apart from the 10 commits that Orca shows, the OCE needs to investigate an additional $(c-10)/2$ commits in expectation.

Figure 2.5 shows the number of commits investigated for all 33 bugs that we evaluated. Over all 33 bugs, without Orca, the OCE investigates an average of 36 commits. With Orca, she investigates an average of only 8.9 commits. Therefore, using Orca causes a 4x, or a 75% reduction in the OCE's workload. For the 26 bugs Orca localizes, this reduction factor is much higher, i.e. 15.4x. For the 3 bugs that were caught only because of the build provenance graph, the OCE had to investigate an average of 111 commits without Orca, and only 2.7 commits with it. This is a 41x improvement. These numbers point out the benefits that the build provenance graph gives us, and the benefits overall of using Orca for localization.

2.6.3. Performance

Finally, we evaluate the performance of Orca. We run Orca on a 32 core, 2GHz Intel Xeon E7-4820 CPU with 64 GB memory. We ran all 33 queries in sequence to obtain

these results. First, we evaluated the effect of the Redis cache on Orca's average query response time. Using Redis, the average response time reduced from 12.4 seconds to 5.97 seconds, a gain of 51.8%. Next, we varied the degree of parallelism in Orca from 32 to 128 using the C# MAXDOP parameter (Russinovich et al., 2012) and noticed a significant effect on average query response time. With parallelism set to 32, the average response time is 30.94 seconds, whereas with parallelism of 128, it is 5.97 seconds. Our evaluation shows that there is a significant potential for parallelizing Orca further, thereby catering to many more queries and providing lower response times. Therefore we can effectively scale Orca out to more services within our enterprise without loss in performance.

2.7. Conclusion and Future Work

In this chapter, we described Orca and the differential bug localization algorithm. Orca uses differential code analysis and the build provenance graph to find buggy commits in large-scale services. Orca is deployed with a large email and collaboration platform. We have shown that Orca finds the correct buggy commits in about 79% of bugs that we studied. We have also shown that Orca is efficient, accurate and easy to deploy.

In the future, we intend to work on techniques that will further improve Orca's recall. Dealing with performance bugs and configuration-related issues is our immediate next-step. Apart from that, we also intend to try more semantic-level program analysis techniques within Orca. Finally, we intend to scale out Orca to many other large-scale services.

3

Nudge: Accelerating Overdue Pull Requests Towards Completion

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 towards 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. Lastly, 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. Lastly, 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.

3.1. 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 (Asthana et al., 2019; Y. Yu et al., 2016), prioritization (Van Der Veen et al., 2015), and duplication (Q. Wang et al., 2019). Additionally, several bots and extensions have been built for platforms like GitHub and Azure DevOps to automate various software development workflows (Dey et al., 2020; Kumar et al., 2019b).

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, 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, the review process could be slowed down significantly. Hence, if the pull request's author and reviewers do not actively engage, 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. (Y. Yu et al., 2015) 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 ("Long-Running Branches Considered Harmful", 2020). Some of the most common side effects caused by long-lived feature branches or pull requests are:

- 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. (Dias et al., 2020b) 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.

In order to address these concerns, we designed and deployed Nudge, a service for accelerating overdue pull requests towards 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. (Y. Yu et al., 2015), 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 (Layman et al., 2008) 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 (Ostrand et al., 2004), reviewer information (MacLeod et al., 2018), and ownership information (Layman et al., 2008) to build our pull request lifetime prediction model.

While effort estimation models have been shown to be accurate (Attarzadeh et al., 2012), 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 nine 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 work 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 3.4–3.6).

2. We design and deploy a scalable implementation of our approach in a tool called Nudge (Section 3.7).

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 3.8).

4. We apply Nudge to 8,000 systems and demonstrate that its benefits remain present at scale (Section 3.8).

This work is a substantially revised extension of our earlier publication (C. Maddila et al., 2019). New in the present chapter 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.

3.2. 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 (Bettenburg et al., 2015; B. W. Boehm, 1984; Briand et al., 2000; Chulani et al., 1999; Menzies et al., 2013). 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 (B. W. Boehm, 1984), which he later updated to COCOMO 2.0 in 1995 (B. Boehm et al., 1995). This work was followed up by Briand et al. (Briand et al., 1999) who compared various effort estimation modeling 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. (Menzies et al., 2013) and Bettenburg et al. (Bettenburg et al., 2015) 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. (Soares et al., 2015), and Tsay et al. (Tsay et al., 2014) looked at a variety of factors to see which one had an impact on pull request acceptance. More specifically, Terrell et al. (Terrell et al., 2017) and Rastogi et al. (Rastogi et al., 2018) looked at gender or geographical location impact on a pull request acceptance. The work closest to our work is by Yu et al. (Y. Yu et al., 2015), 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 and provide actionable feedback to the developers leveraging that knowledge.

As shown by various studies (Bettenburg et al., 2015; B. W. Boehm, 1984; Briand et al., 2000; Chulani et al., 1999; Menzies et al., 2013), 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 ("GitHub Marketplace", 2020) of bots and automated services. It has also spawned active research (Storey et al., 2020) on understanding and building bots to assist with various software engineering tasks. Storey et al. (Storey et al., 2020) 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. (Lebeuf et al., 2019) 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 (Asthana et al., 2019; Y. Yu et al., 2016), diagnosing issues (Bansal et al., 2020; Bhagwan et al., 2018; Mehta et al., 2020), refactoring (Ren et al., 2019; Wyrich & Bogner, 2019), and even intent understanding of the code changes (S. Wang et al., 2020; S. Wang et al., 2019b). 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 IFTTT (if-this-then-that) (Ovadia, 2014) and Microsoft Power Automate ("Microsoft Power Automate", 2020) can be used for creating various automation workflows in domains such as smart home automation (Kumer et al., 2021), healthcare (Faro et al., 2021), and smart mobility (Menon et al., 2020). 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 ("GitHub", n.d.) could check if a pull request is active. If it is active, 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 which help determine whom the notification should be redirected to.

3.3. Background: A Pull Request's Life Cycle

In this chapter, we assume a pull request goes through the life cycle as depicted in Figure 3.1. 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.

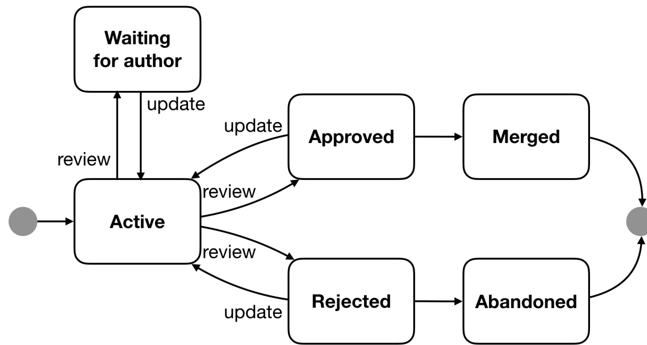


Figure 3.1: The life cycle of a pull request

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, pull request will have a better chance of making progress quickly. In Figure 3.1, 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 3.1, 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 3.1. 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 3.6.1) by influencing the pull request state changes.

Pull request life-cycle is a complex process involving several actors and activities. However, it's 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. (Baysal et al., 2013) 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.

3.4. Nudge System Design

The side effects manifested by pull requests that are open for longer periods of time, 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.

3.4.1. 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 3.5. 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, 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 last two years (from the date of retraining) to make sure the training data reflects 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 explic-

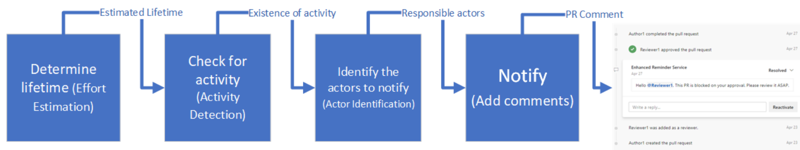


Figure 3.2: Nudge Workflow

itly 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 towards the change blocker.

Nudge Workflow The three modules are combined with a notification system to form Nudge as shown in Figure 3.2. 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 3.5) 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, 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.

3.4.2. 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, etc.

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 six 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 is ingested itself. This helped in reducing the overall training time. The second strategy we adopted is to not train and build repository-specific models if there is 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 which will enable us to scale to 1000s 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, etc. 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 In order 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 which is refreshed automatically when the numbers are updated.

3.5. Pull Request Lifetime Prediction

To be able to 'nudge' developers on overdue pull requests, the Nudge algorithm, first of all, 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 is 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 (B. W. Boehm, 1984; Briand et al., 1999; Chulani et al., 1999; Y. Yu et al., 2015) as discussed in Section 3.2), 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 data set.

3. We use the training data set collected in Step 2 to build a pull request lifetime prediction model and evaluate the performance of the model.

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 which 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

Table 3.1: Feature description and the correlation between features and the pull request lifetime (sorted in the descending order of correlation)

3.5.1. 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 (two 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 3.1). We then used a gradient boosting regression algorithm to perform the regression analysis and calculate coefficients (listed in Table 3.1). 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 3.1 lists all the features we use including their correlation to pull request completion time.

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

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 towards 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 which 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, 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, etc. 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, etc.

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

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 which result in faster completion of pull requests. We used the models developed by Wang et al (S. Wang et al., 2019b) to determine the intent of the pull requests. These are language models which analyze the pull request title and description to classify the intent. We used these Random

Table 3.2: 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

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.

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 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, it takes them less time to merge their changes. We get this information from the human resources database at Microsoft.

3.5.2. 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 3.1, dropping the ones with a very low (absolute) magnitude of correlation. We used 0.008 as a cut-off,

thus dropping three features. This helped to speed up the training and inference tasks without materially impacting the 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 (Mean Absolute Error), and MMRE (Mean Magnitude of Relative Error). 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 3.2. A detailed discussion on prediction accuracy and its significance in the context of the application we are building is presented in Section 3.8.

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 two 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 ("GitHub", 2020), without 'repeated cross-validation', as follows:

1. We separate the data set 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.

3.6. 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 3.1,

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

3.6.1. 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 out 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 3.1):

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

Table 3.3: 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

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 3.7) or when state transitions have been observed. Based on a user study described in Section 3.8, we find that activity detection improves user experience, reduces false alarms, and thus increases the usefulness of the Nudge service.

3.6.2. Actor Identification

In a pull request, there are different actors involved (as explained in Section 3.3) 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 which 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 3.3 lists the classes and the actor responsible, and the number of pull requests that fall under each class. 70 pull requests (out 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

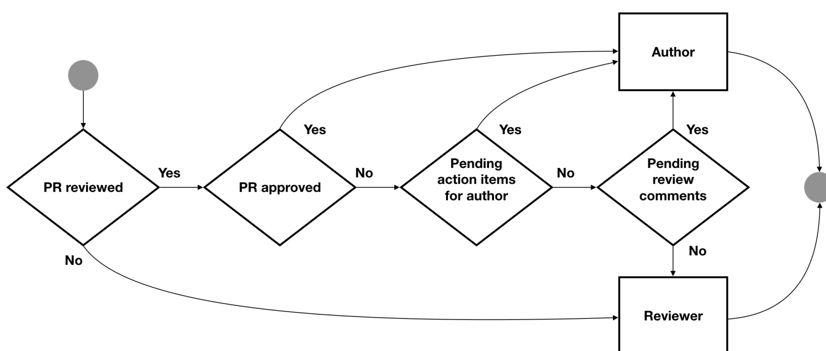


Figure 3.3: Flowchart to determine the change blockers for active pull requests

notifications to the author is prioritized. The flow chart shown in Figure 3.3 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, they can resolve it or if they disagree, 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:

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

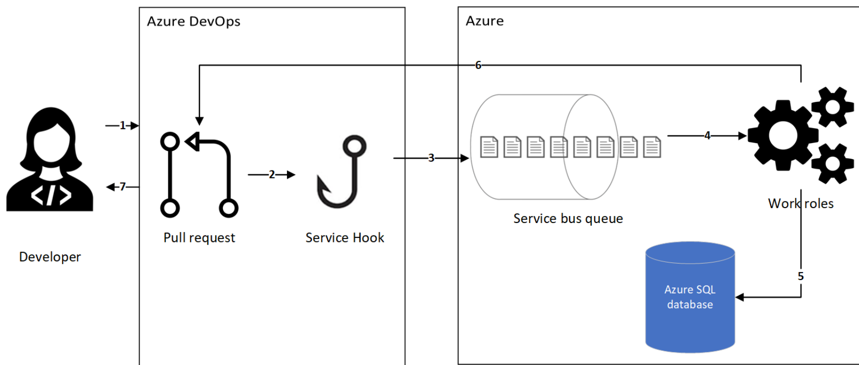


Figure 3.4: Nudge Architecture

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 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, reviewers are accountable and should be notified to unblock the change.

3.7. 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.

3.7.1. Nudge Service Architecture

Figure 3.4 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, 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.

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, 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 six hours, until the pull request is abandoned or completed.

3.7.2. 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 ("Azure DevOps REST API", 2021) 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.

3.7.3. Activity detection

We use Azure DevOps's REST APIs ("Azure DevOps REST API", 2021) to collect data that is 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 3.6.1) 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', etc.

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 3.5). However, it waits at least 24 hours before sending a notification, if there has been any activity observed.

3.7.4. ACTOR IDENTIFICATION

We rely on Azure DevOps's REST APIs to collect data for identifying the actors. In line with Section 3.6.2, 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.

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, 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), we prioritize notifications to the reviewers in the list of reviewers who commented.

3.7.5. NUDGE NOTIFICATION

Figure 3.5 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.

3.8. 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:

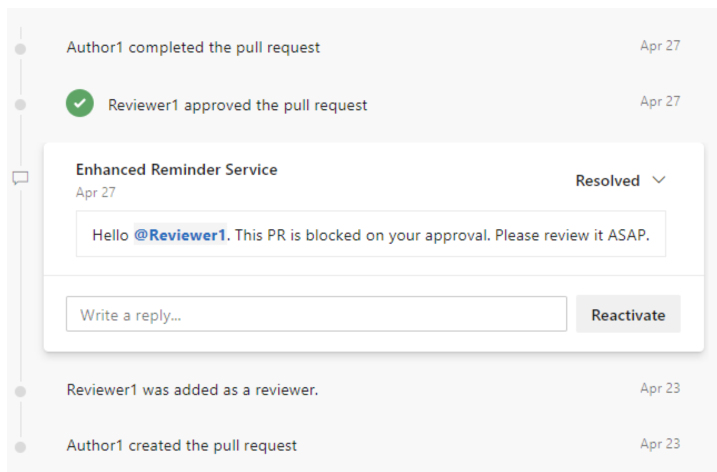


Figure 3.5: Nudge notification, with @Reviewer1 tagged in the reminder

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?

3.8.1. DATA COLLECTION AND METHODOLOGY

We obtained data from the large-scale deployment of the Nudge service for nine months on 147 repositories in Microsoft. The data includes 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 nine-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 data set. 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 data set is independent of the data used to train the model (as explained in Section 3.5.2). As for the correlation analysis (Section 3.5.1), 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 3.6 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 which 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 (Dell et al., 2012).

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

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

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.

Write a reply... Reactivate

Figure 3.6: A pull request with a lifetime prediction notification in Azure DevOps

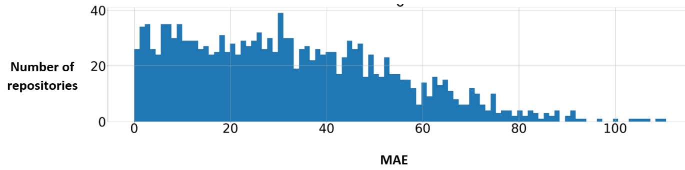


Figure 3.7: MAE distribution

3.8.2. 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: Mean Absolute Error (MAE) and Mean Magnitude of Relative Error (MMRE). For the pull request level effort estimation model, the MAE is 32.60 hours (Figure 3.7 shows the distribution of MAE) and MMRE is 0.58 (Figure 3.8 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.

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.

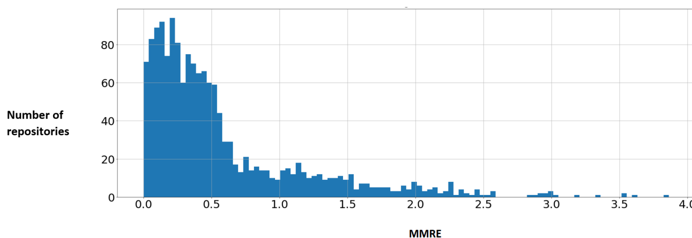


Figure 3.8: MMRE distribution

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 3.6. 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 the figure3.6, 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.

3.8.3. 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.

Table 3.4: Comparison of average pull request lifetime (hours)

Service	Avg PR lifetime	Number of PRs
None	197.2	3856
Nudge-LT	112.6	4117
Nudge-FULL	77.7	4383

2. Distribution of the number of pull requests that are completed within a day, in three 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), etc. Therefore, if we compare pull requests from two different repositories or from two different time periods, 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 3.4 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 3.9, we plot the distribution of pull requests that are completed within a working day, three days, a week or more than a week after Nudge sent the notification. Only 1570 pull requests out of 8500 pull requests (18.47%) have taken more than a

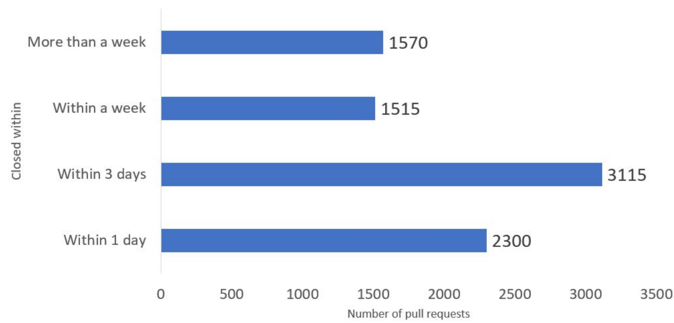


Figure 3.9: Distribution of completed pull requests after sending a notification

week to close. 81.53% of the pull requests are closed within a week. An important observation to make is that 2300 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.

3.8.4. RQ3: What are developers' perceptions about the usefulness of the Nudge service?

To understand whether or not users are favorable towards 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.

Notification Resolution

Table 3.5 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

Table 3.5: 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

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 (Liu et al., 2019; Steck, 2011). 73% of the developers who received Nudge notifications explicitly resolving the notifications positively indicates a clear positive sentiment that the developers exhibit towards the Nudge service.

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 towards a terminal state which 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 chapter. 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 towards the completion within eight 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 she 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.

NUDGE-FULL USER FEEDBACK

Consistent with the many positive notification resolutions (Section 3.8.4), 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 towards 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 3.9.

3.8.5. 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 until 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, three 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. Similar 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.

3.9. Discussion

In this chapter, 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.

3.9.1. 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 (Calikli et al., 2010; Nickerson, 1998). 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.

3.9.2. 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 which can be addressed asynchronously. This lightweight workflow is no different from other notifications which 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.

3.9.3. 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 life cycles. In case of bugs and incidents, faster code reviews can help with faster resolution of bugs and quicker service restoration.

In this chapter 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. On the other hand, 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.

3.9.4. 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 3.8.2, 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.

3.9.5. Addressing Nudge Limitations

20% 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.

Lastly, the Nudge system, at this point, does not "learn" based on user feedback. If a user passes negative feedback, 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.

3.9.6. THREATS TO VALIDITY

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. Lastly, for the error estimation of the machine learning models, we have used a single run of the ten-fold cross-validation. Using repeated cross-validation can result in a more accurate estimation of the performance of machine learning models.

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, etc. However, the coefficients or the factors that impact the completion time of the pull requests, change blockers, etc 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.

3.10. CONCLUSION

Pull request is a key part of the collaborative software development process. In this chapter, 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 which were *nudged*.

We further scaled out Nudge to 8500 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 chapter 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.

4

ConE: A Concurrent Edit Detection Tool for Large Scale Software Development

Modern, complex software systems are being continuously extended and adjusted. The developers responsible for this may come from different teams or organizations, and may be distributed over the world. This may make it difficult to keep track of what other developers are doing, which may result in multiple developers concurrently editing the same code areas. This, in turn, may lead to hard-to-merge changes or even merge conflicts, logical bugs that are difficult to detect, duplication of work, and wasted developer productivity. To address this, we explore the extent of this problem in the pull request based software development model. We study half a year of changes made to six large repositories in Microsoft in which at least 1,000 pull requests are created each month. We find that files concurrently edited in different pull requests are more likely to introduce bugs. Motivated by these findings, we design, implement, and deploy a service named ConE (Concurrent Edit Detector) that proactively detects pull requests containing concurrent edits, to help mitigate the problems caused by them. ConE has been designed to scale, and to minimize false alarms while still flagging relevant concurrently edited files. Key concepts of ConE include the detection of the Extent of Overlap *between pull requests, and the identification of* Rarely Concurrently Edited Files*. To evaluate ConE, we report on its operational deployment on 234 repositories inside Microsoft. ConE assessed 26,000 pull requests and made 775 recommendations about conflicting changes, which were rated as use-*

ful in over 70% (554) of the cases. From interviews with 48 users we learned that they believed ConE would save time in conflict resolution and avoiding duplicate work, and that over 90% intend to keep using the service on a daily basis.

4.1. Introduction

In a collaborative software development environment, developers, commonly, work on their individual work items independently by forking a copy of the code base from the latest main branch and editing the source code files locally. They then create pull requests to merge their local changes into the main branch. With the rise of globally distributed and large software development teams, this adds a layer of complexity due to the fact that developers working on overlapping parts of the same codebase might be in different teams or geographies or both. While such collaborative software development is essential for building complex software systems that meet the expected quality thresholds and delivery deadlines, it may have unintended consequences or 'side effects' (Accioly et al., 2018; C. Bird et al., 2009; Kalliamvakou et al., 2014; McKee et al., 2017). The side effects can be as simple as syntactic merge conflicts, which can be handled by version control systems (Rochkind, 1975) and various techniques/tools (de Souza et al., 2003b; Grinter, 1995; Horwitz et al., 1989), to semantic conflicts (Fowler, 2020). Such bugs can be very hard to detect and may cause substantial disruptions (Horwitz et al., 1989). Primarily, all of this happens due to lack of awareness and early communication among developers editing the same source code file or area, at the same time, through active pull requests.

There is no substitute to resolving merge or semantic conflicts (or fixing logical bugs or refactoring duplicate code) when the issue is manifested. Studies show that pull requests getting into merge conflicts is a prevalent problem (Brindescu et al., 2019; Brun, Holmes, Ernst, et al., 2011; Kasi & Sarma, 2013a). Merge conflicts have a significant impact on code quality and can disrupt the developer workflow (Ahmed et al., 2017; de Souza et al., 2003a; Nieminen, 2012). Sometimes, the conflict becomes so convoluted that one of the developers involved in the conflict has to abandon their change and start afresh. Because of that, developers often defer resolving their conflicts (Nelson et al., 2019) which makes the conflict resolution even harder at a later point of time (Berczuk & Appleton, 2002; Nelson et al., 2019). Time spent in conflict resolution or refactoring activities is going to take away valuable time and prohibits developers from fulfilling their primary responsibility, which is to deliver value to the organization in the form of new functionality, bug fixes and maintaining the service. In addition to loss of time and money, this causes frustration (C. Bird & Zimmermann, 2012; Sarma et al., 2003). Studies have shown that these problems can be avoided by following strategies such as effective communication within the team (Guzzi et al., 2015), and developing awareness about others' changes that have a potential to incur conflicts (Estler et al., 2014).

Our goal is to design a method to help developers discover changes made on other branches that might conflict with their own changes. This goal is particularly

challenging for modern, large scale software development, involving thousands of developers working on a shared code base. One of the design choices that we had to make was to minimize the false alarms by making it more conservative. Studies have shown that, in large organizations, tools that generate many false alarms are not used and eventually deprecated (Winters et al., 2020).

The direct source of inspiration for our research is complex, large scale software development as taking place at Microsoft. Microsoft employs ~166K employees worldwide and 58.6% of Microsoft's employees are in engineering organizations. Microsoft employs ~69K employees outside of the United States making it truly multinational ("Microsoft Facts", 2020). Because of the scale and breadth of the organization, tools and technologies used across the company, it is very common for Microsoft's developers to constantly work on overlapping parts of the source code, at the same time, and encounter some of the problems explained above.

Over a period of twelve months, we studied pull requests, source control systems, code review tools, conflict detection processes, and team and organizational structures, across Microsoft and across different geographies. This greatly helped us assess the extent of the problem and practices followed to mitigate the issues induced by the collaborative software development process. We make three key observations:

1. *Discovering others' changes is not trivial.* There are several solutions offered by source control systems like GitHub or Azure DevOps ("Azure DevOps", 2021; "GitHub", 2020) that enable developers to subscribe to email notifications when new pull requests are created or existing ones are updated. In addition, products like Microsoft Teams or Slack can show a feed of changes that are happening in a repository a user is interested in. The notification feed becomes noisy over time and it becomes very hard for developers to digest all of this information and locate pull requests that might cause conflicts. This problem is aggravated when a developer works on multiple repositories.

2. *Tools have to fit into developers' workflows.* Making developers install several client tools and making them switch their focus between different tools and windows is a big obstacle for adoption of any solution. There exists a plethora of tools (Biehl et al., 2007; Brun, Holmes, Ernst, et al., 2011a; Sarma et al., 2003) that aim to solve this problem in bits and pieces. Despite this, usability is still a challenge because none of them fit naturally into developers' workflows. Therefore, they cause more inconvenience than the potential benefits they might yield.

3. *Suggestions about conflicting changes must be accurate and scalable.* There exist solutions which attempt to merge changes proactively between a developer's local branch and the latest version of main branch or two developer branches. These tools notify the developers when they detect a merge conflict situation (Brun, Holmes, Ernst, et al., 2011a, 2013; Sarma et al., 2003). Such solutions are impractical to implement in large development environments as

the huge infrastructure costs incurred by them may outweigh the gains realized in terms of saved developer productivity.

Keeping these observations in mind, we propose ConE, a novel technique to i) calculate the Extent Of Overlap (EOO) between two pull requests that are active at the same time frame, and ii) determine the existence of Rarely Concurrently Edited files (RCEs). We also derived thresholds to filter out noise and implemented ranking techniques to prioritize conflicting changes.

We have implemented and deployed ConE on 234 repositories across different product lines and large scale cloud development environments within Microsoft. Since deployed, in March 2020, ConE evaluated 26,000 pull requests and made 775 recommendations about conflicting changes.

This chapter describes ConE and makes the following contributions:

- We characterize empirically how concurrent edits and the probability of source code files introducing bugs vary based on the fashion in which edits to them are made, i.e., concurrent vs non-concurrent edits (Section 4.3).
- We introduce the ConE algorithm that leverages light-weight heuristics such as the extent of overlap and the existence of rarely concurrently edited files, and ConE's thresholding and ranking algorithm that filters and prioritizes conflicting changes for notification (Section 4.4).
- We provide implementation and design details on how we built ConE as a scalable cloud service that can process tens of thousands of pull requests across different product lines every week (Section 4.5).
- We present results from our quantitative and qualitative evaluation of the ConE system (Section 4.6).

To the best of our knowledge, this is the first study of an early conflict detection system that is also deployed, in a large scale, cloud based, enterprise setting comprised of a diverse set of developers who work with multiple frameworks and programming languages, on multiple disparate product lines and who are from multiple geographies and cultural contexts. We have observed overwhelmingly positive response to this system with a 71.48 % positive feedback provided by the end users: A very good user interaction rate (2.5 clicks per recommendation that is surfaced by ConE to learn more about conflicting changes) and 93.75% of the users indicating their intent to use or keep using the tool on a daily basis.

Our interactions and interviews with developers across the company made us realize that developers find it valuable to have a service that can facilitate better communication among them about edits that are happening elsewhere (to the same files or functions that are being edited by them) through simple and non-obtrusive notifications. This is reflected strongly in the qualitative feedback that we have received (explained in detail in section 4.6).

4.2. Related Work

The software engineering community has extensively studied the impact of merge conflicts on software quality (Ahmed et al., 2017; Brindescu et al., 2019), investigated various methodologies and tools that can help developers discover conflicting changes through interactive visualizations, and developed speculative analysis tools (de Souza et al., 2003b; Grinter, 1995; Horwitz et al., 1989). While ConE draws inspiration from some of this prior work, it is more ambitious, targeting a method that is effective while not resource intensive, can be easily scaled to work on tens of thousands of repositories of all sizes, is easy to integrate and fits naturally into existing software development workflows and tools with very little to no disruption.

A conflict detection system that has to work for large organizations with disparate sets of programming languages, tools, product portfolio and has thousands of developers that are also geographically distributed, has to satisfy the requirements listed below:

- *language-independent*: the techniques and tooling built should be language-independent in nature and support repositories that hosts code written in any programming language and should support new languages with no or minimal customization.
- *non-intrusive*: the recommendations passed by the tool should naturally fit into developer workflows and environment.
- *scalable*: finally, the techniques proposed and the system should be performant and responsive without consuming a lot of computing resources and demanding lot of infrastructure to scale them up.

  We now explain some of the prior work that is relevant and explain why they do not satisfy some or all of the requirements.

Tools based on edit activity. Manhattan (Lanza et al., 2013) is a tool that generates visualizations about team activity whenever a developer edits a class and notifies developers through a client program, in real time. While this shows useful 3D visualizations about merge conflicts in the IDE itself (thus being non-intrusive and natural to use), it is not adaptive (it does not automatically reflect any changes to the code in the visualization, unless the user decides to re-import the code base), not generic (it works only for Java and Eclipse) and not scalable as it operates on the client side and has to go through the cycle of import-analyze-present again and again for every change that is made, inside the IDE environment. Similarly, FASTDash (Biehl et al., 2007) is a tool that scans every single file that is edited/opened in every developer local workspace and communicates about their changes back and forth through a central server. This is impractical to implement across large development teams. It requires tracking changes at the client side with the help of an agent program that runs on each client. Furthermore it then keeps listening to every file edit activity in the workspace,

then communicating that information with a central server which mediates communication between different workspaces. This is prone to failures and runs into scale issues even with a linear increase in developers and pull requests in the system.

Tools based on early merging. Some tools were built upon the idea of attempting actual merging and notifying the developers through a separate program that runs on the client (Brun, Holmes, Ernst, et al., 2011a, 2013; Sarma et al., 2003). These solutions are very resource intensive because the system needs to perform the actual source code merge for every pull request or developer branch with the latest version of the main branch (despite implementing optimization techniques like caching and tweaking the algorithm to compute relationships between changes when there is a change to the history of the repository). It is not possible to implement and scale this at a company like Microsoft where tens of thousands of pull requests are created every week. Additionally, these solutions do not attempt to merge between two different user branches or two different active pull requests but attempt to merge a developer branch with the latest version of the main branch. This will not find conflicting changes that exist in independent developer branches and thus cannot trigger early intervention. Palantir (Sarma et al., 2003) is a tool that addresses some of the performance issues by leveraging a cache for doing dependency analysis. It is, however, still hard to scale due to the fact that there is client-server communication involved between IDEs and centralized version control servers to scan, detect, merge and update every workspace with information about remote conflicting changes. Some solutions explore speculative merging (Brun, Holmes, Ernst, et al., 2011b; Guimarães & Silva, 2012; Kasi & Sarma, 2013b) but the concerns with scalability, non-obtrusiveness remain valid with all of them.

Predictive tools. Owhadi-Kareshk *et al.* explored the idea of building binary classifiers to predict conflicting changes (Owhadi-Kareshk et al., 2019). Their model consists of nine features, of which the number of jointly edited files is the dominant one. The model has been evaluated on a dataset of syntactic merge conflicts reverse engineered from git histories. The model's reported performance in terms of precision ranges from 0.48 to 0.63 (depending on the programming languages).

While one of our proposed metrics, our Extent of Overlap, is akin to the dominant feature in Owhadi-Kareshk's model, unfortunately their proposed approach cannot be applied in our context. In particular the reported precision is too low and would generate too many false alarms which would render our tool unused (Winters et al., 2020). Furthermore, the reported precision and recall are measured based on a gold standard of syntactic changes. Instead, we target an evaluation with actual developers, based on a service deployed on repositories they are working with on a daily basis. As we will see in our evaluation, these developers not only value warnings about syntactic changes, but also semantic conflicts (Fowler, 2020), or even cases of code/effort duplication (as explained in Section 4.6.3).

Empirical studies of merge conflicts and collaboration. There exists many studies that do not propose tools, but study merge conflicts or present methods to

predict conflicts or recommend coordination. Zhang *et al.* (F. Zhang et al., 2012) conducted an empirical study of the effect of file editing patterns on software quality. They conducted their study on three open source software systems to investigate the individual and the combined impact of the four patterns on software quality. To the best of our knowledge ours is the first empirical study that is conducted at scale, on industry data. We perform analysis on 67K bug reports, from 83K files (in comparison to the studies conducted by Zhange *et al.* which looked at 98 bugs, from 2,140 files).

Ashraf *et al.* presented reports from mining cross-task artifact dependencies from developer interactions (Ashraf et al., 2019). Dias *et al.* proposed methods to understanding predictive factors for merge conflicts (Dias et al., 2020a), i.e., how conflict occurrence is affected by technical and organizational factors. Studies conducted by Blincoe *et al.* and Cataldo *et al.* (Ashraf et al., 2019; Cataldo et al., 2006) show the importance of timely and efficient recommendations and the implications for the design of collaboration awareness tools. Studies like this form a basis for building solutions that are scalable and responsive (the large-scale ConE service that we deployed at Microsoft) and their importance in creating awareness of the potential conflicts.

Costa *et al.* proposed methods to recommend experts for integrating changes across branches (Costa et al., 2016) and characterized the problem of developers' assignment for merging branches (Costa et al., 2014). They analyzed merge profiles of eight software projects and checked if the development history is an appropriate source of information for identifying the key participants for collaborative merge. They also presented a survey on developers about what actions they take when they need to merge branches, and especially when a conflict arises during the merge. Their studies report that the majority of the developers (75%) prefer collaborative merging (as opposed to merging and taking decisions alone). This reiterates the fact that tools that facilitate collaboration, by providing early warnings, are important in handling merge conflict situations.

4.3. CONCURRENT VERSUS NON-CONCURRENT EDITS IN PRACTICE

The differences in the fashion in which edits are made to source code files (concurrent vs non-concurrent) can cause various unintended consequences (as explained in section 4.1). We performed large scale empirical analysis of source code edits to understand the ability of concurrent edits to cause bugs. We picked bugs as a candidate for our case study because it is relatively easy to mine and generate massive amounts of ground truth data about bugs and map them back to the changes that induced the bugs, by leveraging some of the techniques proposed by Wang *et al.* (S. Wang et al., 2019a), at Microsoft's scale. Understanding the extent of the problem, i.e., the side effects caused by concurrent source code edits in a systematic way, is an essential first step towards making a case for building an early intervention service like ConE. This allows us to quickly sign up customers inside the company and deploy the ConE

system on thousands of repositories, for tens of thousands of developers, across Microsoft. To that extent, we formulate two research questions that we would like to find answers for.

- **RQ1** How do concurrent and non-concurrent edits to files compare in the number of bugs introduced in these files?

- **RQ2** To what extent are concurrent, non-concurrent, and all edits, correlated with subsequent bug fixes to these files?

Answering the questions above allows us to assess the urgency of the problem. The methods, techniques and outcomes used can also be employed to inform decision makers, when investments in the adoption of techniques like ConE need to be made.

We performed an empirical study on data that is collected from multiple, differently sized repositories. For our study, we focused on one of the important side effects that is induced by collaborative software development, i.e., the "number of bugs introduced by concurrent edits". We chose this scenario as we have an option to generate an extensive set of ground truth data, by leveraging techniques proposed by Wang *et al.* (S. Wang et al., 2019a), to tag pull requests as bug fixes. They employ two simple heuristics to tag bug fixes: the commit message should contain the words "bug" or "fix", but not "test case" or "unit test". Tagging changes that introduce bugs is not a practice that is followed very well in organizations. Studies have shown that files changed in bug fixes can be considered as a good proxy to files that introduced the bugs in the first place (Kim et al., 2006; Williams & Spacco, 2008). Combining both ideas we created a ground truth data set which we used in our empirical analysis. We broadly classify our empirical study into three main steps.

1. Data collection: Collect data using the data ingestion framework that we have built which ingests meta data about pull requests (author, created/closed dates, commits, reviewers etc), iterations/updates of pull requests, file changes in pull requests, and intent of the pull request (feature work, bug fix, refactoring etc).

2. Use the data collected in Step 1 to analyze the impact of concurrent edits on bugs or bug fixes in comparison to non-concurrent edits.

3. Explain the differences in correlations between concurrently versus non-concurrently edited files to the number of bugs that they introduce.

For the purpose of the empirical analysis we define concurrently and non-concurrently edited files as follows:

- Concurrently edited files: Files which have been edited in two or more pull requests, at the same time, while the pull requests are active. A pull request is in an 'active' state when it is being reviewed but not completed or merged.

- Non-concurrently edited files: Files which have never been edited in two pull requests while they both are in active state. So, we are sure that changes made to these files are always made in the latest version and are merged before they are edited through another active pull request

4.3.1. Data Collection

We collected data about file edits (concurrent and non-concurrent) from the pull request data, for six months, from six repositories. We picked repositories in which at least 1,000 pull requests are created every month. After reducing the repositories to a subset, we randomly selected six repositories for the purpose of the analysis. We made sure our data set is representative in various dimensions like size (small (1), medium (2), large (3)), the nature of the product (on-prem product (2) vs cloud service (4)), geographical distribution of the teams (US only (2) versus split between different countries and time zones (4)), and programming languages (as listed in Table 4.3). We performed data cleansing by applying the filters listed below:

- Exclude PRs that are open for more than 30 days: the majority of these pull requests are 'Stale PRs' which will be left open forever or abandoned at a later point of time. Studies shows that 70% of the pull requests gets completed within a week after creation (Gousios et al., 2014).

- Exclude PRs with more than 50 files (this is the 90th percentile for file counts in our pull request data set). This is one of the proxies that we use to to exclude PRs which are created by non-human developers that do mass refactoring or styling changes etc.

- Exclude edits made to certain file types. We are primarily interested in understanding the effects of concurrent edits on source code changes as opposed to files like configuration or initialization files which are edited by lot of developers through lot of concurrent pull requests, all the time. For the purpose of this study, we consider only the following file types: .cs, .c, .cpp, .ts, .py, .java, .js, .sql.

- Exclude files that are edited a lot: For example, files that contain global constants, key value pairs, configuration values, or enums are usually seen in a lot of active pull requests at the same time. We studied 200 pull requests to understand the concurrent edits to these files. They typically are in the order of a few thousands of lines in size, which is well above the median file size. In all cases the edits are localized to different areas of the files and surgical in nature. Sometimes, the line numbers of the edits are far away (few thousands of lines away, at least). Therefore, we impose a filter on the edit count of fewer than twenty times in a month (90th percentile of edit counts for all source code files) and exclude any files that are edited more than this. Without this filter, these

Table 4.1: Distribution of concurrently and non-concurrently edited files per repository

Repo	Distinct number of concurrently edited files	Distinct number of non-concurrently edited files	Number of bug fix pull requests	Percentage of concurrently edited files	Percentage of non-concurrently edited files
Repo-1	3500	4875	4781	41.7	58.2
Repo-2	10470	16879	15678	38.2	61.8
Repo-3	2907	4119	5467	41.3	58.7
Repo-4	5560	7550	8972	42.4	57.6
Repo-5	4110	7569	9786	35.2	64.8
Repo-6	5987	9541	9443	38.5	61.5
Total	32534	50533	54127	39.1	60.9

frequently edited files would dominate the results of the ConE recommendations thus yielding too many warnings for harmless concurrent edits.

We started with a data set of 208,556 pull requests. As bug fixes is our main concentration for the empirical analysis, we removed all the pull requests that are not bug fixes. That reduced the data set to 67,155 pull requests (32.2% of the pull requests are bug fixes). Then we applied other filters mentioned above, which further reduced the data set to 54,127 pull requests (25.95%). Table 4.1 shows the distribution of concurrently and non-concurrently edited files per repository.

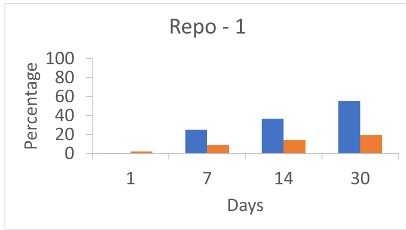
4.3.2. RQ1: Concurrent versus non-concurrent bug inducing edits

We take every (concurrently or non-concurrently) edited file, and check whether the nature of the edit has any effect on the likelihood of that file appearing in bug fixes after the edit has been merged. We compare how the percentage of edited files that are seen in bug fixes (within a day, a week, two weeks and a month), varies with the nature of the edit (concurrent vs non-concurrent).

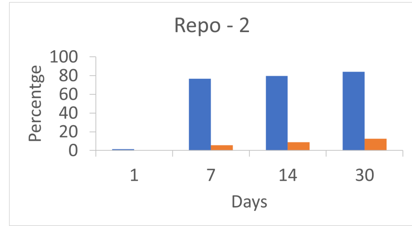
Figure 4.1 shows the impact of concurrent versus non-concurrent edits on the number of bugs being introduced. Across all six repositories, the percentage of bug inducing edits is consistently higher for concurrently edited files (blue bars) than for non-concurrently edited ones (orange bars).

4.3.3. RQ2: Edits in files versus bug fixes in files

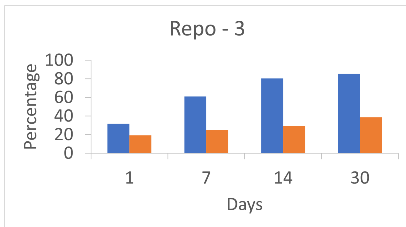
We use Spearman's rank correlation to analyze how the total number of edits, concurrent edits, and non-concurrent edits to files each correlate with the number of bug fixes seen in those files.



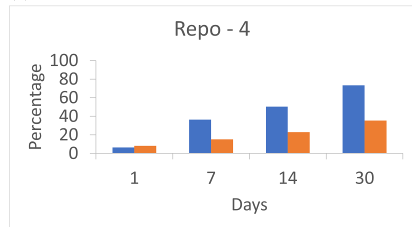
(a)



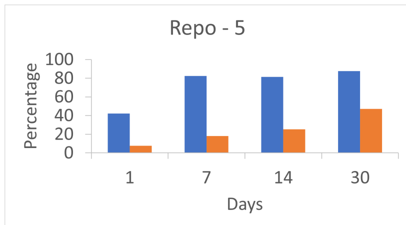
(b)



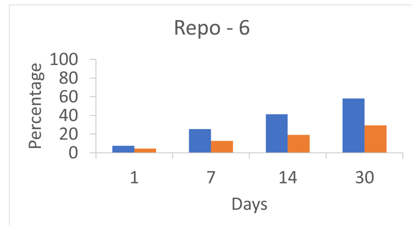
(c)



(d)



(e)



(f)

Figure 4.1: Graphs showing how the percentage of files seen in bug fixes (within a day, a week, two weeks and a month) is changing. Blue and orange bars represent concurrently and non-concurrently edited files, respectively.

Table 4.2: Spearman rank correlation analysis for total edits, concurrent edits, non-concurrent edits vs bug fixes.

Repo	Total Edits to Bug Fixes	Concurrent Edits to Bug Fixes	Non-Concurrent Edits to Bug Fixes
Repo-1	0.145***	0.298***	0.034**
Repo-2	0.072***	0.140***	0.057**
Repo-3	0.140*	0.330*	0.120*
Repo-4	−0.077***	0.451***	−0.461***
Repo-5	0.164***	0.472***	0.091***
Repo-6	0.084**	0.196***	0.005*

$^{***}p < 0.001$, $^{**}p < 0.01$, $^{*}p < 0.05$

While Figure 4.1 shows that more concurrently edited files are seen in bug fix pull requests (compared to non-concurrently edited ones), this might also be because these files are frequently edited and seen in bug fix pull requests naturally. To validate this, we performed Spearman rank correlation analysis for each file that is ever edited with respect to how many times it is seen in bug fixes (the numbers of data points from the six repositories are listed in Table 4.1):

- The total number of times a file is seen in *all* completed pull requests vs the number of bug fixes in which it is seen

- The total number of times a file is seen in *concurrent* pull requests vs the number of bug fixes in which it is seen

- The total number of times a file is seen in *non-concurrent* pull requests vs the number of bug fixes in which it is seen

The results are in Table 4.2. We observe that concurrent edits (third column) consistently are correlated with bug fixes, more so than non-concurrent edits (column 4) and all edits (column 2). For all repositories except Repo-4, there exists almost no correlation between non-concurrent edits (column 4) and bug fixes.

For Repo-4, frequently edited files are not necessarily the ones seen in more bug fixes: there exists a *negative* correlation between total edits (column 2) and the number of bug fixes. However, files that are *concurrently* edited (column 3) do have a positive correlation with the number of bug fixes.

The variety in the correlations can be explained by the fact that concurrent editing is just one of many factors related to the need for bug fixing. Other factors might include the level of modularization, developer skills, the test adequacy, engineering system efficiency, and so on.

4.4. System Design

Backed by the correlation analysis suggesting that concurrent edits maybe prone to causing issues. Also, there exists a huge demand from engineering organizations, inside Microsoft, for a better tool that can detect conflicting changes early on and facilitate better communication among developers, we moved forward to materialize the idea of ConE into reality. We then performed large scale testing and validation by deploying ConE on 234 repositories. Details about the implementation, deployment and scale-out are provided in section 4.5.

In this section we describe ConE's conflict change detection methodology, algorithm and system design in detail. We will use the following terminology:

- *Reference pull request* is a pull request in which a new commit/update is pushed thus triggering the ConE algorithm to be run on that pull request.
- *Active pull request* is a pull request whose state is 'active' when the ConE algorithm is triggered to be run on a reference pull request.

A key design consideration is that we want to avoid false alarms. In the current state of the practice developers never receive warnings about potentially harmful concurrent edits. Based on this we believe it is acceptable to miss a few warnings. On the other hand, giving false warnings will likely lead to rejection of a tool like ConE. For that reason, ConE has several built-in heuristics that are aimed at reducing such false alarms.

Due to the nature of the problem, the domain we are operating in, and the algorithm we have in-place, it is possible to see notifications that are false alarms. One of the design choices that we had to make was to minimize the false alarms by making it more conservative. A side effect of this is our coverage (number of pull requests for which we send a notification) will be lower. Studies have shown that, in large organizations, tools that generate many false alarms are not used and eventually deprecated (Winters et al., 2020). However, recent techniques proposed by Brindescu *et al.* (Brindescu et al., 2020), can potentially aid in facilitating a decision by determining the merge conflict situations to flag, based on the complexity of the merge conflict.

4.4.1. Core Concepts

ConE constantly listens to events that happen in an Azure DevOps environment ("Azure DevOps", 2021). When any new activity is recorded (e.g., pushing a new update or commit) in a pull request, the ConE algorithm is run against that pull request. Based on the outcome, ConE notifies the author of the pull request about conflicting changes. We describe two novel constructs that we came up with for detecting conflicting changes and determining candidates for notifications: Extent of overlap (EOO) and the existence of 'Rarely Concurrently Edited' files (RCEs). Next, we provide a detailed description of ConE's conflict change detection algorithm and the parameters we have in place to tune ConE's algorithm.

Extent of Overlap (EOO)

ConE scans all the active pull requests which meet our filtering criteria (explained in section 4.3.1) and for each such pull request (reference pull request) calculates the percentage of files edited in the reference pull request that overlap with each of the active pull requests.

$$Extent\ of\ Overlap = \frac{|\, F_r \cap F_a - F_e \,|}{|\, F_r \,|} * 100$$

where F_r = Files edited in reference pull request, F_a = Files edited in a given active pull request, F_e = Files excluded i.e., files that are not of types listed in the paragraph below. The idea is to find the percentage of items that are commonly edited in multiple active pull requests and create a pairwise overlap score for each of the active and reference pull request pairs. Intuitively, if the overlap between two active pull requests is high, the probability of them doing duplicate work or causing merge conflicts when they are merged is also going to be high. We use this technique to calculate the overlap in terms of number of overlapping files for now. This can be easily extended to calculate the overlap between two active pull requests in terms of number of classes or methods or stubs if that data is available.

A milder version of EOO is used by the model proposed by Owhadi-Kareshk et al (Owhadi-Kareshk et al., 2019), which looks at the number of files that are commonly edited in two pull requests when determining conflicting changes. While calculating extent of overlap it is important to exclude edits to certain file types whose probability of inducing conflicts is minimal. This helps in reducing false alarms in our notifications significantly. Based on a manual inspection of 500 randomly selected bug fix pull requests, by the first three authors, we concluded that concurrent edits to initialization or configuration files are relatively safe, but that concurrent edits to source code files are more likely to lead to problems. Therefore, we created an *allow list* based on file types as shown in Table 4.3. As can be seen, this eliminates around 6.4% of the files. Note that such an *allow list* is programming language-specific. When ConE is to be applied in different contexts, different allow lists are likely needed.

Rarely Concurrently Edited files (RCEs)

These are the files which typically are *not edited concurrently*, recently. Usually all the updates or edits to them are performed, in a controlled fashion, by a single person or small set of people. Seeing RCEs in multiple active pull requests is an anomalous phenomenon. For example, a file foo.cs is always edited by a given developer, through one active pull request at any point. The ConE system keeps a track of such files and tags them as RCEs. In the future, if multiple active pull requests are seen editing this file simultaneously, ConE flags them. Our intuition is that, if a lot of RCEs are seen in multiple active pull requests, which is unusual, changes to these files should be reviewed carefully and everyone involved in editing them should be aware of others' changes.

Table 4.3: Distribution of file types seen in bug fixes

File type	Percentage	On ConE *allow list*?
.cs	44.32	yes
.cpp	18.55	yes
.c	11.27	yes
.sql	6.20	yes
.java	5.36	yes
.js	3.98	yes
.ts	3.79	yes
.ini	0.20	no
.csproj	0.04	no
others	6.29	no

Table 4.4: Number of Bug Fixes with RCEs and No RCEs

Edit type	Count	Percentage
Bug fix PRs with no RCEs	1617	78.3
Bug fix PRs with at least one RCE	446	21.7

We performed an empirical analysis, from our shadow mode deployment data (as explained in Section 4.5.2), to understand how pervasive RCEs really are. As explained in Table 4.4, 21.7% of bug fixes contains at least one RCE in them while the total number of RCEs in these repositories is *just* 2%. Based on this data and anecdotal feedback from developers, we realized that concurrent edits to RCEs is an unusual activity which should not be seen a lot. But, if observed, it should be notified to all the developers involved.

For building the ConE system, we ran the RCE detection algorithm that looks at the pull requests that are created in a repository within the last three months from when the algorithm runs. The duration can be increased or decreased based on how big or how active the system is. This process, after each run, creates a list of RCEs. Once the initial bootstrapping is done and a list of RCEs is prepared, that list is used by the ConE algorithm when checking for the existence of the RCEs in a pair of pull requests. The RCE list is updated and refreshed once every week, through a separate process. The process of detecting and updating RCEs is resource intensive. So, we need to strike a balance between how quickly we would like to update the RCE list versus how many resources we need to throw at the system, without compromising the quality of the suggestions. We picked one week as the refresh interval through multiple iterations of experiments. This process guarantees that the ConE system reacts to the changes in the rarity of concurrent edits, especially the cases where an RCE becomes a non-RCE due to the concurrent edits it experiences. The steps involved in creating and updating RCEs are listed below.

Creating the RCE list:

1. Get all the pull requests created in the last three months from when the algorithm is run. Create a list of all the files that are edited in these pull requests by applying the filters explained in the paragraph above on file types.

2. Prepare sets of pull requests that overlap with others. Prepare a list of files edited in the overlapping pull requests by applying the filters explained in the paragraph above on file types.

3. The list of files created in step-1 minus the list of files created in step-2 constitutes the list of rarely concurrently edited files (RCEs).

Updating the RCE list:

4. Remove files from the RCE list if they are seen in overlapping pull requests when the algorithm is run the next time. Because, if they are seen in overlapping pull requests, they will not be qualified to be RCEs anymore.

5. Refresh the list by adding the new RCEs discovered in the latest edits, when the algorithm is run again.

4.4.2. The ConE Algorithm

ConE's algorithm to select candidate pull requests that developers need to be notified about primarily leverages the techniques explained above: Extent of Overlap (EOO) and existence of Rarely Concurrently Edited files (RCEs). Together these serve to reduce the total number of active pull requests under consideration, in order to pick the pull requests that need to be notified about. The ConE algorithm consists of seven steps listed below:

Step 1: Check if the reference pull request's age is more than 30 days. Studies have shown that pull requests which are active for so long may not even be completed (Gousios et al., 2014). Exclude all such pull requests.

Step 2: Construct a list of files that are being edited in the reference pull request. While constructing this set, we exclude any files of types that are not in the allow list from Table 4.3.

Step 3: Construct a set of files that are being edited in each of the active pull requests, using the methodology mentioned in Steps 1 and 2. One extra filter that we apply here is to exclude PRs which are being interacted by the author of the reference pull request. If the author of the reference pull request is already aware of this pull request there is no need to notify thems.

Step 4: Calculate the extent of overlap using the formula described in section 4.4.1. For every pair of reference pull request PR_r and active pull request PR_{a1}, calculate the tuple $T_{ea1} = \langle PR_r, PR_{a1}, E_1 \rangle$, where E_1 is the extent of overlap between the two pull requests. Do this for all the active pull requests with respect to a reference

pull request. At the end of this step we have a list of tuples, T_{ea} = [$\langle PR_1, PR_7, 55\rangle$, $\langle PR_1, PR_{12}, 95\rangle$, $\langle PR_1, PR_{34}, 35\rangle$....].

Step 5: Check for the existence of rarely concurrently edited files (RCEs) and the number of RCEs between each pair of reference and active pull request. Create a tuple $T_r = \langle PR_r, PR_{a1}, R_1\rangle$ where PR_r is the reference pull request, PR_{a1} is active pull request and R_1 is the number of RCEs in the overlap of reference and active pull requests. Do this for all reference and active pull request combinations. At the end of this step we have a list of tuples, T_{ra} = [$\langle PR_1, PR_7, 2\rangle$, $\langle PR_1, PR_{12}, 2\rangle$, $\langle PR_1, PR_{34}, 9\rangle$....]

Step 6: Apply thresholds on the values for extent of overlap and the number of RCEs, as explained in section 4.4.3. For example, we can apply a threshold that we select the pull requests whose extent of overlap is greater than 50% OR there should be at least two RCEs. We go through the list of tuples that we have generated in Steps 4 and 5 above and apply the thresholding criteria.

Step 7: Apply a ranking algorithm to prioritize the pull requests that need to be looked at first if multiple pull requests are selected by the algorithm. We rank candidate pull requests based on the number of RCEs present and then by the extent of overlap. This is because RCEs being edited through multiple active pull requests is an anomalous phenomenon which needs to be prioritized.

4.4.3. Default Thresholds and Parameter Tuning

In this section we describe the thresholding criteria, and the rationale that needs to be applied while choosing parameter values for large scale deployment. The parameters that we have in place are: the extent of overlap (EOO), the number of rarely concurrently edited files (RCEs), the window of time period (i.e., the number of months to consider for determining RCEs), and the total number of file edits in the reference PR.

In line with our objectives, we are searching for parameter settings that find actual conflicts, yet minimize false alarms. Furthermore, we target settings that are easy to explain (e.g., "this PR was flagged because half of the files changed it are also touched in another PR").

Threshold for EOO: For Extent of Overlap, we explored what would happen if we put the threshold at 50%: if at least half of the files edited in another pull request, consider it for notification. To assess the consequences of this, we randomly selected 1654 pull requests, which have at least one file overlapped with another pull request. This data set is a subset of the data collected to perform empirical analysis on concurrent edits (see Section 4.3). We manually inspected each of these 1654 pull requests to make sure the overlap we observe is indeed correct. Our empirical analysis (see Table 4.5), shows that 50% of the pull requests have an overlap of 50% or less. Thus, this simple heuristic eliminates half of the candidate pull requests for notification, substantially reducing potential false alarms, and keeping the candidates

Table 4.5: Distribution of extent of overlap (EOO)

Percentage of overlap (range)	Number of PRs
0-10	309
11-20	223
21-30	137
31-40	87
41-50	25
51-60	359
61-70	92
71-80	21
81-90	23
91-100	378

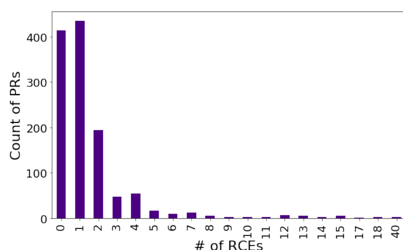


Figure 4.2: Distribution of the number of PRs with a given number of RCEs

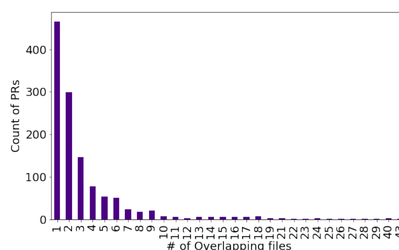


Figure 4.3: Distribution of the number of PRs with a given number of overlapping files.

that are more likely to be in conflict.

Threshold for RCEs: For RCEs we again followed a simple rule: if the active-reference pull request pair contains at least two files that are modified in them, which are always edited in isolation, select the active pull request as a candidate. As shown in Figure 4.2, the majority of the pull requests contains fewer than two RCEs. To be conservative, we imposed a threshold on RCE ≥ 2, i.e., to select a PR as a candidate, that pull request needs to have at least two RCEs that are commonly edited between the reference and active pull requests.

Number of overlapping files: Assume a developer creates a pull request by editing two files and one of them is also edited in another active pull request. Here EOO is 50%. This means this pull request qualifies to be picked as a candidate for notification. Editing just one file in two active pull requests might not be enough to reasonably make an assumption about the potential of conflicts arising. Therefore, we impose a threshold on the "number of files" that needs to be edited, simultaneously, in both

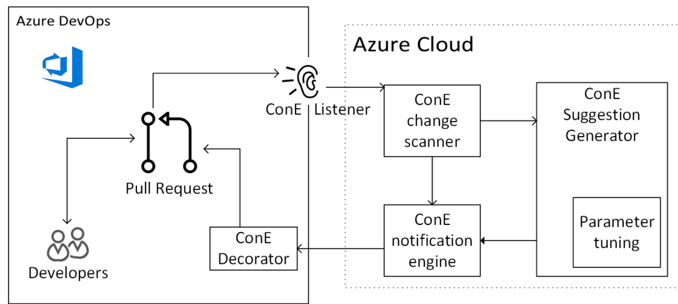


Figure 4.4: ConE System design: Pull requests from Azure DevOps are listened to by the ConE change scanner, suggestion generator, notification engine, and decorator

pull requests. As a starting point, we imposed a threshold of two, i.e., every candidate pull request should have more than two overlapping files (in addition to satisfying the EOO condition of >= 50%). We plotted the distribution of the number of overlapping files in Figure 4.3. As shown in Figure 4.3, the number of PRs (on the Y-axis) drops sharply after the number of overlapping file edits is two. Therefore, we picked two as the default threshold.

Threshold Customization: In addition to the empirical analysis, we collected initial feedback from developers working with the production systems through our shadow mode deployment (Section 4.5.2). One of the prominent requests from the developers was to enable the repository administrators to change the values of the parameters explained above based on the developer feedback. Therefore, we provided customization provisions to make ConE system suit each repository's needs. Based on the pull request patterns and needs of the repository, system administrators can tune the thresholds to optimize the efficacy of the ConE system for particular reporsitories.

4.5. Implementation and deployment

4.5.1. Core Components and Implementation

The core ConE components are displayed in Figure 4.4. ConE is implemented on Azure DevOps (ADO), the DevOps platform provided by Microsoft. We chose to develop ConE on ADO due to its extensibility that allows third party services to interact with pull requests through various collaboration points such as adding comments in pull requests, a rich set of APIs provided by ADO to read meta data about pull requests, and service hooks which allow a third party application to listen to events such as updates that happen inside the pull request environment.

Within Azure DevOps, as shown in the left box of Figure 4.4, ConE listens to events triggered by pull requests, and has the ability to decorate pull requests with

notifications about potentially conflicting other pull requests. The ConE service itself, shown at the right in Figure 4.4, runs within the Azure Cloud. The ConE change scanner listens to pull request events, and dispatches them to workers in the ConE suggestion generator. Furthermore, the scanner monitors telemetry data from interactions with ConE notifications. The core ConE algorithm is offered as a scalable service in the Suggestion Generator, with parameters tunable as explained in Section 4.4.3.

The ConE Service is implemented using C# and .NET 4.7. It has been built on top of Microsoft Azure cloud services: Azure Batch ("Azure Batch", 2020) for compute, Azure DevOps service hooks for event notification, Azure worker roles and its service bus for processing events, Azure SQL for data storage, Azure Active Directory for authentication and Application Insights for telemetry and alerting.

4.5.2. ConE Deployment

We selected 234 repositories to pilot ConE in the first phase. Some of the key attributes based on which the repository selection process has taken place are listed below:

- Prioritize repositories where we have developers and managers who volunteered to try ConE, since we expect them to be willing to provide meaningful feedback.
- Include repositories that are of different sizes (based on the number of files present in them): very large, large, medium, and small.
- Include repositories that host source code for diverse set of products and services. That includes client side products, mobile apps, enterprise services, cloud services, and gaming services.
- Consider repositories which have cross-geography and cross-timezone collaborators, as well as repositories that have most of the collaborators from a single country.
- Consider repositories that host source code written in multiple programming languages including combinations of Java, C#, C++, Objective C, Swift, Javascript, React, SQL etc.
- Include repositories which contain a mix of developers with different levels of experience (based on their job titles): Senior, mid-level and junior.

We enabled ConE in *shadow mode* on 60 repositories for two months (with a more liberal set of parameters to maximize the number of suggestions we generate). In this mode we actively listen to pull requests, run the ConE algorithm, generate suggestions, and save all the suggestions in our SQL data store for further analysis, without sending the notifications to the developers. We generated and saved 1200 suggestions by enabling ConE in this mode for two months. We then went through

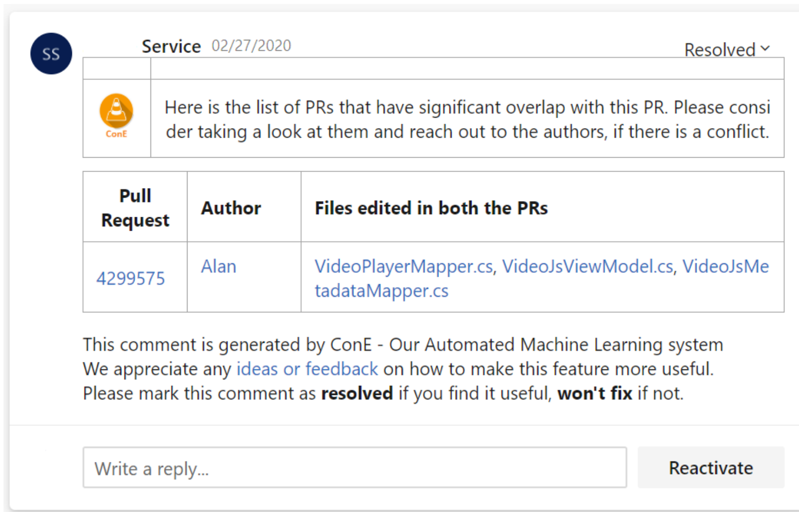


Figure 4.5: ConE notifications that a pull requesthas significant overlap with another pull request.

the suggestions and the telemetry collected to optimize the system before a large scale roll out.

The primary purpose of shadow mode deployment is to validate whether operationalizing a service like ConE is even possible at the scale of Microsoft. Furthermore, it allowed us to check whether we indeed can flag meaningful conflicting pull requests, and what developers would think of the corresponding notifications. The telemetry we collected includes the time it takes to run the ConE algorithm, resource utilization, the number of suggestions the ConE system would have made, etc. We experimented with tuning our parameters (explained in subsection 4.4.3) and their impact on the processing time and system utilization. This helped us in understanding the scale and infrastructure requirements and overall feasibility.

We collected feedback from the developers by reaching out to them directly. We have shown them the suggestions we would have made if the ConE system was enabled on their pull requests, format of the suggestions and the mode of notifications. We iterated over the design of the notification based on the user feedback before settling on the version of the notification as shown in Figure 3.5.

After multiple iterations of user studies and feedback collection, on the design, frequency, and the quality of the ConE suggestions as validated by the developers participated in our shadow mode deployment program, we turned on the notifications on 234 repositories.

4.5.3. Notification Mechanism

We leveraged Azure DevOps's collaboration points to send notifications to developers. A notification is a comment placed by our system in Azure DevOps pull requests. Figure 4.5 shows a screenshot of a comment placed by ConE on an actual pull request. It displays the key elements of a ConE notification: a comment text which provides a brief description of the notification, the id of the conflicting pull request, the name(s) of the author(s) of the conflicting pull request, files that are commonly edited in the pull requests, a provision to provide feedback by resolving or not fixing a comment (marked as "Resolved" in the example), and the option to reply to the comment inline to provide explicit written feedback.

While ConE actively monitors every commit that is being pushed to a pull request, it will only add a second comment on the same pull request again if the state of the active or the reference pull request is significantly changed in subsequent updates and ConE finds a different set of pull requests as candidates for notification.

In a ConE comment, elements like pull request id, file names, author name are actually hyperlinks. The pull request id hyperlink points to the respective pull request's page in Azure DevOps. The file name hyperlink points to a page that shows the diff between the versions of the file in the current and conflicting pull requests. The author name element, upon clicking, spins up a chat window with the author of the conflicting pull request instantly. When people interact with these elements by clicking them, we track those telemetry events (which is consented by the users of the Azure DevOps system, in Microsoft) to better understand the level of interaction developers are having with the ConE system.

4.5.4. Scale

The ConE system has been deployed on 234 repositories in Microsoft. The repositories have been picked based to maximize the diversity and variety of the repositories in various dimensions as explained in Section 4.5.2. Since enabled in March 2020, until September 2020 (when we pulled the telemetry data) ConE evaluated 26,000 pull requests which were created in all the repositories on which ConE has been enabled. Within these 26,000 pull requests, an additional 156,000 update events (commits on the same branch, possibly affecting new files) occurred. Thus, ConE had to react to and process a total of 182,000 events that were generated, within Azure DevOps, in those six months. For every update, ConE has to compare the reference pull request with all active pull requests that match ConE's filtering criteria. In total ConE made a total of approximately two million comparisons.

The scale of operations and processing is expected to grow as we onboard new and large repositories. The simple and lightweight nature of the ConE algorithm combined with the scalable architecture and efficient design, and its engineering on Azure cloud has given us the ability to process events at this scale with a response rate of less than four seconds per event. The time it takes to process an event end to end, i.e., receiving the pull request creation or update event, running the ConE algorithm and

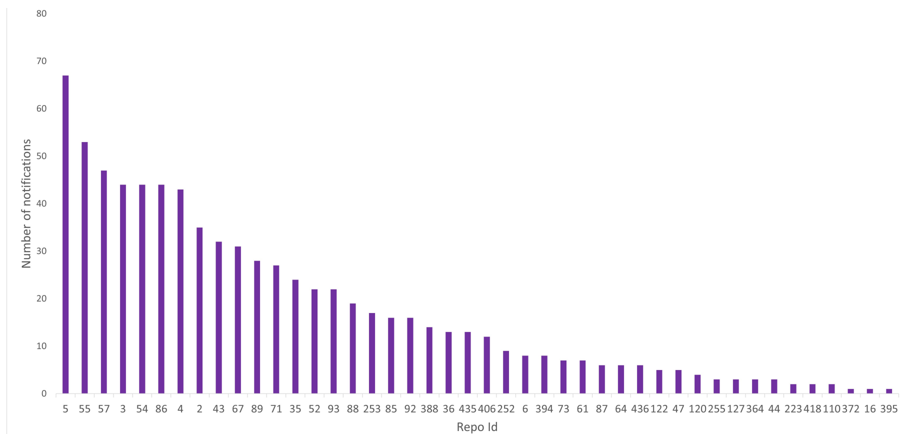


Figure 4.6: Distribution of notifications per repository

passing the recommendations back (if any) has never taken more than four seconds. ConE employed a single service bus queue and four worker roles in Azure to handle the current scale. As per our monitoring and telemetry (resource utilization on Azure infrastructure, processing latency, etc.) ConE still had bandwidth left to serve the next hundred repositories of similar scale with the current infrastructure setup.

4.6. Evaluation: Developers perceptions about ConE's usefulness

Out of the 26,000 pull requests under analysis during ConE's six month deployment (Section 4.5.4), ConE's filtering algorithm (Section 4.4.2) excluded 2,735 pull requests. In the remaining 23,265 pull requests, ConE identified 775 pull requests to send notifications to (3.33%). In this section, we evaluate the usefulness of these 775 notifications.

All repositories were analyzed with the standard configuration; No adjustments were made to the parameters. Though the service is enabled to send notifications in 234 repositories, during the six-month observation period, ConE raised alerts on just 44 distinct repositories. As shown in Figure 4.6, the notification volume varies between repositories.

4.6.1. Comment resolution percentage

ConE offers an option for users to provide explicit feedback on every comment it placed, within their pull requests. Users can select the "Resolved" option if they like or agree with the notification, and the "Won't fix" option if they think it is not useful. A subset of users were given instructions and training on how to use these options.

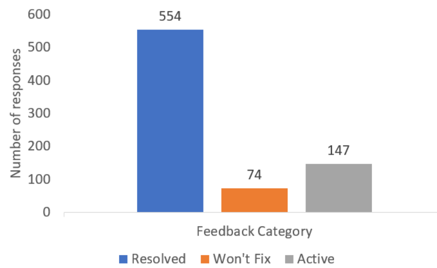


Figure 4.7: Number of positive (Resolved), negative (Won't Fix), and neutral (Active) responses to ConE notifications

The notification itself also contains instructions, as shown in Figure 3.5. A user can choose not to provide any feedback by just leaving the comment as is, in the "Active" state. Through this we collect direct feedback from the users of the ConE system.

Figure 4.7 shows the distribution of the feedback received. The vast majority (554 out of 775 for 71.48 %) of notifications was flagged as "Resolved". For 147 (18.96%) of the notifications, no feedback was provided. Various studies have shown that users tend to provide explicit negative feedback when they do not like or agree with a recommendation, while tend not be so explicit about positive feedback (Liu et al., 2019; Steck, 2011). Therefore, we cautiously interpret this as neutral to positive.

We manually analyzed all 74 (9.5%) cases where the developers provided negative feedback. For the majority of them, the developer was already aware of the other conflicting pull request. In some cases the developers thought that ConE is raising a false alarm as they expect no one else to be making changes to the same files as the ones they are editing. When we show them other overlapping pull requests that were active while they were working on their pull request, to their surprise, the notification were not false alarms. We list some of the anecdotes in subsection 4.6.4.

4.6.2. Extent of interaction

As discussed in Section 4.5.3, a typical ConE notification/comment has multiple elements that a developer can interact with: For each conflicting pull request, the pull request id, files with conflicting changes, and the author name are shown. These are deep links. Developers can just take a look at the comment and ignore it or interact with it by clicking on one of the "clickable elements" in the ConE notification. If the user decides to pursue further clicking on one of these elements, that action is also logged as telemetry (in Azure AppInsights).

From March 2020 to September 2020, we logged 2170 interactions on 775 comments that ConE has placed, which amounts to 2.8 clicks per notification on average. Measured over time, as shown in Figure 4.8, the number of interactions and the "clicks per notification" are clearly increasing as more and more people are getting

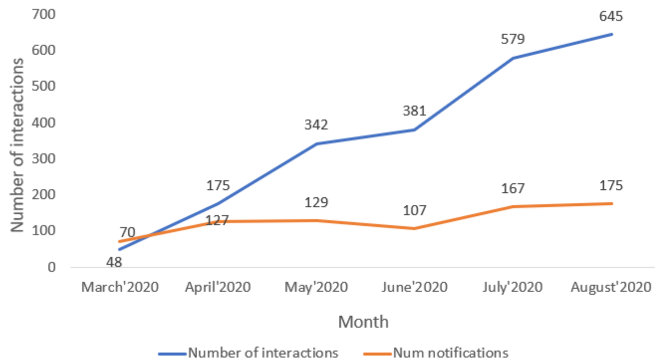


Figure 4.8: Number of notifications (orange), and number of interactions (blue) with those notifications, per month. As developers become more familiar with ConE, they increasingly interact with its notifications

used to ConE comments, and are using it to learn more about conflicting pull requests recommended by ConE.

Note that the extent of interaction does not include additional actions developers can take to contact authors of conflicting pull requests once ConE has made them aware of the conflicts, such as reaching out by phone, walking into each other's office, or a simple talk at the water cooler.

4.6.3. User Interviews

The quantitative feedback discussed so far captures both direct (comment resolution percentage) and indirect (extent of interaction) feedback. To better understand the usefulness we directly reached out (via Microsoft Teams, asynchronously) to authors of 100 randomly selected pull requests for which ConE placed comments. The user feedback for these 100 pull requests is 45% positively resolved, 35% won't fix, and 20% no response. The interviewers did not know these authors, nor had worked with them before, also because the teams working on the systems under study are organizationally far away from the interviewers.

The interview format is semi-structured where users are free to bring up their own ideas and free to express their opinions about the ConE system. We posed the following questions:

1. Is it useful to know about these other PRs that change the same file as yours?

2. If yes, roughly how much effort do you estimate was saved as a result of finding out about the overlapping PRs? If not, is there other information about overlapping PRs that could be useful to you?

3. Does knowing about the overlapping PRs help you to avoid or mitigate a future merge conflict?

Table 4.6: Distribution of qualitative feedback responses

	Category	# of responses
Favorable	I'd love to use ConE	25 (52.08%)
	I will use ConE	20 (41.67%)
Unfavorable	I don't want to use ConE	3 (6.25%)

4. What action (if any) will you likely take now that you know about the overlapping PRs?

5. Would you be interested in keeping using ConE which notifies you about overlapping PRs in the future? (Note that we aim to avoid being too noisy by not alerting if the overlapping files are frequently edited by many people, if they are not source code files, etc.)

We did not receive the responses in a uniform format directly based on the structure of the questions. We used Microsoft Teams to reach out to the developers and the questions are open ended. Therefore, we could not enforce a strict policy on the number of questions the respondents should answer and on the length of the answers. Some of the participants answered all questions, while some answered only one or two. Some respondents were detailed in their response, while some were succinct with 'yes' or ''no' answers. Some of the respondents provided a free-form response, with an average word count of just 47 words per response. So, we could not calculate the distribution of responses for all questions. However, we see that for question-5, there were responses. We coded and categorized the responses we received for question-5 as explained below.

The first three authors, together, grouped the responses that we received (48 out of 100), until consensus was reached, into two categories: Favorable (if the users would like to continue using ConE, i.e., the answer to question 5 is along the lines of 'I will use ConE' or a 'I'd love to use/keep using ConE') and Unfavorable (users do not find the ConE system to be useful and do not want to continue using it.). Table 4.6 shows the distribution of the feedback: 93.75% of the respondents indicated their interest and willingness to use ConE.

4.6.4. Representative Quotes

To offer an impression, we list some typical quotes (positive and negative) that we received from the developers. In one of the pull requests where we sent a ConE notification notifying about potential conflicting changes, a developer said:

> *"I wasn't aware about other 2 conflicting PRs that are notified by ConE. I believe that would be very helpful to have a tool that could provide information about existence of other PRs and let you know if they perform duplicate work or conflicting change!!"*

It turned out that the other two developers (the authors of the conflicting pull requests flagged by ConE) are from entirely different organizations and geographies. Their common denominator is the CEO of the company. It would be very difficult for the author of the reference pull request to know about the existence of the other two pull requests without ConE bringing it to their notice.

Several remarks are clear indicators of the usefulness of the ConE system:

> *"Yes, I would be really interested in a tool that would notify overlapping PRs."*

> *"Looking forward to use it! Very promising!"*

> *"ConE is such a neat tool! Very simple but super effective!"*

> *"ConE is a great tool, looking forward to seeing more recommendations from ConE"*

> *"This is an awesome tool, Thank you so much for working to improve our engineering!"*

> *"It is a nice feature and when altering files that are critical or very complex, it is great to know."*

Some developers mentioned that ConE helped them saving time and/or effort significantly by providing early intervention:

> *"ConE is very useful. It saved at least two hours to resolve the conflicts and smoke again"*

> *"This would save a couple of hours of dev investigation time a month"*

> *"ConE would have saved probably an hour or so for PR <XYZ>"*

We also received feedback from some developers who expressed a feeling that a tool like ConE may not necessarily be useful for their scenarios:

> *"For me no, I generally have context on all other ongoing PRs and work that might cause merge issues. No, thank you!"*

> *"For my team and the repositories that I work in, I don't think the benefit would be that great. I can see where it could be useful in some cases though"*

> *"It's not helpful for my specific change, but don't let that discourage you. I can see how something like ConE be definitely useful for repositories like <XYZ> which has a lot of common code"*

Table 4.7: Distribution of quantitative feedback based on size of the repository

Feedback	Large repositories	Small repositories	Total
Positively resolved	404 (77.69%)	150 (58.82%)	554 (71.48%)
Won't fix	33 (6.34%)	41 (16.08%)	74 (9.54%)
No response	83 (15.96%)	64 (25.10%)	147 (18.96%)
Total	520 (67.09%)	255 (32.90%)	775 (100.0%)

Another interesting case we noticed is, ConE's ability to help in detecting duplication of work. ConE notified a developer (D1) about an active pull request authored by another developer (D2). After the ConE notification was sent to D1, they realized that D2's pull request is already solving the same problem and D2 made more progress. D1 ended up abandoning their pull request and pushed several code changes in D2's pull request, which was eventually completed and merged. When we reached out to D1, they said:

> *"Due to poor communication / project planning D2 and I ended up working on the same work item. Even if I was not notified about this situation, I would have eventually learned about it, but that would have costed me so much time. This is great!"*

Though we do not observe scenarios like this frequently, this case demonstrates an example of the kind of potential conflicts ConE can surface, in addition to flagging syntactic conflicts.

4.6.5. Factors Affecting ConE Appreciation

After analyzing all the responses from our interviews, analyzing the pull requests on which we received 'Won't Fix' and interviewing respective pull request authors, we identified the following main factors as to what makes a developer incline towards using a system like ConE.

Developers who found the ConE notifications useful: These are the developers who typically work on large services with distributed development teams across multiple organizations, geographies and time zones. They also tend to work on core platforms or common infrastructure (as opposed to the ones who make changes to the specific components of the product or service). To corroborate this, the first author classified the repositories into large and small manually, based on the size and the activity volume in those repositories. We then, programmatically, categorized the 628 responses based on their repository sizes. The results, in Table 4.7), show that for large repositories developers are positive for 77.69% (404/520) of the cases, whereas for small repositories this is 58.82% (150/255).

Developers who found ConE not so useful: These developers are the ones who work on small micro services or small scale products and typically work in smaller teams. These developers, and their teams, tend to have delineated responsibilities. They usually have more control over who makes changes to their code base. Interestingly, there were cases where some of these developers were surprised to see another active pull request, created by a different developer, from a different team sometimes, which was editing the same area of the source code as their pull request. This could be a result of underestimating the pace with which service dependencies are introduced, product road maps change, and codebases are re-purposed in large scale organizations.

4.7. Discussion

In this section we describe the outlook and future work. We also explain some of the limitations of the ConE system and how we plan to address them.

4.7.1. Outlook

One of the immediate goals of the ConE system is to expand its reach beyond the initial 234 on which it is enabled, and eventually on every source code repository in Microsoft. Furthermore, in the long run, Microsoft may consider offering ConE as part of its Azure DevOps pipeline, making it available to its customers across the world. Likewise, GitHub may consider to develop a free version of ConE as an extension on the GitHub marketplace for the broader developer community to benefit from this work.

As explained, ConE is expected to generate false alarms because of the fact that it is a heuristics based system.To improve the system and reduce the number of false alarms at this point, ConE checks for very simple but effective heuristics (see Section 4.4.2) and conditions to flag conflicting changes that causes unintended consequences. We offer three configuration parameters (see Section 4.4.3), that help us make the solution effective by striking a suitable balance between the rate of false alarms and coverage, and customize the solution based on individual repository needs.

To further improve precision, we would like to investigate the options that let us go one level deeper from file level to, e.g., analyze actual code diffs. Understanding code diffs and performing semantic analysis on them is a natural next a step for a system like ConE. Providing diff information across every developer branch is fundamentally expensive, so it is not offered by Azure DevOps, the source control system on which ConE is operationalized, nor by other commercial or free source control systems like GitLab or GitHub. A possible remedy is to bring the diff information into the ConE system. This involves checking out two versions of the same file, within ConE, and finding differences. This has to happen in real-time, in a scalable and language agnostic fashion.

Once we have the diff information, another idea is to apply deep learning and code embeddings to develop better contextual understanding of code changes. We can use

the semantic understanding in combination with the historical data about concurrent and non-concurrent edits to develop better prediction models and raise alarms when concurrent edits are problematic.

ConE was found to be useful by facilitating early intervention about the potential conflicting change. However, this does not fully solve the problem i.e., fixing the merge conflicts or merging the duplicate code. Exploring auto-fixing of conflicts or code duplication as a natural extension to ConE's conflict detection algorithm will help in alleviating the problems caused by the conflicts and fixing them in an automated fashion.

4.7.2. Threats to validity

Concerning internal validity, our qualitative analysis was conducted by reaching out to the developers via Microsoft Teams, asynchronously. None of the interviewers know the people that were reached out neither worked with them before. We purposefully avoided deploying ConE on repositories that are under the same organization as any of the researchers involved in this work. As Microsoft is a huge company and most of the users of the ConE service are organizationally distant from the interviewers, the risk of response bias is very minimal. However, there is a small chance that respondents may be positive about the system because they want to make the interviewers, who are from the same company, happy.

Concerning external validity, the empirical analysis, design and deployment, evaluation and feedback collection are done specifically in the context of Microsoft. The correlations we reported in Table 4.2 can vary based on the setting of the organization in which the analysis is performed. As Microsoft is one of the world's largest concentration of developers, and developers at Microsoft uses very diverse set of tools, frameworks, programming languages, our research and the ConE system will have a broader applicability. However, at this point the results are not verified in the context of other organizations.

4.8. Conclusion and future work

In this chapter, we seek to address problems originating from concurrent edits to overlapping files in different pull requests. We start out by exploring the extent of the problem, establishing a statistical relationship between concurrent edits and the need for bug fixes in six active industrial repositories from Microsoft.

Inspired by these findings we set out to design ConE, an approach to detect concurrently edited files in pull requests at scale. It is based on heuristics like the extent of overlap and the presence of rarely concurrently edited files between pairs of pull requests. To make sure the precision of the system is sufficiently high, we deploy various filters and parameters that help in controlling the behavior of the ConE system.

ConE has been deployed on 234 repositories inside Microsoft. During a period of six months, ConE generated 775 notifications, from which 71.48 % received positive feedback. Interviews with 48 developers showed 93% favorable feedback, and

applicability in avoiding merge conflicts as well as duplicate work.

In the future, we anticipate ConE will be employed at substantially more systems within Microsoft. As ConE has been deployed and found to be useful by the developers in a large and diverse (in terms of programming languages used, tools, engineering systems, geographical presence, etc) organization like Microsoft, we believe the techniques and the system has applicability beyond Microsoft. Furthermore, we see opportunities for implementing a ConE service for systems like GitHub or GitLab. Future research surrounding ConE might entail improving its precision by learning from past user feedback or by leveraging diffs without sacrificing scalability. Beyond warnings, future research could also target automating actions to be taken to address the pull request conflicts detected by ConE.

5

Nalanda: A Socio-Technical Graph Platform for Building Software Analytics Tools at Enterprise Scale

Software development is information-dense knowledge work that requires collaboration with other developers and awareness of artifacts such as work items, pull requests, and file changes. With the speed of development increasing, information overload, and information discovery are challenges for people developing and maintaining these systems. Finding information about similar *code changes and* experts *is difficult for software engineers, especially when they work in large software systems or have just recently joined a project. In this chapter, we build a large-scale data platform named Nalanda platform to address the challenges of information overload and discovery. Nalanda contains two subsystems: (1) a large-scale socio-technical graph system, named* Nalanda graph system*, and (2) a large-scale index system, named* Nalanda index system *that aims at satisfying the information needs of software developers.*

To show the versatility of the Nalanda platform, we built two applications: (1) a software analytics application with a news feed named MyNalanda that has Daily Active

*Users (DAU) of 290 and Monthly Active Users (MAU) of 590, and (2) a recommendation system for related work items and pull requests that accomplished similar tasks (*artifact recommendation*) and a recommendation system for subject matter experts (*expert recommendation*), augmented by the Nalanda socio-technical graph. Initial studies of the two applications found that developers and engineering managers are favorable toward continued use of the news feed application for information discovery. The studies also found that developers agreed that a system like Nalanda artifact and expert recommendation application could reduce the time spent and the number of places needed to visit to find information.*

5.1. INTRODUCTION

Building software is a highly collaborative process that requires awareness of the activities by many different stakeholders and interaction with many different artifacts such as files, pull requests, and work items. At the same time, large-scale software development creates lots of data about how people work with each other and with software artifacts. As a consequence, finding information can be hard, especially when software engineers work on large software projects with thousands of files and team members. A lot of times this knowledge about software development activity and expertise is hidden in the form of software development process data and the interaction map between stakeholders and artifacts. This data is hard to mine and represent in a form that allows practitioners to build applications on top of this data. This is primarily due to the scale at which this data is generated and the fact that this data is scattered across disparate data sources and systems. Therefore, it is hard to take full advantage of this data and extract hidden knowledge, without employing a plethora of tailored tools, customized for each source control system and the software development environment.

Socio-technical data that captures social and technical aspects of software development (Sarma, Maccherone, Wagstrom, & Herbsleb, 2009) is often captured in graph structures. For example, the Hipikat tool builds a project memory from past activities to support newcomers with software modification tasks (Cubranic et al., 2005). In 2010, the Codebook framework was introduced with a focus on discovering and exploiting relationships in software organizations to support inter-team coordination (Begel et al., 2010). Codebook provided a graph and a query language to support a wide range of applications: find the most relevant engineers, find out why a recent change was made, and general awareness of engineering activity (Begel & Zimmermann, 2010). Codebook was built for a single team with 420 developers only.

With the advent of cloud services, the scale at which software development happens and the volume of data generated during the software development process increased significantly (Ma et al., 2019; Potvin & Levenberg, 2016). To address the challenges that come with scale, in this chapter, we present a large-scale software analytics data platform named Nalanda[1] which is built on top of the software development activity data and the artifacts. Nalanda builds a socio-technical graph at

enterprise scale, with thousands of repositories. Additionally, the Nalanda index system, helps with the search and recommendation of software development artifacts and the experts. Nalanda stores its graph in a native graph database and optimizes heavily to query complex relationships so that software analytics applications can operate directly on the graph via cloud services and a high degree of performance.

The Nalanda platform, which is a generic and *enterprise scale* software analytics data platform consists of two subsystems: the Nalanda graph system, which provides a *large scale* socio-technical graph of software data, and the Nalanda index system, which is an *enterprise scale* index system that can be used to support a wide range of software engineering tasks such as recommendation and search. Nalanda scales to enterprise-scale data from 6,500 repositories. The socio-technical graph has 37,410,706 nodes and 128,745,590 edges. The index system contains 8,079,748 documents.

To show the *versatility* of the Nalanda platform, we describe two tools that have been built on top of the Nalanda platform and *deployed* at Microsoft: A software analytics news feed application built on top of the Nalanda graph system named MyNalanda, and a novel recommendation system (Nalanda artifact and expert recommendation application) leveraging the socio-technical graph for ranking the recommendations.

The goal of this chapter is to describe the design, implementation, and deployment of the Nalanda graph system, the index system, and two successful applications (MyNalanda and the Nalanda artifact and expert recommendation application) built and deployed at Microsoft. We also share details about the extensive analyses and user studies that we conducted to evaluate the perceived usefulness of MyNalanda and Nalanda artifact and expert recommendation application from our deployments at Microsoft. Additionally, we share insights from building the Nalanda platform, MyNalanda, and the Nalanda artifact and expert recommendation application.

To that end, we explain the construction of the Nalanda graph system in Section 5.2 and the index system in Section 5.3. We explain the applications built leveraging these two systems i.e., MyNalanda in Section 5.4 and the Nalanda artifact and expert recommendation application in Section 5.5.

5.2. BUILDING THE NALANDA GRAPH

Key challenges in the construction of the Nalanda Graph are scale and consistency. In this section, we lay out what content we store in the Nalanda graph, from which sources we collect the data, and how we ensure that the graph is kept up to date and consistent as hundreds of thousands of events from thousands of repositories arrive on a daily basis.

[1]Nalanda is named after an ancient university and knowledge center located in India. It is famous for its huge corpus of scriptures, books, and knowledge repositories.

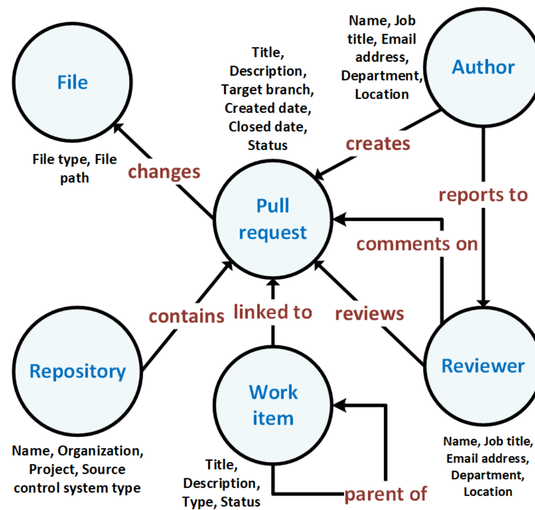


Figure 5.1: Nalanda's Graph Schema

5.2.1. Nalanda's Graph Schema

The schema of the Nalanda graph is shown in Figure 5.1. The nodes represent the actors or entities involved in the software development life cycle, while the edges represent the relationships that exist between them.

Each node in the Nalanda graph has a type associated with it and attributes specific to that node type, as listed in Figure 5.1. The central node is the Pull Request, which has incoming edges from Author, Reviewer, Work item, and Repository nodes, and has a outgoing edges to File nodes changed by the pull request. A developer takes the role of an author when they make source code changes and submit pull requests and they assume the role of a reviewer when they perform code reviews. These are represented as user nodes in the Nalanda graph with different edge types but are listed here as two different nodes in Figure 5.1 for clarity. For File nodes, different types are distinguished, including source code, configuration, and project files. Files are edited by the authors via pull requests. Files are represented as nodes in the Nalanda graph with a second-order relationship established between the user and file nodes via a Pull request node.

Edges in the Nalanda graph represent the relationships between various actors and entities. Like nodes, edges can be of different types and can have properties associated with them. An edge is created between an author node and a pull request node when a developer creates a pull request. Similarly, an edge is established between the reviewer and the pull request nodes when a developer is assigned a code review. A linked to edge is created when developers link a pull request to a work item, commonly done in Azure DevOps ("Azure DevOps", 2021) to connect earlier, related,

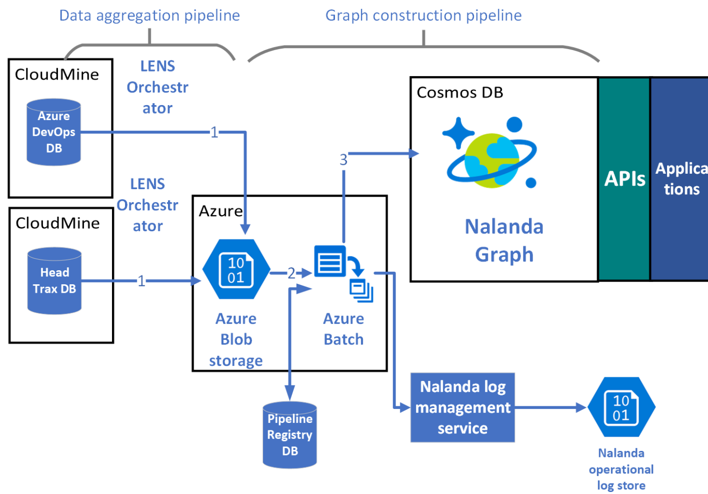


Figure 5.2: Nalanda's data collection and graph construction architecture

pull requests to new issues. Likewise, a parent of edge is created between two work items if they are linked by the developers with a parent-child relationship in Azure DevOps. Finally, a reports to edge is created between two user nodes if one of them is the reporting manager of the other.

5.2.2. Data Collection and Graph Construction

The Nalanda platform architecture is shown in Figure 5.2. The primary source of data for the Nalanda graph is Azure DevOps. Instead of directly crawling the Azure DevOps system for data, we leverage an intermediate data source called CloudMine (Czerwonka et al., 2013). The Nalanda platform takes the raw event data from CloudMine and processes it to create the nodes and edges of the Nalanda graph. The graph can be queried using the APIs we provide, or directly by means of the graph query language Gremlin ("Gremlin query language", 2021).

The platform builds upon Azure ("Azure services", 2021): key services used include Azure batch, Azure CosmosDB, Azure SQL Server, Azure Blob Storage, and Lens explorer ("Lens Explorer", 2021). As shown in Figure 5.2, the Nalanda platform is built using two independently operated pipelines: a data aggregation pipeline and a graph construction pipeline, as explained below.

Nalanda's Data Aggregation Pipeline

As indicated as step 1 in Figure 5.2, the data aggregation pipeline is responsible for fetching data from different data sources (most notably CloudMine) and making it available for the graph construction pipeline to process.

We use Lens Orchestrator ("Lens Explorer", 2021) for orchestration and schedul-

ing purposes. Lens has the ability to connect to multiple data sources and systems and move data around. In the aggregator pipeline, Lens first connects to the CloudMine data store (which is hosted on Cosmos ("Azure services", 2021)) and executes Scope scripts to gather data from various data streams, such as the pull request stream, the work item stream, the code review stream, etc. Lens saves the aggregated data in the form of CSV files in Cosmos. Then, Lens connects to Azure Blob Storage to temporarily store these CSV files for further processing. This intermediate store is required as CloudMine does not allow any other service (except Lens) to connect to and access the data files for security and compliance reasons.

Additionally, we use the Lens job scheduling utilities to configure a job in Lens to run once every eight hours to pull the latest data from CloudMine and save it to the Azure Blob Storage.

Nalanda's Graph Construction Pipeline

Once the data is available in Azure Blob Storage, we process it using an Azure batch job to construct the Nalanda graph (Steps 2 and 3 in Figure 5.2). We use Azure CosmosDB as our graph data store.

When the batch job discovers new data files as generated by the aggregation pipeline, it updates the pipeline registry with new file information. This includes file names, size, timestamp, whether the file contains data from the bootstrap or the incremental stream, file processing status, processing duration, etc. The pipeline registry is a SQL database whose purpose is to serve as a transactional store for the graph construction pipeline. We create one row in the pipeline registry database for every file discovered. After the new data is downloaded, for each data file the corresponding node and edges are added to, deleted from, or updated in the Nalanda graph. Once all data for a file is read, its registry status is set to "completed".

The graph construction pipeline operates in two modes: bootstrap mode and incremental mode.

Bootstrap mode. This mode helps ingest all of a repository's data, from repository creation time until when it is run. Typically, we run this mode for a repository when it is being onboarded onto the Nalanda graph platform for the first time.

Incremental mode. This mode helps keep the data in the Nalanda graph updated without needing to read the massive original streams of CloudMine (whose size is in the orders of hundreds of Terabytes), using the incremental streams offered instead (with sizes in the order of tens of Gigabytes). We run the incremental pipeline once every eight hours.

These separate modes offer the flexibility to bootstrap any new or existing repository data in an independent and asynchronous manner. Furthermore, the separation of bootstrap and incremental pipelines offers substantial performance improvement in terms of run time and resource utilization, as illustrated in Table 5.1.

Refreshing the data by querying the original streams of CloudMine each time the pipeline is run, takes 28 hours for 6,500 repositories. As the incremental streams are substantially smaller, each incremental job finishes in 20 minutes, yielding an

Table 5.1: Comparison of pipeline run time and number of records with the increase in number of repositories

Mode	Run time		# Records to process	
	350 repos	6500 repos	350 repos	6500 repos
Bootstrap	9 hrs	28 hrs	1.05M	7.41M
Incremental	10 min	20 min	10K	57K

improvement of 98.8% in pipeline run time, for each pipeline run. Note that for an increase of the number of repositories by a factor of 20, the run time for the bootstrap mode was increased by a factor of 3 only. This is an effect of the careful design and implementation of the data pipeline by massively parallelizing the data processing code and enabling distributed processing on multiple Azure batch nodes.

5.2.3. Data consistency and self-healing

The Nalanda graph platform is a distributed system that works with multiple external data sources and large-scale data processing systems, which are prone to introduce data inconsistencies. Data gaps can manifest due to various factors related to infrastructure and availability of the CloudMine crawlers.

Detecting and remediating data gaps in such a massive distributed system is not a trivial task. We devised a novel self-healing system that detects data gaps and consistency issues proactively and performs self-healing. This helps the pipeline to guarantee data consistency irrespective of the failures manifested in external data sources such as CloudMine. The self-healing system uses the pipeline registry to monitor pipeline states and can switch from incremental to bootstrap mode if this is warranted.

When an incremental pipeline is run, the timestamp of the oldest record to be processed is compared with the timestamp of the last successful pipeline run. If the difference between these timestamps is bigger than three days (the period for which incremental streams hold their data), this means a data gap has occurred. To address this, the bootstrap mode is triggered for the repositories involved, and ongoing incremental pipelines are halted. Furthermore, the pipeline registry is updated to indicate that bootstrapping is taking place, thereby locking new incremental jobs.

5.2.4. Scale

The Nalanda graph has been designed to accommodate thousands of repositories. At the time of writing, it holds the software development activity data from 6,500 repositories at Microsoft. We ingest data starting from January 1, 2019, or from more recent repositories when their first pull request is created.

To keep its graph up to date, the Nalanda platform processes 500,000 events per day. These events include new pull requests, updates or commits on those pull re-

Table 5.2: Nalanda node and edge types and their prevalence

Node type	Count
file	14,537,998
text	12,104,427
pull request	7,568,949
work item	3,067,754
user	131,578
repository	6,500
Total nodes	**37,410,706**

Edge type	Count
changes	65,706,621
reviews	39,447,635
creates	7,569,086
contains	7,337,036
linked to	7,094,597
parent of	843,728
comments on	746,887
Total edges	**128,745,590**

quests, pull request state changes, code review assignments, and code review comments. At the time of writing, the Nalanda graph contains 37 million nodes and 128 million edges as detailed in Table 5.2.

5.3. Indexing Nalanda for Information Retrieval

Many nodes in the Nalanda Graph contain text. To facilitate *search* over such text at the Nalanda scale, we need to create appropriate indexes. The actual indices needed may depend on the specific applications built on top of the Nalanda graph. In this section, we discuss the indices we create and how we ensure they remain up to date at scale.

For every search, we use the BM25 algorithm (Robertson et al., 1992) for determining text similarity between a query and documents (pull requests, work items, ...) and ranking the results. BM25 is a bag-of-words model developed based on the probabilistic retrieval framework ("The Probabilistic Relevance Framework: BM25 and Beyond", 2009). For a given query Q containing keywords q_1,...,q_n, the BM25 score for a document D is:

$$\text{score}(D,Q) = \sum_{i=1}^{n} \text{IDF}(q_i) \cdot \frac{f(q_i,D) \cdot (k_1+1)}{f(q_i,D) + k_1 \cdot (1 - b + b \cdot \frac{|D|}{\text{avgdl}})} \tag{5.1}$$

where $f(q_i,D)$ is q_i's term frequency in the document D, IDF(q_i) is q_i's inverse document frequency, $|D|$ is the length of the document D in words, and avgdl is the average document length in the text collection from which documents are drawn. k_1 and b are free parameters. We use standard recommended values ($b = 0.75$, $k_1 = 1.2$) for these constants (Schütze et al., 2008).

5.3.1. The Nalanda Artifact Index

The Nalanda artifact indexfacilitates search through pull requests and work items. We index the metadata, titles, and descriptions of the artifacts (pull requests and work

items). The metadata consists of elementary properties, namely project name, repository name, and organization name.

The corpus used by the Nalanda artifact index comprises two types of documents: pull requests and work items. We index the metadata, titles, and descriptions of the artifacts (pull requests and work items). The metadata consists of elementary properties, namely project name, repository name, and organization name.

The Nalanda artifact indexcan be used to find relevant pull requests given a work item, feature, technical, or functional concept. Furthermore, with this index completed pull requests and work items can be used as a template and inspiration to solve similar problems. They provide code samples, expose code review comments, and help as informal documentation to learn the best practices. Additionally, they help understand the team or project-specific processes involved in getting such pull requests completed.

5.3.2. The Nalanda Expert Index

The Nalanda expert index is built to map subject matter experts (SMEs) to technical and functional skills. Experts can be of two types: functional and technical. Functional experts have expertise in specific functionality of a software product or service, such as the query optimizer in an RDBMS product or the ranker in a search engine product. Technical experts have expertise with a technology concept such as socket programming in Java.

The Nalanda expert index relies on a collaborative software development platform like Azure DevOps to mine and associate expertise with people. We match pull requests and work items (as also used for the artifact index) to their authors and contributors. The process of building the Nalanda expert index consists of the following steps:

1. Find all pull requests and work items completed by a developer and extract key phrases from them.

2. Create a document using the data from step 1, which is the representation of a developer's skills.

3. Repeat steps 1-2 for every developer and builds the corpus for the expert index

Similar to the Nalanda artifact index, we use the BM25 algorithm for querying the expert index corpus. The intuition is that if a developer makes frequent code changes related to a topic (functional or technical), they must be knowledgeable in that topic area. We represent the frequency of a topic in a developer's activity as term frequency in the BM25 index. Therefore, the more a topic appears in the document corpus constructed for that developer, the more weight that topic is given.

We leverage the Nalanda socio-technical graph for re-ranking the search results returned by the artifact and expert indices. A detailed analysis of the impact of employing the Nalanda graph in refining the search results is discussed in Section 5.5.3.

5.3.3. Scale

Building indices such as the artifact and expert index at scale is an expensive operation. We carefully crafted the system design to make the Nalanda index creation and refresh pipelines robust and tolerant to failures (details about implementation are explained in Section 5.5.2). The Nalanda search system has been built as a cloud-native service. This enables us to scale out the system horizontally with the increase in data and query volume. This also helps us in meeting high uptime Service Level Agreements (SLAs) requirements to move to production. Currently, the artifact index contains 8,018,320 documents and the expert index contains 61,428 documents.

We ingest data from 6,500 repositories. Our index data refresh pipeline, which runs once every week, completes in 65 minutes on average. The graph data refresh pipeline, which runs every 8 hours and finishes in 18 minutes. We optimized the API service to return the response in 1.7 seconds in accordance to the SLA requirements.

5.4. Nalanda Applications (I): The MyNalanda Portal

The Nalanda graph and indexing platform can be used to build many applications to support software development teams and organizations in their daily work. The first application we discuss is MyNalanda, an online news feed in production at Microsoft, in which developers and managers alike can monitor ongoing software development activities.

5.4.1. MyNalanda Motivation

The motivation behind MyNalanda is that it is common practice for developers to work on multiple work items or pull requests at once. It is also common practice for developers at Microsoft to work on multiple source code repositories simultaneously. Microsoft does not have many large mono-repositories, but a lot of small or medium sized repositories. Keeping track of one's work items in their repository or across multiple repositories is a difficult and time consuming task. Moreover, these tools operate in a workitem-centric fashion, i.e., the primary goal of the tool is to search for and find a work item one is interested in.

By contrast, MyNalanda is a *developer-centric* news feed. Upon login, MyNalanda shows the activity (pull request, work item, code review) of a developer, from multiple repositories, in their homepage. Additionally, MyNalanda enables developers to discover what their teammates and other collaborators are working on without the hassle of going to different Azure DevOps repositories.

5.4.2. The MyNalanda Homepage

For a user, the central hub in MyNalanda is their *homepage*. An example of such a MyNalanda homepage is shown in Figure 5.3. Its information is organized in the following sections:

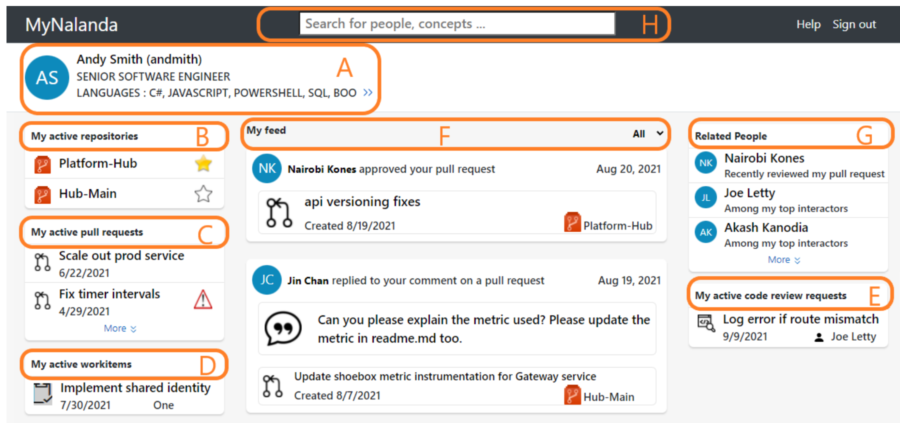


Figure 5.3: The MyNalanda homepage

News feed: The centralized news feed ('F' in Figure 5.3) is located in the middle of the page. The news feed shows events such as updates in pull requests, code review comments, and pull request status changes from all the repositories a developer works in. For managers, the news feed provides updates from their reports' activity.

User details: This section ('A' in Figure 5.3) provides details about developers, such as name, email address, job title, and their expertise (extracted from their software development activity data). This helps in facilitating easy discovery of developers' skills and their current projects.

Active items: There are separate sections for active repositories, pull requests, work items, and code review requests ('B'-'E' in Figure 5.3). Users of MyNalanda can prioritize the discovery of updates from these items by *following* or *unfollowing* them.

Related people: This section ('G' in Figure 5.3) visualizes who a developer collaborates with and how local software development communities are formed. A developer's collaborators include others who work together with the developer on a coding task or work item, or who are either being reviewed by that developer, or who are involved in reviewing a developers' pull request.

Search box: The search box ('H' in Figure 5.3) can be used to find developers and discover their activity. It also can help with searching for technical and functional concepts by leveraging the Nalanda artifact and expert indices (as explained in Section 5.3).

All elements in MyNalanda, such as pull requests, work items, people, and repositories have embedded URLs which take them to the corresponding item in Azure DevOps. This makes it easy for developers to navigate between MyNalanda and Azure DevOps.

Additionally, MyNalanda facilitates integration of other machine learning recom-

menders due to its extensible architecture. For example, overdue pull request are indicated with a subtle warning icon in the active pull requests section. This is powered by the Nudge machine learning models (C. S. Maddila et al., 2021).

5.4.3. MyNalanda Usage

MyNalanda leverages the graph representation of the data (the Nalanda graph) and its schema design to navigate efficiently through complex relationships and find the content presented in various sections. As a result, the MyNalanda homepage including the news feed and the other sections loads in less than a second. As quoted by one of the MyNalanda users *"It is simple, blazing fast to load, adapts to screen size"*.

Based on organic growth alone, MyNalanda reached 290 Daily Active Users (DAU), and 590 MonthlyActive Users (MAU), in the first six months of deployment of the Beta version at Microsoft.

5.4.4. MyNalanda Evaluation: Perceived Usefulness

To evaluate how developers and engineering managers perceive the usefulness of MyNalanda, we follow a mixed method research design involving interviews and surveys. With increasing sources of development-related information available, there remains an open question about how they want to access and integrate information about their own development activities and the development work done by their peers, and if current platforms are adequate. Through this evaluation we assess how MyNalanda matches the corresponding information needs.

Evaluation Setup

Semi-structured Interviews. We conducted interviews to investigate information discovery and overload, and if users might use an interface like MyNalanda. Participants included five developers and two engineering managers; seven participants were men and zero were women. Semi-structured interviews were conducted remotely, and ranged from 30-45 minutes. Interview topics included interest in accessing information about their own and peers' development activity, information overload and how they typically get information (in both in-office and work-from-home contexts). We then showed a deployed version of MyNalanda and asked for reactions including if they would like it, what information they would find useful, and where they would want to see it. If the interviewee was an engineering manager, they were also asked what information they would be interested in seeing related to their team's work.

Immediately following the interviews, notes were taken by the interviewer to augment the transcription and interviews were coded for emergent themes. Following each subsequent interview, themes were revisited to see if any codes should be combined or separated. Once no new themes emerged (i.e., theoretical saturation), we concluded our interview phase. After seven interviews, themes remained consistent. Finally, we reviewed notable excerpts from all interviews and organized the themes by topic.

Surveys. Following interviews, we conducted surveys to validate and quantify the themes that emerged. Survey participants included full-time employees who were developers or engineering managers. While their work spanned various areas across Microsoft, all had contributed to at least one of the 6,500 repositories included in the Nalanda Graph, which allowed participants to access their MyNalanda homepage during the survey.

We designed our survey based on themes that emerged during our interviews, resulting in a 19-item instrument that took a median of 8.5 minutes to complete. Participants were shown their MyNalanda newsfeed (with their own development activity) and given the survey. Topics included demographics, usefulness of information included in their MyNalanda feed, information pain points and privacy concerns, and current and anticipated work location (e.g., office or work-from-home). We included items asking about usefulness of MyNalanda information, and preferences for possible features (based on jobs-to-be-done). We asked where respondents would like to see MyNalanda integrated (if at all), and their comfort level in sharing their development activity through something like MyNalanda. The full survey instrument is available online at research.microsoft.com (C. Maddila et al., 2021).

The survey was sent to 2,000 people in total (1,400 developers and 600 engineering managers) with 144 responses (92 developers and 44 engineering managers), resulting in an 8% response rate after considering the 10% out of office responses. Our low response rate could be due to the fact that the survey is not a trivial one to fill in (cognitively and time it takes to complete the survey) and/or because we did not offer any incentives for participation. Of those respondents, 92 (67.65%) were developers (including software development engineers, senior software engineers, etc.) and 44 (32.35%) were engineering managers (including software engineering manager, software engineering lead, etc.). Our respondents included five women (6.17%), 73 men (90.12%), one who preferred to self-describe (1.32%), and two who preferred not to answer (2.63%). They reported an average of 10.15 years working at Microsoft, ranging from 0.6 to 29.9 years (standard deviation 7.13).

Results of the study

We first present findings from our semi structured interviews, with each noted as developer (D) or engineering manager (EM). We then present our survey results.

Interviews. Participants expressed two themes related to information needs: integration and overload. Information integration was echoed by many interviewees; we define this as having development-related information for self and others integrated into a single, easy-to-access interface. P1 (EM) and P2 (D) discussed easily *linking* design docs and associated artifacts like pull requests. P1 (EM) discussed the usefulness of graphic summaries for their teams' development activity across time periods; this reflects consolidation via visualization. P2 (D) called out that the ability to see detailed information about peers' work is helpful when coordinating work, and is not readily available.

Table 5.3: MyNalanda survey Feedback

Feature	Cumulative (n=85)	Developers (n=63)	Engineering Managers (n=22)
Active pull requests (C)	55.3%	55.6%	54.5%
Active code review requests (E)	48.2%	49.2%	45.5%
Active repositories (B)	44.7%	46.0%	40.9%
Active work items (D)	34.1%	34.9%	31.8%
Feed (F)	28.2%	34.9%	9.1%
User Details (A)	24.7%	27%	18.2%
Related people (G)	20%	23.8%	9.1%

P5 (D) spoke about viewing the information in different ways to deal with information overload and ensure they were not missing things: using a 'Most recent' view for chronologically-ordered information, an algorithmic 'Relevance' view to combine information across teams, and team-only view to further filter. This speaks to strategies they use to ensure keep up on development-related information both within and across teams they collaborate with. Similar sentiments were expressed by other interviewees.

Surveys. Based on themes from our interviews, we developed survey items to evaluate information integration and overload. We asked participants to rate how useful each feature of MyNalanda was (including development activity, the news feed, and collaborators) on a five-point Likert-type scale ranging from 1= Not at all useful to 5= Extremely useful. All items were optional; of those who took the survey, 85 answered questions about MyNalanda features (63 developers and 22 engineering managers). This allowed us to capture participant reactions to a real integrated information platform instead of a hypothetical one.

Table 5.3 lists the accumulated percentages of "Extremely useful" (Likert=5) and "Very useful" (Likert = 4) for each feature. Here, we see that *Active pull requests* (55.6% and 54.5% among developers and engineering managers, respectively) and *Active code review requests* (49.2% and 45.5% among developers and engineering managers, respectively) are the highest rated features, with *User details* (27.0% and 38.2% among developers and engineering managers, respectively) and *Related People* (23.8% and 9.1% among developers and engineering managers, respectively) rated the lowest.

Based on this, we conclude that the active items (pull requests, code review requests, repositories, and work items) are the most valued features of MyNalanda, and that user details are primarily of interest to engineering managers.

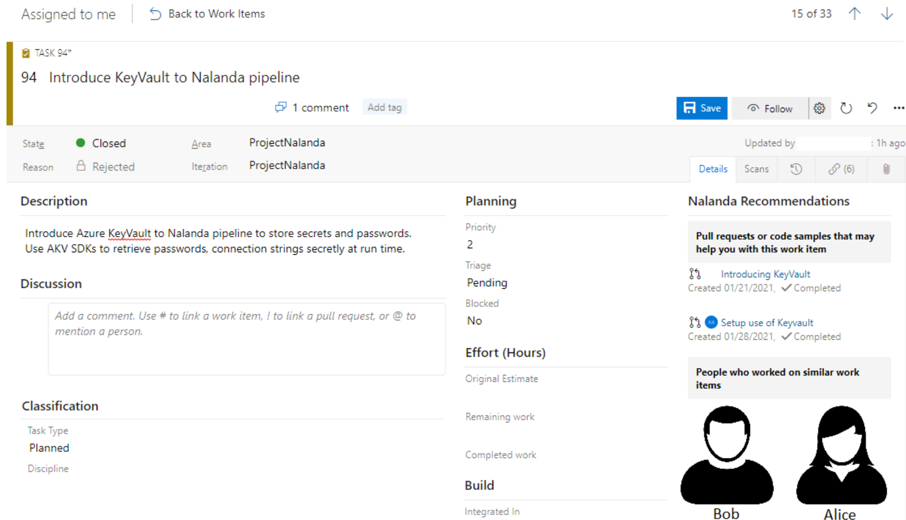


Figure 5.4: Nalanda recommendations in Azure DevOps

5.5. Nalanda Applications (II): Artifact and Expert Recommender

When a developer is working on a work item or feature, finding the relevant pull requests and code samples is one of the biggest pain points for developers (C. Maddila et al., 2021). The intensity of the problem multiplies in large organizations with teams working on multiple source code repositories. Similarly, finding functional and technical experts in large organizations is a difficult task. A lot of times, this involves the developers to mine git history, going through wiki pages, design documents, etc.

The Nalanda artifact and expert recommendation application implements a recommendation plug-in for Azure DevOps (AzDO) ("Azure DevOps", 2021). When a new work item is assigned to a developer, the plug-in triggers an API call to the Nalanda search API. The client passes the necessary input parameters, such as the search query (work item title and description), the work item owner, and the repository metadata. Upon receiving the results (artifact and expert recommendations), the client add recommendations to the work item page (in Azure DevOps). A sample work item recommendation page from live deployment (Beta version) is shown in Figure 5.4. Besides, through its easy-to-use APIs, the Nalanda artifact and expert recommendation application system powers other applications such as the *search box* in MyNalanda (as explained in Section 5.4.2).

When a work item or an issue is assigned to a developer, the Nalanda artifact and expert recommendation application uses a combination of title and description of the assigned work item as an input search query and provides recommendations about

work items or pull requests that accomplished similar tasks. Additionally, the system also provides a list of subject matter experts whom a developer can reach out to seek help while working on that work item.

5.5.1. The Nalanda Ranking Algorithm

To construct the Nalanda Expert and Artifact recommenders, we devised a ranking algorithm consisting of three steps: 1) querying the artifact index (see Section 5.3.1) to get the relevant pull requests and work items, 2) querying the expert index (see Section 5.3.2) to get a list of relevant experts, and 3) re-ranking the results using the Nalanda graph. To that end, we take the following steps:

Step 1: We first construct a query as a combination of the title and description of the work item. Then, we tokenize the query using heuristics that we built for the software engineering domain, such as splitting strings into camel-cased or pascal-cased tokens. We also create n-gram based tokens since we found that bi-grams and tri-grams, such as `ImapTransfer` and `MailboxSyncEngine`, capture important information. Next, we filter out stop words (Luhn, 1960).

We employ the BM25 algorithm, which takes care of prioritizing important tokens using Inverse Document Frequency (IDF) scores. It assigns more weight to the documents with a higher overlap with the search query tokens (term frequency), and calculates a relevance score as shown in Equation 5.1.

Step 2: We query the expert index, which contains one document per person. These documents contain the tokens mined from a developer's pull requests and work items history. We perform the same pre-processing explained in Step 1 on the query. The BM25 index returns a ranked list of experts.

Step 3: A heuristics-based filtering scheme is used to filter out results with relevance scores below a threshold. We determine the threshold empirically to set it at the 75th percentile of the relevance score distribution. We determined this value based on a series of experiments to optimize the accuracy of the results while reducing the size of the results set that is passed to the next step.

Step 4: We use the Nalanda socio-technical graph to assign proximity scores based on the edge distance between the person performing the search and the results returned by the BM25 index (obtained from Step-3). The proximity score is the length of the shortest path between two nodes. We use the proximity score to re-rank the results. For example, the proximity score is 1 if a pull request, work item, or people node is 1-edge away from a developer node, in their shortest path.

Step 5: We then pick the top-k results from the results set and return them to the user. Items that have a higher BM25 relevance score and are in close proximity to a developer are ranked higher. Furthermore, we use proximity score to break the tie between results where relevance scores are the same.

5.5.2. Implementation

We implemented the Nalanda search system on the Azure platform, with an emphasis on scalability to thousands of repositories. The underlying architecture is visualized

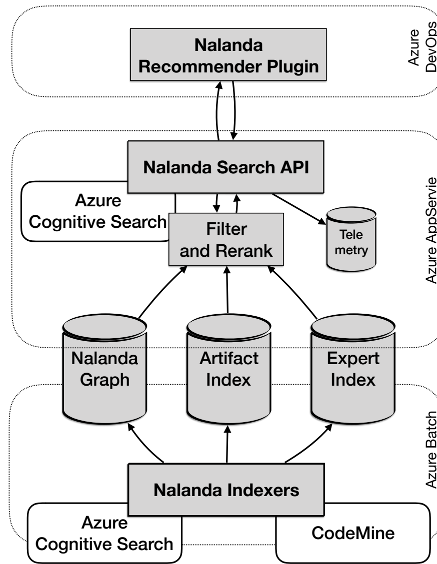


Figure 5.5: Nalanda artifact and expert recommendation application architecture

in Figure 5.5.

Nalanda Indexers

The backbone of the recommendation system is the batch jobs creating and continuously updating the indices and the socio-technical graph, as discussed in Section 5.3 and shown at the bottom of Figure 5.5. We leverage CodeMine (Czerwonka et al., 2013) to help us with aggregating source control system data from thousands of repositories.

We rely on Azure Cognitive Search (ACS) ("Azure Cognitive Search", 2021), which provides BM25 as a service, to store and access the indices. This helps us in alleviating the problems associated with service maintenance, uptime, and scale-out.

Search API

Given the indices and the socio-technical graph, the Nalanda Search API (shown in the middle of Figure 5.5) implements the search algorithm from Section 5.5.1. Each search query (typically initiated by the users in MyNalanda or through Azure DevOps) passed to the Nalanda search API is processed in real-time. The mean response time for the API call is 1.7 seconds.

The Nalanda API service asynchronously saves telemetry to an Azure SQL database without impacting the query performance. The telemetry includes the search query, user metadata, search results, click logs, and the API response time. We use this

Table 5.4: Evaluation Data Summary

	Artifact Recommendation	Expert Recommendation
Index data	Pull request title and description, socio-technical graph	Pull request title and description, socio-technical graph
Test set	Work Item title and description	Internal StackOverflow questions
Ground truth	Pull requests linked to the work item	People who answered the post and people who have answered at least five other questions with the same tag as the post
Data set	80K randomly sampled work items from the 6,500 repositories	10K randomly sampled questions and answers from StackOverflow

information to evaluate engagement and to improve the performance of the Nalanda search system and the API service.

5.5.3. Quantitative Evaluation

To understand the efficacy and usefulness of the Nalanda artifact and expert recommendation application, we conduct a large-scale offline evaluation and a user study.

Experiment setup

We randomly sample 80,000 work items from the 6,500 repositories such that there are at least 10 work items selected from each repository. Subsequently, we use the title and description of each of these work items as the input to the Nalanda search API. We expect the right pull requests and people to be returned from the search API.

Since a recommendation system like ours does not exist in the company, we do not have a ground truth to conduct a large-scale evaluation. Therefore, we rely on the pull requests manually tagged by developers to the work items in Azure DevOps to build the evaluation dataset. To create the ground truth dataset for the expert recommendations, we leverage the private instance of StackOverflow deployed at Microsoft. Details about the index, ground truth, and test sets used for these experiments are shown in Table 5.4.

Results

We use two commonly used metrics for recommender systems: 1. Top K accuracy, which measures the number of times the correct item is found in the top K recommendations 2. Mean Reciprocal Rank (MRR), which calculates the reciprocal of the rank at which the first relevant document was retrieved (Craswell, 2009).

Table 5.5 shows the results from the evaluation for different values of K. We can see that incorporating more attributes of the pull request, such as its title and description, improves both the MRR and accuracy considerably. Furthermore, re-ranking the results using the Nalanda graph also substantially improves the recommendations.

Table 5.5: Evaluation and Comparative study for Artifact Recommendation

Indexed Properties	K = 3		K = 5		K = 10	
	Accuracy	**MRR**	**Accuracy**	**MRR**	**Accuracy**	**MRR**
pull request metadata	0.26	0.23	0.29	0.28	0.32	0.30
pull request attributes						
+ pull request title	0.38	0.36	0.43	0.41	0.51	0.48
+ pull request description	0.49	0.47	0.53	0.51	0.60	0.59
+ socio-technical graph	**0.71**	**0.71**	**0.74**	**0.73**	**0.78**	**0.77**

Table 5.6: Evaluation and Comparative study for Expert Recommendation

Indexed Properties	K = 3		K = 5		K = 10	
	Accuracy	**MRR**	**Accuracy**	**MRR**	**Accuracy**	**MRR**
pull request metadata	0.35	0.30	0.39	0.33	0.43	0.38
pull request attributes						
+ pull request title	0.51	0.46	0.54	0.49	0.59	0.53
+ pull request description	0.60	0.54	0.64	0.59	0.69	0.61
+ socio-technical graph	**0.63**	**0.60**	**0.69**	**0.63**	**0.75**	**0.67**

Similar improvements can be noticed for the expert recommendation task too (Table 5.6).

5.5.4. User Perception

We conducted a user study among developers regarding the usefulness of the recommendations. We selected ten participants (identified as P1–P10) from Microsoft to evaluate the recommendations on their recently completed work items.

Participants and protocol

We conducted semi-structured interviews, which were conducted remotely, and ranged from 15-30 minutes. The average experience of the subjects is 7.7 years in the company and ranged from 10 months to 21 years.

We employ a one-group pretest-posttest pre-experimental design (Campbell & Stanley, 1963). We used the Likert scale (Robinson, 2014) for rating the responses. The respondents can provide their responses on a 1 to 5 scale, ranging from 'strongly disagree' to 'strongly agree'. We posed them the questions listed below.

When you are working on a work item, how useful would the recommendations be in completing the work item (on a scale of 1 to 5):

1. You are going to refer to the work items and pull requests recommended as inspiration and informal documentation on accomplishing the work item.

2. You would likely reach out to the recommended people for consultation on accomplishing the work item.

Results

Do the users feel work item, pull request, and expert recommendations are useful? The participants expressed that the Nalanda recommendation system can be a great value addition to the software development process. 60% of the participants

Figure 5.6: Participants' evaluation of the Nalanda system

rated 'agree' or 'strongly agree' when asked whether they find the artifact recommendations useful and 40% responded favorably to the expert recommendations.

Through question 2 we measure the difference between the expectations the participants had of a hypothetical recommendation system with the Nalanda artifact and expert recommendation application. This question measures the dependent variable for the user study (introduction of the Nalanda system). In Figure 5.6, the radar chart shows some differences between the participants' original expectations and their perception of the Nalanda recommendations.

These differences can be observed better in Figure 5.7, in which the averages of the rating are shown. The difference in expectation versus perception was more apparent with the expert recommendations compared to the artifact recommendations.

To offer an impression, we list some typical quotes (positive and negative) that we received from the developers.

> *"A tool like this will help greatly to understand the processes involved in pushing my changes through."*

> *"It is great to see the recommendations about people to talk to. My team is large, mostly remote, and I am new. So this is very helpful."*

> *"Finding people to talk to has never been a problem for me as I have been working in the same org for a while."*

5.6. Discussion

5.6.1. Outlook

In the future, we anticipate both the Nalanda graph system and MyNalanda to be scaled out significantly inside Microsoft in terms of the number of repositories and users. Furthermore, we see opportunities for implementing the Nalanda graph platform as a service on top of the open-source software development data mined from systems like GitHub.

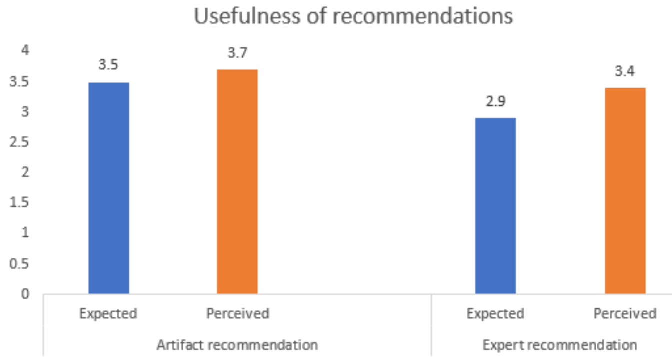


Figure 5.7: Expectations and perceptions of the Nalanda system

We also expect the Nalanda artifact and expert recommendation application to be employed at substantially more software development systems at Microsoft. Similar to MyNalanda we see opportunities to implement the Nalanda artifact and expert recommendation application in open source systems such as GitHub. Future research could entail including other types of useful recommendations such as internal and external documentation, tutorials, and recommendations from question-answer forums.

The rich socio-technical data in the Nalanda graph in combination with advances in deep learning and Graph Convolutional Networks (GCNs), hold promise for applications such as neural reviewer recommendations by leveraging the socio-technical structures. The Nalanda graph system lays the foundation to bring various techniques from the graph representation world ("Graph Neural Network Techniques", 2021), such as link prediction, social network analysis, etc. into the software engineering and analytics domain.

5.6.2. Threats and Limitations

Internal Validity

Conducting trustworthy experiments on data collected from thousands of repositories is challenging especially due to the problems of avoiding data leakage and obtaining credible ground truth. In our experiments, we addressed this (see Table 5.4). While our results are highly promising and an important first step, more experiments are needed to better understand the true nature of the graph's contribution to expert and artifact recommendation.

The risk of response bias is minimal in all our studies because all the participants of the user study are organizationally distant from the people involved in building this system. However, there remains a small chance that people in the user study may be positive about the system because they want to make the developers who are from the same company motivated and happy.

External Validity

The context of our evaluation and user studies is a software development company with a large number of developers. These developers work on a portfolio of products across many contexts and domains. By conducting a user study within one single company, we were able to control for factors like culture, tooling, frameworks, and programming languages. However, our results may not be generalizable across all developers in all contexts. Hence, our results are not verified in the context of other organizations or the open-source community. Therefore, our findings may be limited and warrant further research. Future work could investigate user interfaces, integrating our findings with design guidelines that span usability and technical and organizational complexity (Jaferian et al., 2008).

5.7. Related work

Graph Representations Hipikat (Cubranic et al., 2005) is one of the earliest works to build a graph for software development entities like tasks, file versions, and documents, using a fixed schema. It was built to onboard new hires quickly by providing easy access to relevant artifacts. Codebook (Begel et al., 2010) builds a prototype graph consisting of various software development entities. The data was mined from software repositories for a single team at Microsoft. Bhattacharya et al. (Bhattacharya et al., 2012) use graph representation of source code and bug tracking information to construct predictors for software engineering metrics like bug severity and maintenance efforts. Other applications of graph representations of software artifacts include visualizing relationships among project entities (Sarma, Maccherone, Wagstrom, & Herbsleb, 2009) and extracting changes in variability models (Dintzner et al., 2018). Compared to all this work, the scale of the Nalanda graph is significantly larger with 37M nodes and 128M edges, and Nalanda has been designed to be actively used in a production setting.

Source Control Dashboards GitHub offers a dashboard for developers on its homepage ("GitHub personal dashboard", 2021). It displays the repositories, active pull requests, and issues (work items). A key limitation of the GitHub dashboard is that one cannot see the activity of another developer. In contrast, MyNalanda helps find the complete development activity of any developer. Another difference is the "Team feed" in MyNalanda, which displays the development activity of all the developers in a team. This helps the discovery of the items worked by other team members.

Information Needs Ko et al. (Ko et al., 2007) studied information needs in colocated development teams. Fritz and Murphy (Fritz & Murphy, 2010) provide a list of questions developers ask for the most frequently sought-after information within a project. Information needs have also been studied in the context of change tasks (Sillito et al., 2008), inter-team coordination (Begel et al., 2010) and software analytics (Begel & Zimmermann, 2014; Buse & Zimmermann, 2012; Huijgens et al., 2020) Through its applications, Nalanda can efficiently address most information needs re-

lated to people, code, and work items. For example, the answer to questions such as "Who is working on what" and "What are coworkers working on right now" is easily available in the MyNalanda application.

Artifact Recommendation systems for software developers Recommendation systems for software engineering aim at assisting developers with activities such as code reusability, writing effective bug reports, etc. (M. Robillard et al., 2010). Tools like CodeBroker (Ye & Fischer, 2002) help in finding the relevant code samples extracted from the standard Java documentation generated by Javadoc from Java source programs and deliver the suggestion to the Emacs editor. Anvik et al. (Anvik et al., 2006) proposed a semi-automated method to assign bug reports to reporters based on their expertise using a machine learning algorithm. Mockus and Herbsleb(Mockus & Herbsleb, 2002) used quantity as a measure of expertise. The tool is presented in the form of HTML pages with search engine and hierarchical navigation. Fu et al.(Fu et al., 2017) used the node2vec algorithm to convert file entities within projects into knowledge mappings. They proposed four features to capture the social relationships between developers. Devrec(X. Zhang et al., 2017), a developer recommendation system, mines development activities of developers in GitHub and StackOverflow to recommend collaborators for a given project. Hammad et al (Hammad et al., 2020) use keywords from the textual content of commits. Each expert is associated with the extracted keywords for a recommendation. On the other hand, Canfora et al (Canfora et al., 2012) use mailing lists and versioning systems to recommend experts for newcomers joining a software project. Compared to these approaches, the Nalanda artifact and expert recommendation application is designed to be highly scalable and provide responses in real time.

5.8. Conclusion

In this chapter, we seek to build a *large scale* software analytics data platform named Nalanda with two subsystems (the Nalanda graph systemand the Nalanda index system). The Nalanda graph system consists of a socio-technical graph encompassing the entities, people, and relationships involved in the software development life cycle. The Nalanda index system is an *enterprise scale* index system that can be used to support a wide range of software engineering tasks such as recommendation, and search.

We built the Nalanda graph system using software development activity data from 6,500 source code repositories. The graph consists of 37,410,706 nodes and 128,745,590 edges. To the best of our knowledge, it is the largest socio-technical graph built to date using private software development data. Similarly, the Nalanda index system contains 8,018,320 documents in its artifact index and 61,428 documents in its expert index with data ingested from 6,500 repositories at Microsoft.

The Nalanda platform and its applications (MyNalanda and Nalanda artifact and expert recommendation application) help in developing awareness of each other's work, and building connections between developers across repositories, while offer-

ing mechanisms to discover information while managing information overload. We also seek to address the problems of information discovery by finding related work items and experts for software developers.

Based on organic growth alone, MyNalanda has Daily Active Users (DAU) of 290 and Monthly Active Users (MAU) of 590. A preliminary user study shows that 74% of developers and engineering managers surveyed are favorable toward continued use of MyNalanda for information discovery. The Nalanda artifact and expert recommendation application, with the help of the socio-technical graph for customization, lifted the accuracy of artifact recommendations by 30.45 percentage points to 0.78. In a study with ten professional software developers, participants agreed that a system like Nalanda artifact and expert recommendation application could reduce the time spent and the number of places needed to visit to find information.

In the future, we anticipate both the Nalanda graph system and MyNalanda to be scaled out significantly inside Microsoft in terms of the number of repositories and users. We believe the systems and the techniques have applicability beyond Microsoft. Furthermore, we see opportunities for implementing the Nalanda graph platform as a service on top of the open-source data mined from platforms like GitHub.

6

CONCLUSION

The goal of this dissertation is to help make the DevOps lifecycle efficient by developing recommenders that aid various stakeholders in the DevOps lifecycle. To that end, we develop recommenders such as ORCA, Nudge, ConE, MyNalanda, and Nalanda Recommendations and propose methods to measure the impact of such recommenders in terms of efficiency gains. We also propose a general-purpose unified framework that helps in paving the path to develop more such recommenders by significantly reusing various functionalities or modules of the recommender system development, such as data collection, training, inference, recommendation, and evaluation.

6.1. CONTRIBUTIONS

In this dissertation, we make the following main contributions:

In Chapter 2, we present our work on automated differential bug localization, 'Orca: Differential Bug Localization in Large-Scale Services.' This results in:

- An algorithm, implemented in the tool ORCA, that helps developers and operations staff quickly root causing the incidents or bugs that caused outages in an automated fashion.
- Evidence that light-weight IR-based (Information Retrieval) bug localization techniques can be applied at scale without compromising on performance
- Evidence that our general-purpose and unified data framework is powerful in building such applications at scale by bringing the data from the left (code authoring or inner loop) and right (deployment or outer loop) side data from the DevOps lifecycle into a single place

In Chapter 3, we present our work based on 'Predicting pull request completion time: a case study on large scale cloud services and 'Nudge: Accelerating Overdue Pull Requests Towards Completion.' This results in:

- A new modality of effort estimation, i.e. performing effort estimation at pull request level
- A deeper understanding of the factors that impact the change progression
- An algorithm to programmatically detect the dependent actors and engage the right change blockers in the nudge notifications
- A framework for detecting activity in a pull request by consuming various signals and actions generated in the pull request environment
- Evidence that predicting the pull request completion time is a possibility and operationalizing such machine learning models at a large scale can be done
- Evidence that systems such as Nudge, which are built using the pull request level effort estimation prediction models, are accurate with their predictions
- Demonstrating the ability of the impact of Nudge on development practices and change velocity through mixed-methods approaches

In Chapter 4, we presented 'ConE: A Concurrent Edit Detection Tool for Large Scale Software Development'. This results in:

- Evidence that changes localized to specific files have a high probability of causing merge conflicts in large software code repositories
- Empirical analysis that explains the characteristics of changes that has a potential to cause merge conflicts when merged with the main branch
- Statistical evidence that a model like ConE can accurately predict the occurrence of such merge conflicts
- Evidence that developers are inclined towards using tools like ConE and consuming the recommendations made to avoid future merge conflicts, which is a major waste of their time
- Mixed methods approach to help understand the impact and reception of tools like ConE

In Chapter 5, we presented 'Nalanda: A Socio-Technical Graph Platform for Building Software Analytics Tools at Enterprise Scale.' This results in:

- A general-purpose graph data platform that captures the interactions between various artifacts and developers involved in the software development process

- A Socio-Technical graph that consists of the knowledge of expertise, ownership, and social interactions between code authors, pull requests, code reviewers, source code files, etc.
- MyNalanda, which is a news feed for discovering individual and team activity of software developers and helps in dealing with the problems of information overload and information discovery
- A system for recommending the related tasks and subject matter experts (SMEs) for software developers leveraging the socio-technical graph
- Large scale studies (survey and interviews) that aim at understanding the reception of software developers towards such applications and recommender systems

6.2. Reflection on the Research Questions

In light of our findings, in this section, we answer the research questions defined in Section 1.2.2.

6.2.1. Research Question 1: What are the modalities of recommenders that need to be built to support modern DevOps?

Our studies with hundreds of software developers have shown that problems associated with information discovery are real in large software development environments. For example, a task such as finding the commit that caused an incident involves digging through the incessant amounts of log files, querying different data sources, and talking to different people. The intensity of the problem goes up with other influencing factors such as the severity of the incident, the complex dependency chain of microservices, etc. Similarly, there exist issues with pushing stale pull requests towards completion. Finding a simple piece of information such as the change blocker (author or reviewer of the pull request) requires an engineering manager or a project manager to go through the history of the pull request and review comments.

During code authoring, software developers need to refer to many documents, such as wiki pages, or design docs. It is also common for some teams to follow a more informal documentation culture and the knowledge is confined to few experts. In that case, finding who can help with a task is not trivial. It is a herculean task for a developer to find tasks or pull request that solved similar problems for inspiration or use them as a guide. The intensity of these problems grows with developers moving across teams or companies.

By conducting a deeper analysis of the needs of various stakeholders, the scenarios, and the problem space, we developed a solid understanding of the problems associated with information needs of software developers. With that background, we developed a series of recommenders to help alleviate the problems mentioned above.

To address the incident root causing problem, we designed a novel search technique that we call *differential bug localization*. Using descriptions of the bug as a query, we detect changes to the abstract syntax tree in the source code and search only these changes for text-based similarity. Similarly we designed Nudge, a machine learning service for accelerating overdue pull requests towards completion. As its name suggests, Nudge sends a reminder if a pull request is overdue. To help with developing awareness about potential conflicting changes, we developed ConE. It sends recommendations to developers involved in a potential merge conflict.

To address the problems associated with information needs, we build a large-scale data platform named Nalanda to address the challenges of information overload and discovery. Nalanda uses a socio-technical graph to provide context-sensitive recommendations about similar tasks, pull requests, subject matter experts.

Our research has shown that designing and deploying recommenders such as ORCA, Nudge, CONE in large-scale DevOps environments is practical and useful. One of the contributing factors for the positive reception of these tools is that they shield the developers from the technology aspects and methods. For example, we employed techniques from fields such as information retrieval, machine learning, software engineering, program analysis while designing these recommenders. However, the audience of the recommendations need not be aware of the complexity of methods and implementation details. We focused a lot on user experience, presentation, latency and scalability using these factors as guiding principles while conducting our research. The methods we devised to satisfy these requirements such as IR-based bug localization, pull request level effort estimation, and light-weight merge conflict prediction are the results of such thinking.

6.2.2. Research Question 2: How to assess the efficacy of smart recommenders and their impact on DevOps processes?

Gauging the impact of a recommender is extremely important, but can be challenging. For example, to measure the accuracy of an auto root causing service, we need a ground truth dataset of root causes. This sometimes is a big culture change for teams who are not used to documenting root causes before closing incidents. Similarly, to understand the impact of a reminder service such as Nudge, we need to come up with careful strategies that eliminate the effect of other confounding factors. Complexity only goes up with adding factors such as geographically distributed teams, complex dependency chains of services, or disparate development practices followed by teams in the same company.

While this is challenging, it is possible to devise methods and inculcate practices (sometimes, cultural changes) to perform evaluation of such recommenders. However, being intentional about developing methods for conducting evaluation is an important consideration. For example, conducting A/B testing or performing telemetry analysis cannot be done unless we build them when we operationalize the recom-

menders. At the same time, a one-size-fits-all approach will not work for evaluating such a diverse set of recommenders. Though there exist general evaluation components, such as user studies or surveys, we came up with methods and techniques to evaluate each recommender in their own specific.

We use mixed-methods approaches to evaluate the impact of our recommenders on software development practices. We use the implicit feedback mechanism to quantitatively evaluate the efficacy at scale. We also conduct randomized trial experiments to evaluate the impact of a certain recommender while leaving all the confounding factors untouched. While through quantitative methods we can understand what happens in a software system, understanding why that happens is often not possible by only observing the data. To obtain richer data to answer this kind of questions, we use interviews, user studies, focus groups, etc.

Being intentional and focused on evaluation is beneficial in many ways. It helped us assess the merit of our methods, in a practical setting, and iteratively improve important metrics such as accuracy, precision, recall, etc. It also helped us with gauging the user sentiment, perceived usefulness, and helped us in devising methods that are relevant. For example, when evaluating ORCA, we used quantitative methods to measure the efficacy of ORCA. We obtained an accuracy of 70%, for top-10 results, in the beginning. Later, we found that developers find our service not that useful because the majority of the correct root causes were listed at or after the fifth position. we then started tracking another metric named MRR and worked towards bringing it to 0.44 (a root cause is guaranteed to be found within top-3 results). The way we achieved this is fundamentally changing our algorithm to include a build provenance graph while performing the search. This is an example of how evaluation under practical settings mixed with early user feedback has driven the research methods and outcomes.

With the same spirit and focus on evaluation mind, we devised various methods for evaluating other recommenders such as Nudge, CONE, Nalanda recommendations. For Nudge, we collect data from 147 repositories. We noticed that the average lifetime of the pull requests on which Nudge notifications were sent was 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). Similarly, for Nalanda artifact and expert recommendation application, the accuracy of artifact recommendations was 78%. We also employ methods like a one-group pretest-posttest experimental design where 60% of the participants rated 'agree' or 'strongly agree' when asked whether they find the artifact recommendations useful.

We conclude that there is no prescribed method to measure and improve the recommendation accuracy. As explained, we take several standard methods and customize them as per the need of the hour. We employ the relevant methods based on the modality of the recommender, the feasibility to conduct large-scale experiments and studies, and the aspects of the recommender we are interested in gaining a deeper understanding of. We believe this diligent selection of methods and tailor-making them as per the need is efficient and inspires future research, and help practitioners

and researchers developing such recommenders and tools.

6.2.3. Research Question 3: How to build a general-purpose and scalable data framework for developing smart DevOps recommenders?

When developing the recommenders, we realized that the infrastructure used by many of these recommenders have patterns in common such as ingesting source control data (commit, pull request histories), extracting attributes from them, learning rules, training models, passing recommendations, gathering telemetry and feedback. We wanted to investigate whether building a data platform that is generic and scalable is possible. Generalizability is not something that becomes important only in the implementation phase. We have to be intentional about it while conducting our research and devising methods. For example, the attributes we use for developing the CONE model are a subset of what we use for developing the Nudge machine learning model. The rules we learn and save when running the Nudge model are leveraged in the MyNalanda application for raising alerts for overdue pull requests.

Our decision to make the data platform generalizable significantly accelerated the development and deployment of our recommenders. We reused a lot of components across recommenders. For example, when we first built ORCA, we built a scalable infrastructure that ingests pull request and commits data from the source control systems, normalizes the data, and materializes that into a relational database system. We also built an API layer for the client services to consume the recommenders. When building Nudge, we reused the data platform but with some customizations to collect the extra data we need to train the machine learning models.

While building the Nalanda socio-technical graph and the artifact, expert indices we used the same design principles. Generally, building large-scale socio-technical graphs, with millions of nodes and edges, is a non-trivial activity, which needs a lot of careful design and implementation. Because of the generic nature of the data platform we built for ORCA, Nudge, and CONE, we were able to spin up the graph platform in just 3 months of time.

Similarly, the framework and the API layer we built for passing recommendations for recommenders such as ORCA and Nudge are reused for ConE, and Nalanda recommendations. We created a general-purpose evaluation framework and controlled experimentation platform for Nudge, which was then used by ConE, MyNalanda, and the Nalanda recommendation systems.

In our experience, the rigor, impact and iteration speed of methods can be increased greatly with reusing the components as much as possible. This also involves being intentional about the generalizability of methods, and evaluation frameworks. Designing the data framework and applications (recommenders) goes hand-in-hand and one influences how the other is conceived or built. This also has an auxiliary side effect of rapid prototyping and helps with early feedback loops. When conducting research and proposing methods, dealing with infrastructure and data issues

is sometimes considered the least interesting part. Doing it the right way helps us with focusing on the research and methods while keeping the issues involved with infrastructure away.

6.3. Concluding Remarks

In this dissertation, we show that DevOps is a complex and collaborative process. DevOps involves deploying large-scale systems, possibly in cloud environments, continuous delivery of new features and functionality, many people working together, and feedback flowing from development to operations (and vice versa) in a loop.

Because of the nature and the complexity involved in the DevOps processes, the success of DevOps processes requires employing smart tools or recommenders, which are either proactive or reactive in nature. These recommenders pass useful recommendations to mitigate, alleviate, and streamline the DevOps processes. However, building such smart recommenders is not a trivial task due to the huge volumes of data, disparate and siloed nature of the data sources, the complexity involved in operationalizing such recommenders at scale, the nonexistence of methods to evaluate the impact of such recommenders, and the socio-technical and human aspects involved in the perception and success of such recommenders.

Online root-causing tools such as ORCA has shown significant promise and set a direction toward developing lightweight bug localization techniques. In the future, we anticipate more research happening in the direction of employing lightweight techniques for bug localization. Dealing with performance bugs and configuration-related issues could be a natural next step. Apart from that, we also anticipate the development of more semantic-level program analysis techniques within Orca.

Similarly, recommenders like Nudge have shown that pull request level effort estimation models can be built and operationalized at scale. We observed a significant reduction in completion time when Nudge is employed, by over 60% on average, for pull requests which were *nudged*. Recommenders like ConE, to detect concurrently edited files in pull requests at scale, are proven to be effective in developing awareness and preventing accidents. During a period of six months, ConE generated 775 notifications, from which 71.48 % received positive feedback. Interviews with 48 developers showed 93% favorable feedback and applicability in avoiding merge conflicts as well as duplicate work.

We believe that Nudge-like and ConE-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, and improving the precision of recommenders like ConE by learning from past user feedback or by leveraging diffs without sacrificing scalability.

Dealing with information overload and satisfying the information needs of developers in the ever-changing and rapid world of DevOps is a perennial problem. The Nalanda platform and its applications (MyNalanda and the Nalanda artifact and expert

recommendation application) help in developing awareness of each other's work, and building connections between developers across repositories, while offering mechanisms to discover information while managing information overload. We also seek to address the problems of information discovery by finding related work items and experts for software developers. In the future, we anticipate both the Nalanda graph system and MyNalanda to be scaled out in many DevOps environments (corporations and open source systems alike). We believe the systems and the techniques have applicability beyond a single company. Furthermore, we see opportunities for implementing the Nalanda graph platform as a service using open source data from, e.g., GitHub.

Finally, the unification of the left and right sides of the data (from code authoring, and code reviewing to incident management) enabled applications like ORCA, which otherwise are impossible to materialize. Similarly, a large part of our success can be attributed to the generic and scalable nature of the data platform and the design principles we followed. This approach significantly accelerated the development and deployment of our recommenders. We reused a lot of components across recommenders. This enabled us to not worry about the infrastructure once it was built and deployed. We were able to focus on the scenarios, the algorithms, and machine learning models, devise methods to evaluate the impact of our recommenders, and finally make the large-scale DevOps processes efficient.

Bibliography

Accioly, P., Borba, P., & Cavalcanti, G. (2018). Understanding semi-structured merge conflict characteristics in open-source java projects. *Empirical Softw. Engg.*, *23*(4), 2051–2085. https://doi.org/10.1007/s10664-017-9586-1

Agrawal, H., & Horgan, J. R. (1990). Dynamic program slicing. *ACM SIGPlan Notices*, *25*(6), 246–256.

Ahmed, I., Brindescu, C., Mannan, U. A., Jensen, C., & Sarma, A. (2017). An empirical examination of the relationship between code smells and merge conflicts. *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 58–67. https://doi.org/10.1109/ESEM.2017.12

Ankolekar, A., Sycara, K., Herbsleb, J., Kraut, R., & Welty, C. (2006). Supporting online problem-solving communities with the semantic web. *Proceedings of the 15th International Conference on World Wide Web*, 575–584. https://doi.org/10.1145/1135777.1135862

Anvik, J., Hiew, L., & Murphy, G. (2006). Who should fix this bug? *28th International Conference on Software Engineering (ICSE)*, *2006*, 361–370. https://doi.org/10.1145/1134336

Ashok, B., Joy, J., Liang, H., Rajamani, S. K., Srinivasa, G., & Vangala, V. (2009). Debugadvisor: A recommender system for debugging. *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, 373–382. https://doi.org/10.1145/1595696.1595766

Ashraf, U., Mayr-Dorn, C., & Egyed, A. (2019). Mining cross-task artifact dependencies from developer interactions. *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 186–196. https://doi.org/10.1109/SANER.2019.8667990

Asthana, S., Kumar, R., Bhagwan, R., Bird, C., Bansal, C., Maddila, C., Mehta, S., & Ashok, B. (2019). Whodo: Automating reviewer suggestions at scale. *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 937–945.

Attariyan, M., Chow, M., & Flinn, J. (2012). X-ray: Automating root-cause diagnosis of performance anomalies in production software. *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, 307–320. http://dl.acm.org/citation.cfm?id=2387880.2387910

Attarzadeh, I., Mehranzadeh, A., & Barati, A. (2012). Proposing an enhanced artificial neural network prediction model to improve the accuracy in software effort estimation. *2012 Fourth International Conference on Computational Intelligence, Communication Systems and Networks*, 167–172.

Azure Batch. (2020). https://azure.microsoft.com/en-us/services/batch/

Azure Cognitive Search. (2021). https://azure.microsoft.com/en-us/services/search/

Azure DevOps. (2021). https://azure.microsoft.com/en-us/services/devops/

Azure DevOps REST API. (2021). https://docs.microsoft.com/en-us/rest/api/azure/devops/

Azure services. (2021). https://azure.microsoft.com/en-us/services/

Bansal, C., Renganathan, S., Asudani, A., Midy, O., & Janakiraman, M. (2020). Decaf: Diagnosing and triaging performance issues in large-scale cloud services. *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice*, 201–210.

Bass, L., Weber, I., & Zhu, L. (2015). *Devops: A software architect's perspective*. Addison-Wesley Professional.

Baysal, O., Kononenko, O., Holmes, R., & Godfrey, M. W. (2013). The influence of non-technical factors on code review. *2013 20th working conference on reverse engineering (WCRE)*, 122–131.

Begel, A., Phang, K. Y., & Zimmermann, T. (2010). Codebook: Discovering and exploiting relationships in software repositories. *Proceedings of the ACM/IEEE 32nd International Conference on Software Engineering*.

Begel, A., & Zimmermann, T. (2010). Keeping up with your friends: Function foo, library bar. dll, and work item 24. *Proceedings of the 1st Workshop on Web 2.0 for Software Engineering*, 20–23.

Begel, A., & Zimmermann, T. (2014). Analyze this! 145 questions for data scientists in software engineering. *Proceedings of the 36th International Conference on Software Engineering*, 12–23.

Berczuk, S., & Appleton, B. (2002). *Software configuration management patterns: Effective teamwork, practical integration*.

Bettenburg, N., Nagappan, M., & Hassan, A. E. (2015). Towards improving statistical modeling of software engineering data: Think locally, act globally! *Empirical Softw. Engg.*, *20*(2), 294–335. https://doi.org/10.1007/s10664-013-9292-6

Bhagwan, R., Kumar, R., Maddila, C. S., & Philip, A. A. (2018). Orca: Differential bug localization in large-scale services. *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI18)*, 493–509.

Bhattacharya, P., Iliofotou, M., Neamtiu, I., & Faloutsos, M. (2012). Graph-based analysis and prediction for software evolution. *2012 34th International Conference on Software Engineering (ICSE)*, 419–429. https://doi.org/10.1109/ICSE.2012.6227173

Biehl, J. T., Czerwinski, M., Smith, G., & Robertson, G. G. (2007). Fastdash: A visual dashboard for fostering awareness in software teams. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 1313–1322. https://doi.org/10.1145/1240624.1240823

Bird, C., Rigby, P. C., Barr, E. T., Hamilton, D. J., German, D. M., & Devanbu, P. (2009). The promises and perils of mining git. *2009 6th IEEE International Working Conference on Mining Software Repositories*, 1–10. https://doi.org/10.1109/MSR.2009.5069475

Bird, C., & Zimmermann, T. (2012). Assessing the value of branches with what-if analysis. *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. https://doi.org/10.1145/2393596.2393648

Bird, D. L., & Munoz, C. U. (1983). Automatic generation of random self-checking test cases. *IBM systems journal*, *22*(3), 229–245.

Boehm, B., Clark, B., Horowitz, E., Westland, J., Madachy, R., & Selby, R. (1995). Cost models for future software life cycle processes: Cocomo 2.0. *Annals of Software Engineering*, *1*, 57–94. https://doi.org/10.1007/BF02249046

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

Bowring, J., Orso, A., & Harrold, M. J. (2002). Monitoring deployed software using software tomography. *ACM SIGSOFT Software Engineering Notes*, *28*(1), 2–9.

Briand, L. C., El Emam, K., Surmann, D., Wieczorek, I., & Maxwell, K. D. (1999). An assessment and comparison of common software cost estimation modeling techniques. *Proceedings of the 21st International Conference on Software Engineering*, 313–322. https://doi.org/10.1145/302405.302647

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

Brindescu, C., Ahmed, I., Jensen, C., & Sarma, A. (2019). An empirical investigation into merge conflicts and their effect on software quality. *Empirical Software Engineering*, *25*. https://doi.org/10.1007/s10664-019-09735-4

Brindescu, C., Ahmed, I., Leano, R., & Sarma, A. (2020). Planning for untangling: Predicting the difficulty of merge conflicts, 801–811. https://doi.org/10.1145/3377811.3380344

Brun, Y., Holmes, R., Ernst, M., & Notkin, D. (2011). Proactive detection of collaboration conflicts. *SIGSOFT/FSE 2011 - Proceedings of the 19th ACM SIGSOFT Symposium on Foundations of Software Engineering*, 168–178. https://doi.org/10.1145/2025113.2025139

Brun, Y., Holmes, R., Ernst, M. D., & Notkin, D. (2011a). Proactive detection of collaboration conflicts. *Proceedings of the 19th ACM SIGSOFT Symposium*

and the 13th European Conference on Foundations of Software Engineering, 168–178. https://doi.org/10.1145/2025113.2025139

Brun, Y., Holmes, R., Ernst, M. D., & Notkin, D. (2011b). Proactive detection of collaboration conflicts. *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, 168–178. https://doi.org/10.1145/2025113.2025139

Brun, Y., Holmes, R., Ernst, M. D., & Notkin, D. (2013). Early detection of collaboration conflicts and risks. *IEEE Trans. Softw. Eng.*, *39*(10), 1358–1375. https://doi.org/10.1109/TSE.2013.28

Buse, R. P., & Zimmermann, T. (2012). Information needs for software development analytics. *2012 34th International Conference on Software Engineering (ICSE)*, 987–996.

Calikli, G., A. Uzundag, B., & Bener, A. (2010). Confirmation bias in software development and testing: An analysis of the effects of company size, experience and reasoning skills. In R. Yates & F. Fagerholm (Eds.), *Proceedings workshop on psychology of programming interest group (ppig)*. PPIG.

Campbell, D. T., & Stanley, J. C. (1963). Experimental and quasi-experi-mental designs for research.

Canfora, G., Di Penta, M., Oliveto, R., & Panichella, S. (2012). Who is going to mentor newcomers in open source projects? *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. https://doi.org/10.1145/2393596.2393647

Carlson, J. L. (2013). *Redis in action*. Manning Publications Co.

Cataldo, M., Wagstrom, P. A., Herbsleb, J. D., & Carley, K. M. (2006). Identification of coordination requirements: Implications for the design of collaboration and awareness tools. *Proceedings of the 2006 20th Anniversary Conference on Computer Supported Cooperative Work*, 353–362. https://doi.org/10.1145/1180875.1180929

Chilimbi, T. M., Liblit, B., Mehra, K., Nori, A. V., & Vaswani, K. (2009). Holmes: Effective statistical debugging via efficient path profiling. *Proceedings of the 31st International Conference on Software Engineering*, 34–44. https://doi.org/10.1109/ICSE.2009.5070506

Chow, M., Meisner, D., Flinn, J., Peek, D., & Wenisch, T. F. (2014). The mystery machine: End-to-end performance analysis of large-scale internet services. *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, 217–231. http://dl.acm.org/citation.cfm?id=2685048.2685066

Chulani, S., Boehm, B., & Steece, B. (1999). Bayesian analysis of empirical software engineering cost models. *IEEE Trans. Softw. Eng.*, *25*(4), 573–583. https://doi.org/10.1109/32.799958

Clarke, E. M., Grumberg, O., & Peled, D. (1999). *Model checking*. MIT press.

Cleve, H., & Zeller, A. (2005). Locating causes of program failures. *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*, 342–351.

Colavita, F. (2016). Devops movement of enterprise agile breakdown silos, create collaboration, increase quality, and application speed. *Proceedings of 4th International Conference in Software Engineering for Defence Applications*, 203–213.

Costa, C., Figueiredo, J., Murta, L., & Sarma, A. (2016). Tipmerge: Recommending experts for integrating changes across branches. *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 523–534. https://doi.org/10.1145/2950290.2950339

Costa, C., Figueiredo, J., Ghiotto lima de Menezes, G., & Murta, L. (2014). Characterizing the problem of developers' assignment for merging branches. *International Journal of Software Engineering and Knowledge Engineering*, *24*, 1489–1508. https://doi.org/10.1142/S0218194014400166

Cousot, P., & Cousot, R. (1977). Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, 238–252.

Craswell, N. (2009). Mean reciprocal rank. In L. Liu & M. T. Özsu (Eds.), *Encyclopedia of database systems* (pp. 1703–1703). Springer US. https://doi.org/10.1007/978-0-387-39940-9_488

Cubranic, D., Murphy, G., Singer, J., & Booth, K. (2005). Hipikat: A project memory for software development. *Software Engineering, IEEE Transactions on*, *31*, 446–465. https://doi.org/10.1109/TSE.2005.71

Czerwonka, J., Nagappan, N., Schulte, W., & Murphy, B. (2013). Codemine: Building a software development data analytics platform at microsoft. *IEEE Software*, *30*(4), 64–71. https://doi.org/10.1109/MS.2013.68

Dagenais, B., & Robillard, M. P. (2011). Recommending adaptive changes for framework evolution. *ACM Trans. Softw. Eng. Methodol.*, *20*(4). https://doi.org/10.1145/2000799.2000805

De Moor, O., Verbaere, M., & Hajiyev, E. (2007). Keynote address:. ql for source code analysis. *Source Code Analysis and Manipulation, 2007. SCAM 2007. Seventh IEEE International Working Conference on*, 3–16.

Dell, N., Vaidyanathan, V., Medhi, I., Cutrell, E., & Thies, W. (2012). "yours is better!": Participant response bias in hci. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 1321–1330. https://doi.org/10.1145/2207676.2208589

de Souza, C. R. B., Redmiles, D., & Dourish, P. (2003a). "breaking the code", moving between private and public work in collaborative software development. *Proceedings of the 2003 International ACM SIGGROUP Conference on Supporting Group Work*, 105–114. https://doi.org/10.1145/958160.958177

de Souza, C. R. B., Redmiles, D., & Dourish, P. (2003b). “breaking the code”, moving between private and public work in collaborative software development. *Proceedings of the 2003 International ACM SIGGROUP Conference on Supporting Group Work*, 105–114. https://doi.org/10.1145/958160.958177

Dey, T., Mousavi, S., Ponce, E., Fry, T., Vasilescu, B., Filippova, A., & Mockus, A. (2020). Detecting and characterizing bots that commit code. *Proceedings of the 17th International Conference on Mining Software Repositories (MSR20)*, 209–219.

Dias, K., Borba, P., & Barreto, M. (2020a). Understanding predictive factors for merge conflicts. *Information and Software Technology*, *121*, 106256. https://doi.org/10.1016/j.infsof.2020.106256

Dias, K., Borba, P., & Barreto, M. (2020b). Understanding predictive factors for merge conflicts. *Information and Software Technology*, *121*, 106256.

Dintzner, N., Deursen, A., & Pinzger, M. (2018). Fever: An approach to analyze feature-oriented changes and artefact co-evolution in highly configurable systems. *Empirical Software Engineering*, *23*.

Ebert, C., Gallardo, G., Hernantes, J., & Serrano, N. (2016). Devops. *Ieee Software*, *33*(3), 94–100.

Estler, H. C., Nordio, M., Furia, C. A., & Meyer, B. (2014). Awareness and merge conflicts in distributed software development. *2014 IEEE 9th International Conference on Global Software Engineering*, 26–35. https://doi.org/10.1109/ICGSE.2014.17

Faro, A., Giordano, D., & Venticinque, M. (2021). Internetworked wrist sensing devices for pervasive and m-connected eldercare. *2021 IEEE 3rd Global Conference on Life Sciences and Technologies (LifeTech)*, 454–456. https://doi.org/10.1109/LifeTech52111.2021.9391828

Forsgren, N., Humble, J., & Kim, G. (2018). *Accelerate: The science of lean software and devops building and scaling high performing technology organizations* (1st). IT Revolution Press.

Fowler, M. (2020). Semantic conflict.

Fritz, T., & Murphy, G. C. (2010). Using information fragments to answer the questions developers ask. *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, 175–184.

Fu, C., Zhou, M., Xuan, Q., & Hu, H.-X. (2017). Expert recommendation in oss projects based on knowledge embedding. *2017 International Workshop on Complex Systems and Networks (IWCSN)*, 149–155. https://doi.org/10.1109/IWCSN.2017.8276520

GitHub. (n.d.). https://flow.microsoft.com/en-us/blog/sending-pull-request-review-reminders-using-ms-flows/

GitHub. (2020). https://www.openml.org/a/estimation-procedures/9

GitHub. (2020). https://github.com/about

GitHub Marketplace. (2020). https://github.com/marketplace

Github personal dashboard. (2021). https://docs.github.com/en/account-and-profile/setting-up-and-managing-your-github-user-account/managing-user-account-settings/about-your-personal-dashboard

Godefroid, P., Levin, M. Y., Molnar, D. A., et al. (2008). Automated whitebox fuzz testing. *NDSS*, *8*, 151–166.

Godlin, B., & Strichman, O. (2009). Regression verification. *Proceedings of the 46th Annual Design Automation Conference*, 466–471.

Gousios, G., Pinzger, M., & van Deursen, A. (2014). An exploratory study of the pull-based software development model. In P. Jalote, L. C. Briand, & A. van der Hoek (Eds.), *Icse* (pp. 345–355). ACM. http://dblp.uni-trier.de/db/conf/icse/icse2014.html#GousiosPD14

Graph Neural Network Techniques. (2021). https://neptune.ai/blog/graph-neural-network-and-some-of-gnn-applications

Gremlin query language. (2021). https://docs.janusgraph.org/getting-started/gremlin/

Grinter, R. E. (1995). Using a configuration management tool to coordinate software development. *Proceedings of Conference on Organizational Computing Systems*, 168–177. https://doi.org/10.1145/224019.224036

Guimarães, M. L., & Silva, A. R. (2012). Improving early detection of software merge conflicts. *2012 34th International Conference on Software Engineering (ICSE)*, 342–352. https://doi.org/10.1109/ICSE.2012.6227180

Guzzi, A., Bacchelli, A., Riche, Y., & van Deursen, A. (2015). Supporting developers' coordination in the ide. *Proceedings of the 18th ACM Conference on Computer Supported Cooperative Work and Social Computing*, 518–532. https://doi.org/10.1145/2675133.2675177

Hammad, M., Hijazi, H., Hammad, M., & Otoom, A. (2020). Mining expertise of developers from software repositories. *International Journal of Computer Applications in Technology*, *62*, 227.

Holmes, R., Walker, R. J., & Murphy, G. C. (2006). Approximate structural context matching: An approach to recommend relevant examples. *IEEE Transactions on Software Engineering*, *32*(12), 952–970. https://doi.org/10.1109/TSE.2006.117

Horwitz, S., Prins, J., & Reps, T. (1989). Integrating noninterfering versions of programs. *ACM Trans. Program. Lang. Syst.*, *11*(3), 345–387. https://doi.org/10.1145/65979.65980

Huijgens, H., Rastogi, A., Mulders, E., Gousios, G., & Deursen, A. v. (2020). Questions for data scientists in software engineering: A replication. *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 568–579.

Infonetics, I. (2016). The cost of server, application, and network downtime: Annual north american enterprise survey and calculator. *Computing*.

Isard, M. (2007). Autopilot: Automatic data center management. *SIGOPS Oper. Syst. Rev.*, *41*(2), 60–67. https://doi.org/10.1145/1243418.1243426

Jaferian, P., Botta, D., Raja, F., Hawkey, K., & Beznosov, K. (2008). Guidelines for designing it security management tools. *Proceedings of the 2nd ACM Symposium on Computer Human interaction For Management of information Technology*, 1–10.

Kalliamvakou, E., Gousios, G., Blincoe, K., Singer, L., German, D. M., & Damian, D. (2014). The promises and perils of mining github. *Proceedings of the 11th Working Conference on Mining Software Repositories*, 92–101. https://doi.org/10.1145/2597073.2597074

Kasi, B. K., & Sarma, A. (2013a). Cassandra: Proactive conflict minimization through optimized task scheduling. *Proceedings of the 2013 International Conference on Software Engineering*, 732–741.

Kasi, B. K., & Sarma, A. (2013b). Cassandra: Proactive conflict minimization through optimized task scheduling. *2013 35th International Conference on Software Engineering (ICSE)*, 732–741. https://doi.org/10.1109/ICSE.2013.6606619

Kendall, J. M. (2003). Designing a research project: Randomised controlled trials and their principles. *Emergency Medicine Journal*, *20*(2), 164–168. https://doi.org/10.1136/emj.20.2.164

Kim, S., Zimmermann, T., Pan, K., & Jr. Whitehead, E. J. (2006). Automatic identification of bug-introducing changes. *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*, 81–90. https://doi.org/10.1109/ASE.2006.23

Ko, A. J., DeLine, R., & Venolia, G. (2007). Information needs in collocated software development teams. *29th International Conference on Software Engineering (ICSE'07)*, 344–353.

Kumar, R., Bansal, C., Maddila, C., Sharma, N., Martelock, S., & Bhargava, R. (2019a). Building sankie: An ai platform for devops. *2019 IEEE/ACM 1st International Workshop on Bots in Software Engineering (BotSE)*, 48–53. https://doi.org/10.1109/BotSE.2019.00020

Kumar, R., Bansal, C., Maddila, C., Sharma, N., Martelock, S., & Bhargava, R. (2019b). Building sankie: An AI platform for devops. *2019 IEEE/ACM 1st International Workshop on Bots in Software Engineering (BotSE)*, 48–53.

Kumer, S. A., Kanakaraja, P., Teja, A. P., Sree, T. H., & Tejaswni, T. (2021). Smart home automation using ifttt and google assistant [International Conference on Materials, Manufacturing and Mechanical Engineering for Sustainable Developments-2020 (ICMSD 2020)]. *Materials Today: Proceedings*, *46*, 4070–4076. https://doi.org/https://doi.org/10.1016/j.matpr.2021.02.610

Lahiri, S. K., McMillan, K. L., Sharma, R., & Hawblitzel, C. (2013). Differential assertion checking. *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, 345–355.

Lahiri, S. K., Vaswani, K., & Hoare, C. A. R. (2010). Differential static analysis: Opportunities, applications, and challenges. *Proceedings of the FSE/SDP*

Workshop on Future of Software Engineering Research, 201–204. https://doi.org/10.1145/1882362.1882405

Lam, A. N., Nguyen, A. T., Nguyen, H. A., & Nguyen, T. N. (2015). Combining deep learning with information retrieval to localize buggy files for bug reports (n). *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, 476–481.

Lanza, M., D'Ambros, M., Bacchelli, A., Hattori, L., & Rigotti, F. (2013). Manhattan: Supporting real-time visual team activity awareness. *2013 21st International Conference on Program Comprehension (ICPC)*, 207–210. https://doi.org/10.1109/ICPC.2013.6613849

Layman, L., Nagappan, N., Guckenheimer, S., Beehler, J., & Begel, A. (2008). Mining software effort data: Preliminary analysis of visual studio team system data., 43–46. https://doi.org/10.1145/1370750.1370762

Lebeuf, C., Storey, M.-A., & Zagalsky, A. (2018). Software bots. *IEEE Software*, *35*(1), 18–23.

Lebeuf, C., Zagalsky, A., Foucault, M., & Storey, M.-A. (2019). Defining and classifying software bots: A faceted taxonomy. *2019 IEEE/ACM 1st International Workshop on Bots in Software Engineering (BotSE)*, 1–6.

Lens Explorer. (2021). https://docs.microsoft.com/en-us/system-center/orchestrator/learn-about-orchestrator

Liblit, B., Aiken, A., Naik, M., & Zheng, A. X. (2005). Scalable statistical bug isolation. *In Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI)*, 15–26.

Liu, D., Lin, C., Zhang, Z., Xiao, Y., & Tong, H. (2019). Spiral of silence in recommender systems. *Proceedings of the Twelfth ACM International Conference on Web Search and Data Mining*, 222–230. https://doi.org/10.1145/3289600.3291003

Long-Running Branches Considered Harmful. (2020).

Luhn, H. P. (1959). Key word-in-context index for technical literature (kwic index). *Proceedings of the 136th Meeting of the American Chemical Society, Division of Chemical Literature*.

Luhn, H. P. (1960). Key word-in-context index for technical literature (kwic index). *American Documentation*, *11*, 288–295.

Ma, Y., Bogart, C., Amreen, S., Zaretzki, R., & Mockus, A. (2019). World of code: An infrastructure for mining the universe of open source vcs data. *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, 143–154.

MacLeod, L., Greiler, M., Storey, M., Bird, C., & Czerwonka, J. (2018). Code reviewing in the trenches: Challenges and best practices. *IEEE Software*, *35*(4), 34–42. https://doi.org/10.1109/MS.2017.265100500

Maddila, C., Agrawal, A., Zimmermann, T., Forsgren, N., Herzig, K., Deursen, A. v., Maddila, C., Forsgren, N., & Herzig, K. (2021). *Appendix to nalanda: A*

large-scale socio-technical graph of entities and relationships in software development environment (tech. rep. MSR-TR-2021-28). Microsoft. https://aka.ms/MSR-TR-2021-28

Maddila, C., Bansal, C., & Nagappan, N. (2019). Predicting pull request completion time: A case study on large scale cloud services. *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 874–882. https://doi.org/10.1145/3338906.3340457

Maddila, C. S., Upadrasta, S. S., Bansal, C., Nagappan, N., Gousios, G., & van Deursen, A. (2021). Nudge: Accelerating overdue pull requests towards completion. *CoRR*, *abs/2011.12468*. https://arxiv.org/abs/2011.12468

Manning, C. D., Raghavan, P., & Schütze, H. (2008). *Introduction to information retrieval*. Cambridge University Press.

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Jia, Y., Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, . . . Xiaoqiang Zheng. (2015). TensorFlow: Large-scale machine learning on heterogeneous systems [Software available from tensorflow.org]. https://www.tensorflow.org/

McKee, S., Nelson, N., Sarma, A., & Dig, D. (2017). Software practitioner perspectives on merge conflicts and resolutions. *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 467–478. https://doi.org/10.1109/ICSME.2017.53

McMillan, K. L. (1993). Symbolic model checking. *Symbolic model checking* (pp. 25–60). Springer.

Mehta, S., Bhagwan, R., Kumar, R., Bansal, C., Maddila, C., Ashok, B., Asthana, S., Bird, C., & Kumar, A. (2020). Rex: Preventing bugs and misconfiguration in large services using correlated change analysis. *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI20)*, 435–448.

Menon, V. G., Jacob, S., Joseph, S., Sehdev, P., Khosravi, M. R., & Al-Turjman, F. (2020). An iot-enabled intelligent automobile system for smart cities. *Internet of Things*, 100213. https://doi.org/https://doi.org/10.1016/j.iot.2020.100213

Menzies, T., Butcher, A., Cok, D., Marcus, A., Layman, L., Shull, F., Turhan, B., & Zimmermann, T. (2013). Local versus global lessons for defect prediction and effort estimation. *IEEE Transactions on Software Engineering*, *39*(6), 822–834. https://doi.org/10.1109/TSE.2012.83

Microsoft facts. (2020).

Microsoft Power Automate. (2020). https://flow.microsoft.com/en-us/

Mockus, A., & Herbsleb, J. (2002). Expertise browser: A quantitative approach to identifying expertise. *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*, 503–512.

Mockus, A., & Herbsleb, J. D. (2002). Expertise browser: A quantitative approach to identifying expertise. *Proceedings of the 24th International Conference on Software Engineering (ICSE 2002)*, 503–512.

Nelson, N., Brindescu, C., McKee, S., Sarma, A., & Dig, D. (2019). The life-cycle of merge conflicts: Processes, barriers, and strategies. *Empirical Software Engineering*, *24*. https://doi.org/10.1007/s10664-018-9674-x

Nickerson, R. (1998). Confirmation bias: A ubiquitous phenomenon in many guises. *Review of General Psychology*, *2*, 175–220. https://doi.org/10.1037/1089-2680.2.2.175

Nieminen, A. (2012). Real-time collaborative resolving of merge conflicts. *8th International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom)*, 540–543. https://doi.org/10.4108/icst.collaboratecom.2012.250435

Ostrand, T. J., Weyuker, E. J., & Bell, R. M. (2004). Where the bugs are. *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, 86–96. https://doi.org/10.1145/1007512.1007524

Ovadia, S. (2014). Automate the internet with “if this then that” (ifttt). *Behavioral & Social Sciences Librarian*, *33*(4), 208–211. https://doi.org/10.1080/01639269.2014.964593

Owhadi-Kareshk, M., Nadi, S., & Rubin, J. (2019). Predicting merge conflicts in collaborative software development. *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 1–11. https://doi.org/10.1109/ESEM.2019.8870173

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Köpf, A., Yang, E. Z., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., . . . Chintala, S. (2019). Pytorch: An imperative style, high-performance deep learning library. *CoRR*, *abs/1912.01703*. http://arxiv.org/abs/1912.01703

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., & Duchesnay, E. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, *12*, 2825–2830.

Person, S., Dwyer, M. B., Elbaum, S., & Pasareanu, C. S. (2008). Differential symbolic execution. *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, 226–237.

Potvin, R., & Levenberg, J. (2016). Why google stores billions of lines of code in a single repository. *Communications of the ACM*, *59*(7), 78–87.

The probabilistic relevance framework: Bm25 and beyond. (2009). *Foundations and Trends in Information Retrieval*, *3*, 333–389. https://doi.org/10.1561/1500000019

Rao, S., & Kak, A. (2011). Retrieval from software libraries for bug localization: A comparative study of generic and composite text models. *Proceedings of the 8th Working Conference on Mining Software Repositories*, 43–52.

Rastogi, A., Nagappan, N., Gousios, G., & van der Hoek, A. (2018). Relationship between geographical location and evaluation of developer contributions in github. *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 22:1–22:8. https://doi.org/10.1145/3239235.3240504

Rath, M., Lo, D., & Mäder, P. (2018). *Replication data for: Analyzing requirements and traceability information to improve bug localization*. https://doi.org/10.7910/DVN/N5APOB

Ren, L., Zhou, S., Kästner, C., & Wąsowski, A. (2019). Identifying redundancies in fork-based development. *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 230–241.

Robertson, S. E., Walker, S., Hancock-Beaulieu, M., Gull, A., & Lau, M. (1992). Okapi at TREC. In D. K. Harman (Ed.), *Proceedings of the first text retrieval conference, TREC 1992, gaithersburg, maryland, usa, november 4-6, 1992* (pp. 21–30). National Institute of Standards; Technology (NIST). http://trec.nist.gov/pubs/trec1/papers/02.txt

Robillard, M., Walker, R., & Zimmermann, T. (2010). Recommendation systems for software engineering. *IEEE Software*, *27*(4), 80–86. https://doi.org/10.1109/MS.2009.161

Robillard, M. P., Maalej, W., Walker, R. J., & Zimmermann, T. (2014). *Recommendation systems in software engineering*. Springer Publishing Company, Incorporated.

Robinson, J. (2014). Likert scale. In A. C. Michalos (Ed.), *Encyclopedia of quality of life and well-being research* (pp. 3620–3621). Springer Netherlands. https://doi.org/10.1007/978-94-007-0753-5_1654

Rocco, J., Di Ruscio, D., Di Sipio, C., Nguyen, P., & Rubei, R. (2021). Development of recommendation systems for software engineering: The crossminer experience. *Empirical Software Engineering*, *26*. https://doi.org/10.1007/s10664-021-09963-7

Rochkind, M. J. (1975). The source code control system. *IEEE Transactions on Software Engineering*, *1*(4), 364–370. https://doi.org/10.1109/TSE.1975.6312866

Runeson, P., & Höst, M. (2009). Guidelines for conducting and reporting case study research in software engineering. *14*(2), 131–164. https://doi.org/10.1007/s10664-008-9102-8

Russinovich, M. E., Solomon, D. A., & Ionescu, A. (2012). *Windows internals, part 1: Covering windows server 2008 r2 and windows 7* (6th). Microsoft Press.

Sarma, A., Maccherone, L., Wagstrom, P., & Herbsleb, J. (2009). Tesseract: Interactive visual exploration of socio-technical relationships in software development. *2009 IEEE 31st International Conference on Software Engineering*, 23–33. https://doi.org/10.1109/ICSE.2009.5070505

Sarma, A., Maccherone, L., Wagstrom, P., & Herbsleb, J. D. (2009). Tesseract: Interactive visual exploration of socio-technical relationships in software development. *2009 IEEE 31st International Conference on Software Engineering*, 23–33.

Sarma, A., Noroozi, Z., & van der Hoek, A. (2003). Palantír: Raising awareness among configuration management workspaces. *Proceedings of the 25th ACM/IEEE International Conference on Software Engineering (ICSE)*, 444–454. https://doi.org/10.1109/ICSE.2003.1201222

Schütze, H., Manning, C. D., & Raghavan, P. (2008). *Introduction to information retrieval* (Vol. 39). Cambridge University Press Cambridge.

Shull, F., Singer, J., & Sjøberg, D. I. (2007). *Guide to advanced empirical software engineering*. Springer.

Sillito, J., Murphy, G. C., & De Volder, K. (2008). Asking and answering questions during a programming change task. *IEEE Transactions on Software Engineering*, *34*(4), 434–451.

Soares, D. M., de Lima Júnior, M. L., Murta, L., & Plastino, A. (2015). Acceptance factors of pull requests in open-source projects. *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, 1541–1546. https://doi.org/10.1145/2695664.2695856

Steck, H. (2011). Item popularity and recommendation accuracy. *Proceedings of the Fifth ACM Conference on Recommender Systems*, 125–132. https://doi.org/10.1145/2043932.2043957

Storey, M.-A., Serebrenik, A., Rosé, C. P., Zimmermann, T., & Herbsleb, J. D. (2020). Botse: Bots in software engineering (dagstuhl seminar 19471). *Dagstuhl Reports*, *9*(11).

Terrell, J., Kofink, A., Middleton, J., Rainear, C., Murphy-Hill, E., Parnin, C., & Stallings, J. (2017). Gender differences and bias in open source: Pull request acceptance of women versus men. *PeerJ Computer Science*, *3*, e111. https://doi.org/10.7717/peerj-cs.111

Thummalapenta, S., & Xie, T. (2007). Parseweb: A programmer assistant for reusing open source code on the web. *ASE'07 - 2007 ACM/IEEE International Conference on Automated Software Engineering*, 204–213. https://doi.org/10.1145/1321631.1321663

Tsay, J., Dabbish, L., & Herbsleb, J. (2014). Influence of social and technical factors for evaluating contribution in github. *Proceedings of the 36th International*

Conference on Software Engineering, 356–366. https://doi.org/10.1145/2568225.2568315

Van Der Veen, E., Gousios, G., & Zaidman, A. (2015). Automatically prioritizing pull requests. *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, 357–361.

Wang, Q., Xu, B., Xia, X., Wang, T., & Li, S. (2019). Duplicate pull request detection: When time matters. *Proceedings of the 11th Asia-Pacific Symposium on Internetware*, 1–10.

Wang, S., & Lo, D. (2016). Amalgam+: Composing rich information sources for accurate bug localization. *Journal of Software: Evolution and Process*, *28*(10), 921–942.

Wang, S., Bansal, C., & Nagappan, N. (2020). Large-scale intent analysis for identifying large-review-effort code changes. *Information and Software Technology*, 106408.

Wang, S., Bansal, C., Nagappan, N., & Philip, A. A. (2019a). Leveraging change intents for characterizing and identifying large-review-effort changes. *Proceedings of the Fifteenth International Conference on Predictive Models and Data Analytics in Software Engineering*, 46–55. https://doi.org/10.1145/3345629.3345635

Wang, S., Bansal, C., Nagappan, N., & Philip, A. A. (2019b). Leveraging change intents for characterizing and identifying large-review-effort changes. *Proceedings of the Fifteenth International Conference on Predictive Models and Data Analytics in Software Engineering*, 46–55.

Weiser, M. (1981). Program slicing. *Proceedings of the 5th international conference on Software engineering*, 439–449.

Wieringa, R. (2014). Design science methodology for information systems and software engineering. *Springer Berlin Heidelberg*.

Williams, C., & Spacco, J. (2008). Szz revisited: Verifying when changes induce fixes. *Proceedings of the 2008 Workshop on Defects in Large Software Systems*, 32–36. https://doi.org/10.1145/1390817.1390826

Winters, T., Manshreck, T., & Wright, H. (2020). *Software engineering at google: Lessons learned from programming over time: The science of microfabrication*. O'Reilly Media.

Wyrich, M., & Bogner, J. (2019). Towards an autonomous bot for automatic source code refactoring. *2019 IEEE/ACM 1st International Workshop on Bots in Software Engineering (BotSE)*, 24–28.

Xu, W., Huang, L., Fox, A., Patterson, D., & Jordan, M. I. (2009). Detecting large-scale system problems by mining console logs. *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, 117–132. https://doi.org/10.1145/1629575.1629587

Yang, J.-M., Cai, R., Jing, F., Wang, S., Zhang, L., & Ma, W.-Y. (2008). Search-based query suggestion. *Proceedings of the 17th ACM Conference on Information*

and Knowledge Management, 1439–1440. https://doi.org/10.1145/1458082.1458321

Ye, Y., & Fischer, G. (2002). Supporting reuse by delivering task-relevant and personalized information. *Proceedings of the 24th International Conference on Software Engineering*, 513–523. https://doi.org/10.1145/581339.581402

Ye, Y., & Fischer, G. (2005). Reuse-conducive development environments. *Autom. Softw. Eng.*, *12*, 199–235. https://doi.org/10.1007/s10515-005-6206-x

Yin, R. K. (2008). *Case study research: Design and methods (applied social research methods)* (Fourth Edition.). Sage Publications.

Youm, K. C., Ahn, J., Kim, J., & Lee, E. (2015). Bug localization based on code change histories and bug reports. *Software Engineering Conference (APSEC), 2015 Asia-Pacific*, 190–197.

Yu, M., Greenberg, A. G., Maltz, D. A., Rexford, J., Yuan, L., Kandula, S., & Kim, C. (2018). Profiling network performance for multi-tier data center applications. *In Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI).*

Yu, Y., Wang, H., Filkov, V., Devanbu, P., & Vasilescu, B. (2015). Wait for it: Determinants of pull request evaluation latency on github. *Proceedings of the 12th Working Conference on Mining Software Repositories*, 367–371. http://dl.acm.org/citation.cfm?id=2820518.2820564

Yu, Y., Wang, H., Yin, G., & Wang, T. (2016). Reviewer recommendation for pull-requests in github: What can we learn from code review and bug assignment? *Information and Software Technology*, *74*, 204–218.

Yuan, D., Park, S., Huang, P., Liu, Y., Lee, M. M., Tang, X., Zhou, Y., & Savage, S. (2012). Be conservative: Enhancing failure diagnosis with proactive logging. *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, 293–306. http://dl.acm.org/citation.cfm?id=2387880.2387909

Zanjani, M. B., Kagdi, H. H., & Bird, C. (2016). Automatically recommending peer reviewers in modern code review. *IEEE Trans. Software Eng.*, *42*(6), 530–543. https://doi.org/10.1109/TSE.2015.2500238

Zeller, A. (2002). Isolating cause-effect chains from computer programs. *Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering*, 1–10.

Zhang, F., Khomh, F., Zou, Y., & Hassan, A. E. (2012). An empirical study of the effect of file editing patterns on software quality. *2012 19th Working Conference on Reverse Engineering*, 456–465. https://doi.org/10.1109/WCRE.2012.55

Zhang, Q., Yu, G., Guo, C., Dang, Y., Swanson, N., Yang, X., Yao, R., Chintalapati, M., Krishnamurthy, A., & Anderson, T. (2011). Deepview: Virtual disk failure diagnosis and pattern detection for azure. *In Proceedings of the*

8th USENIX Symposium on Networked Systems Design and Implementation (NSDI).

Zhang, X., Wang, T., Yin, G., Yang, C., Yu, Y., & Wang, H. (2017). Devrec: A developer recommendation system for open source repositories, 3–11. https://doi.org/10.1007/978-3-319-56856-0_1

Zhao, X., Rodrigues, K., Luo, Y., Stumm, M., Yuan, D., & Zhou, Y. (2017). Log20: Fully automated optimal placement of log printing statements under specified overhead threshold. *Proceedings of the 26th Symposium on Operating Systems Principles*, 565–581. https://doi.org/10.1145/3132747.3132778

Zhao, X., Zhang, Y., Lion, D., Ullah, M. F., Luo, Y., Yuan, D., & Stumm, M. (2014). Lprof: A non-intrusive request flow profiler for distributed systems. *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, 629–644. http://dl.acm.org/citation.cfm?id=2685048.2685099

Zhou, J., Zhang, H., & Lo, D. (2012). Where should the bugs be fixed?-more accurate information retrieval-based bug localization based on bug reports. *Proceedings of the 34th International Conference on Software Engineering*, 14–24.

CURRICULUM VITÆ

Chandra Sekhar MADDILA

EDUCATION

2006–2010	Bachelor of Technology JNTU Kakinada, India
2015–2017	Master of Technology Birla Institute of technology and Sciences Pilani, India
2020–2022	PhD. Computer Science Technische Universiteit Delft, The Netherlands
Thesis:	Recommender Systems for DevOps
Promotor:	Prof. Dr. A. Van Deursen

AWARDS, INVITED TALKS, MEDIA

2018	Jay Lepreau best paper award in USENIX OSDI
2019	Invited talk on differential bug localization (ORCA) at USENIX Annual technical Conference (ATC)
2020	My work (Nudge) was featured in prestigious tech magazine VentureBeat
2020	My work (Sankie) was featured in prestigious Microsoft Research Podcast series.
2021	My work (MEC) was featured in prestigious Time magazine
2021	My work (SPACE framework) was featured in prestigious tech magazines such as Business Insider and InfoQ
2021	My work (SPACE framework) was featured in Communications of ACM (CACM) and ACM Queue
2021	Invited talk on AIOps in continuous software engineering. University of Victoria

List of Publications

18. Chandra Maddila, Apoorva Agrawal, Thomas Zimmermann, Nicole Forsgren, Kim Herzig, Arie van Deursen, *Nalanda: A Socio-Technical Graph for Building Software Analytics Tools at Enterprise Scale*, ESEC/FSE 2022.

17. Chandra Maddila, Nachiappan Nagappan, Christian Bird, Georgios Gousios, Arie van Deursen, *ConE: A Concurrent Edit Detection Tool for Large ScaleSoftware Development*, ESEC/FSE 2022, TOSEM.

16. Denae Ford, Margaret-Anne D. Storey, Thomas Zimmermann, Christian Bird, Sonia Jaffe, Chandra Shekhar Maddila, Jenna L. Butler, Brian Houck, Nachiappan Nagappan, *A Tale of Two Cities: Software Developers Working from Home During the COVID-19 Pandemic*, ICSE 2022.

15. Nusrat Zahan, Laurie Williams, Thomas Zimmermann, Patrice Godefroid, Brendan Murphy, Chandra Maddila, *What are Weak Links in the npm Supply Chain?*, ICSE 2022.

14. Jiyang Zhang, Chandra Maddila, Ram Bairi, Christian Bird, Ujjwal Raizada, Apoorva Agrawal, Yamini Jhawar, Kim Herzig, Arie van Deursen (first two authors contributed equally), *Using Large-scale Heterogeneous Graph Representation Learning for Code Review Recommendations*, arxiv.

13. Nicole Forsgren, Margaret Anne Storey, Chandra Maddila, Thomas Zimmermann, Brian Houck, Jenna Butler, *The SPACE of Developer Productivity: There's more to it than you think*, ACM Queue, January 2021. *This is one of the top downloaded ACM papers with 161K downloads in ten months.*

12. Jaime Teevan, Brent Hecht, Sonia Jaffe, Nancy Baym, Rachel Bergmann, Matt Brodsky, Bill Buxton, Jenna Butler, Adam Coleman, Mary Czerwinski, Brian Houck, Ginger Hudson, Shamsi Iqbal, Chandra Maddila, Kate Nowak, Emily Peloquin, Ricardo Reyna Fernandez, Sean Rintel, Abigail Sellen, Tiffany Smith, Margaret-Anne Storey, Siddharth Suri, Hana Wolf, Longqi Yang, *The New Future of Work: Research from Microsoft into the Pandemic's Impact on Work Practices*, Microsoft Technical Report.

11. Denae Ford, Margaret-Anne D. Storey, Thomas Zimmermann, Christian Bird, Sonia Jaffe, Chandra Shekhar Maddila, Jenna L. Butler, Brian Houck, Nachiappan Nagappan, *A Tale of Two Cities: Software Developers Working from Home During the COVID-19 Pandemic*, TOSEM, April 2022.

10. Chandra Maddila, Sai Surya Upadrasta, Chetan Bansal, Nachiappan Nagappan, Georgios Gousios, Arie van Deursen, *Nudge: Accelerating Overdue Pull Requests Towards Completion*, arxiv.

9. Nikitha Rao, Chetan Bansal, Subhabrata Mukherjee, Chandra Shekhar Maddila, *Product Insights: Analyzing Product Intents in Web Search*, CIKM 2020.

8. Chetan Bansal, Pantazis Deligiannis, Chandra Shekhar Maddila, Nikitha Rao, *Studying Ransomware Attacks Using Web Search Logs*, SIGIR 2020.

7. Sonu Mehta, Ranjita Bhagwan, Rahul Kumar, Chetan Bansal, Chandra Maddila, B Ashok, Sumit Asthana, Christian Bird, Aditya Kumar, *Rex: Preventing Bugs and Misconfiguration in Large Services Using Correlated Change Analysis*, USENIX NSDI 2020.

6. Sumit Asthana, Rahul Kumar, Ranjita Bhagwan, Christian Bird, Chetan Bansal, Chandra Maddila, Sonu Mehta, B. Ashok, *WhoDo: automating reviewer suggestions at scale*, ESEC/SIGSOFT FSE 2019.

5. Chandra Maddila, Chetan Bansal, Nachiappan Nagappan, *Predicting pull request completion time: a case study on large scale cloud services*, ESEC/FSE 2019.

4. Rahul Kumar, Chetan Bansal, Chandra Shekhar Maddila, Nitin Sharma, Shawn Martelock, Ravi Bhargava, *Building sankie: an AI platform for DevOps*, BotSE@ICSE 2019.

3. Adithya Abraham Philip, Ranjita Bhagwan, Rahul Kumar, Chandra Maddila, Nachiappan Nagappan, *FastLane: test minimization for rapidly deployed large-scale online services*, ICSE 2019.

2. Ranjita Bhagwan, Rahul Kumar, Chandra Maddila, Adithya Abraham Philip (author names are listed in the order of last names): ORCA: Differential Bug Localization in Large-Scale Services, *ORCA: Differential Bug Localization in Large-Scale Services*, OSDI 2018. **Jay Lepreau best paper award**

1. Shruti Rijhwani, Royal Sequiera, Monojit Choudhury, Kalika Bali, Chandra Maddila, *Estimating Code-Switching on Twitter with a Novel Generalized Word-Level Language Detection Technique*, ACL 2017.