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

Maddila, Chandra; Nagappan, Nachiappan; Bird, Christian; Gousios, Georgios; van Deursen, Arie

DOI
10.1145/3478019

Publication date
2022

Document Version
Final published version

Published in
ACM Transactions on Software Engineering and Methodology

ConE: A Concurrent Edit Detection Tool for Large-scale Software Development

CHANDRA MADDILA, NACHIAPPAN NAGAPPAN, and CHRISTIAN BIRD,
Microsoft Research
GEORGIOS GOUSIOS, Facebook
ARIE van DEURSEN, Delft University of Technology

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 Concurrent Edit Detector (ConE) 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 useful 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.

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

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

ACM Reference format:
Chandra Maddila, Nachiappan Nagappan, Christian Bird, Georgios Gousios, and Arie van Deursen. 2021. ConE: A Concurrent Edit Detection Tool for Large-scale Software Development. *ACM Trans. Softw. Eng. Methodol.* 31, 2, Article 22 (December 2021), 26 pages.
https://doi.org/10.1145/3478019

N. Nagappan work done while at Microsoft Research.
G. Gousios work done while at Delft University of Technology.
Authors' addresses: C. Maddila, N. Nagappan, and C. Bird, Microsoft Research, 14820 NE 36th St, Redmond, WA 98052, USA; emails: chmaddil@microsoft.com, nchiappan.nagappan@gmail.com, cbird@microsoft.com; G. Gousios, Facebook, 1 Hacker Way, Menlo Park, CA 94025, USA; email: gousiosg@fb.com; A. van Deursen, Delft University of Technology, Building 28, Van Mourik Broekmanweg 6, 2628 XE Delft, The Netherlands; email: Arie.vanDeursen@tudelft.nl.

1049-331X/2021/12-ART22
https://doi.org/10.1145/3478019

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" [1, 8, 30, 36]. The side effects can be as simple as syntactic merge conflicts, which can be handled by version control systems [41] and various techniques/tools [20, 26, 29], to semantic conflicts [23]. Such bugs can be very hard to detect and may cause substantial disruptions [29]. 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 [10, 12, 31]. Merge conflicts have a significant impact on code quality and can disrupt the developer workflow [2, 19, 39]. 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 [38], which makes the conflict resolution even harder at a later point of time [6, 38]. 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 [9, 42]. Studies have shown that these problems can be avoided by following strategies such as effective communication within the team [28], and developing awareness about others' changes that have a potential to incur conflicts [22].

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 [46].

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 [37]. 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 12 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 [5, 24] 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 [7, 13, 42] 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 that 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 [13, 15, 42]. 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 (RCEs)** files. 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 article 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 versus non-concurrent edits (Section 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).
- 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 5).
- We present results from our quantitative and qualitative evaluation of the ConE system (Section 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 6).

2 RELATED WORK

The software engineering community has extensively studied the impact of merge conflicts on software quality [2, 10], investigated various methodologies and tools that can help developers discover conflicting changes through interactive visualizations, and developed speculative analysis tools [20, 26, 29]. 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 a 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 [34] 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 [7] 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 that 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 [13, 15, 42]. 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 [42] 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 [14, 27, 32] 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 [40]. 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 [46]. 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 [23], or even cases of code/effort duplication (as explained in Section 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. [47] 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 [3]. Dias et al. proposed methods to understanding predictive factors for merge conflicts [21], i.e., how conflict occurrence is affected by technical and organizational factors. Studies conducted by Blincoe et al. and Cataldo et al. [3, 16] 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 [18] and characterized the problem of developers' assignment for merging branches [17]. 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.

3 CONCURRENT VERSUS NON-CONCURRENT EDITS IN PRACTICE

The differences in the fashion in which edits are made to source code files (concurrent versus non-concurrent) can cause various unintended consequences (as explained in Section 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. [44], 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 toward 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. [44], 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 [33, 45]. Combining both ideas, we created a ground-truth data set that 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 metadata 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 that 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 that 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.

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) versus cloud service (4)), geographical distribution of the teams (U.S. only (2) versus split between different countries and time zones (4)), and programming languages (as listed in Table 3). We performed data cleansing by applying the filters listed below:

- Exclude **pull requests (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 [25].
- 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 exclude PRs that are created by non-human developers that do mass refactoring or styling changes, and so on.
- 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 a lot of developers through a 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 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 1 shows the distribution of concurrently and non-concurrently edited files per repository.

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.

Table 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	3,500	4,875	4,781	41.7	58.2
Repo-2	10,470	16,879	15,678	38.2	61.8
Repo-3	2,907	4,119	5,467	41.3	58.7
Repo-4	5,560	7,550	8,972	42.4	57.6
Repo-5	4,110	7,569	9,786	35.2	64.8
Repo-6	5,987	9,541	9,443	38.5	61.5
Total	32,534	50,533	54,127	39.1	60.9

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 versus non-concurrent).

Figure 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).

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.

While Figure 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 1):

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

The results are in Table 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.

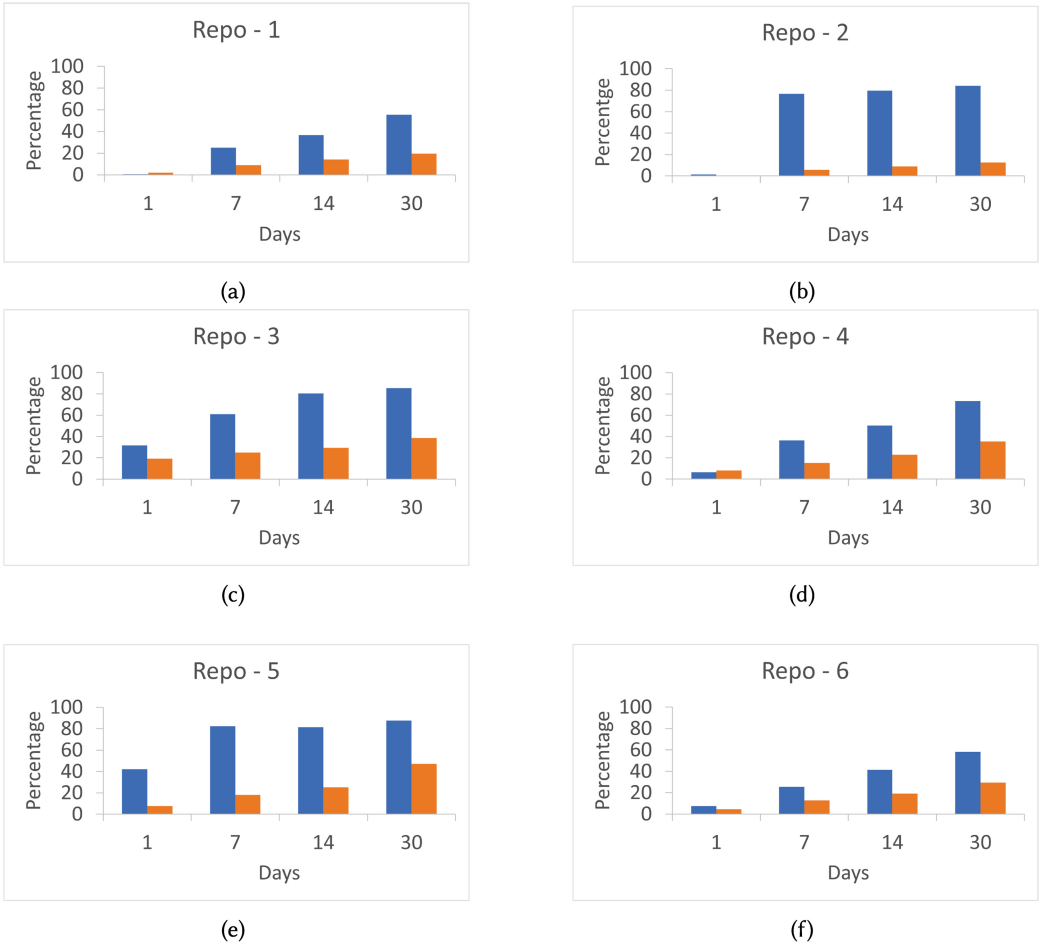

Fig. 1. Graphs showing how the percentage of files seen in bug fixes (within a day, a week, two weeks, and a month) is changing Blue and orange bars represent concurrently and non-concurrently edited files, respectively.

4 SYSTEM DESIGN

Backed by the correlation analysis suggesting that concurrent edits may be 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 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.

Table 2. Spearman Rank Correlation Analysis for Total Edits, Concurrent Edits, Non-concurrent Edits Versus 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^{*}

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

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. However, 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 [46]. However, recent techniques proposed by Brindescu et al. [11], can potentially aid in facilitating a decision by determining the merge conflict situations to flag, based on the complexity of the merge conflict.

4.1 Core Concepts

ConE constantly listens to events that happen in an Azure DevOps environment [5]. 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: EOO and the existence of 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.

4.1.1 Extent of Overlap (EOO). ConE scans all the active pull requests that meet our filtering criteria (explained in Section 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:

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

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, then 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

Table 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. Number of Bug Fixes with RCEs and No RCEs

Edit type	Count	Percentage
Bug fix PRs with no RCEs	1,617	78.3
Bug fix PRs with at least one RCE	446	21.7

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. [40], 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 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.

4.1.2 Rarely Concurrently Edited files (RCEs). These are the files that 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, then changes to these files should be reviewed carefully and everyone involved in editing them should be aware of others' changes.

We performed an empirical analysis, from our shadow mode deployment data (as explained in Section 5.2), to understand how pervasive RCEs really are. As explained in Table 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 that should not be seen a lot. But, if observed, then 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 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.

4.2 The ConE Algorithm

ConE's algorithm to select candidate pull requests that developers need to be notified about primarily leverages the techniques explained above: EOO and existence of RCEs. Together these serve to reduce the total number of active pull requests under consideration, 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 that are active for so long may not even be completed [25]. 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 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 that 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, then there is no need to notify them.

Step 4: Calculate the extent of overlap using the formula described in Section 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, $T_{ea} = [\langle PR_1, PR_7, 55 \rangle, \langle PR_1, PR_{12}, 95 \rangle, \langle PR_1, PR_{34}, 35 \rangle \ldots]$.

Table 5. Distribution of Extent of Overlap

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

Step 5: Check for the existence of 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, $T_{ra} = [\langle PR_1, PR_7, 2 \rangle, \langle PR_1, PR_{12}, 2 \rangle, \langle PR_1, PR_{34}, 9 \rangle \ldots .]$

Step 6: Apply thresholds on the values for extent of overlap and the number of RCEs, as explained in Section 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 that needs to be prioritized.

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 EOO, the number of 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, then consider it for notification. To assess the consequences of this, we randomly selected 1,654 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 3). We manually inspected each of these 1,654 pull requests to make sure the overlap we observe is indeed correct. Our empirical analysis (see Table 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 that are more likely to be in conflict.

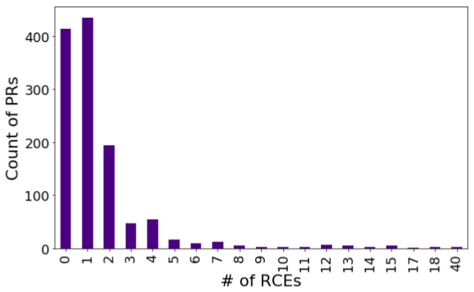

Fig. 2. Distribution of the number of PRs with a given number of RCEs.

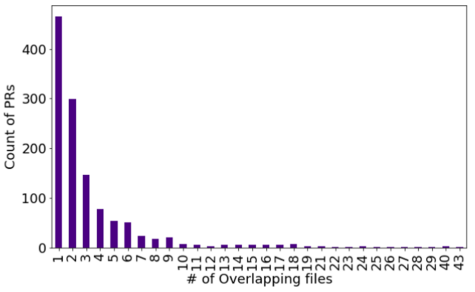

Fig. 3. Distribution of the number of PRs with a given number of overlapping files.

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, then select the active pull request as a candidate. As shown in Figure 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 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 3. As shown in Figure 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 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.

5 IMPLEMENTATION AND DEPLOYMENT

5.1 Core Components and Implementation

The core ConE components are displayed in Figure 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 metadata about pull requests, and service hooks that 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, ConE listens to events triggered by pull requests, and has the ability to decorate pull requests with notifications about potentially

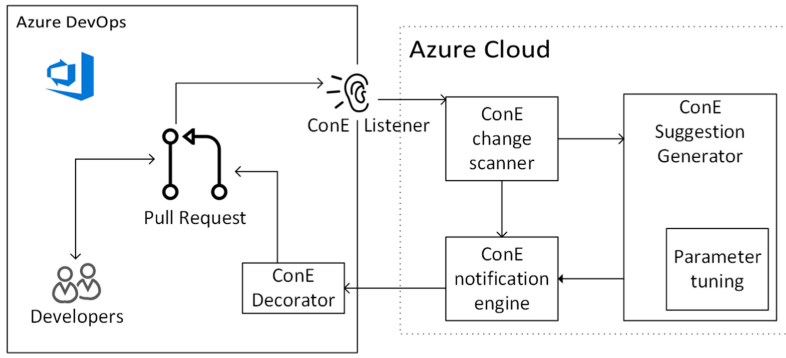

Fig. 4. ConE System design: Pull requests from Azure DevOps are listened to by the ConE change scanner, suggestion generator, notification engine, and decorator.

conflicting other pull requests. The ConE service itself, shown at the right in Figure 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 4.3.

The ConE Service is implemented using C# and .NET 4.7. It has been built on top of Microsoft Azure cloud services: Azure Batch [4] 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.

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 that 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, and so on.
- Include repositories that 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 1,200 suggestions by enabling ConE in this mode for two months. We then

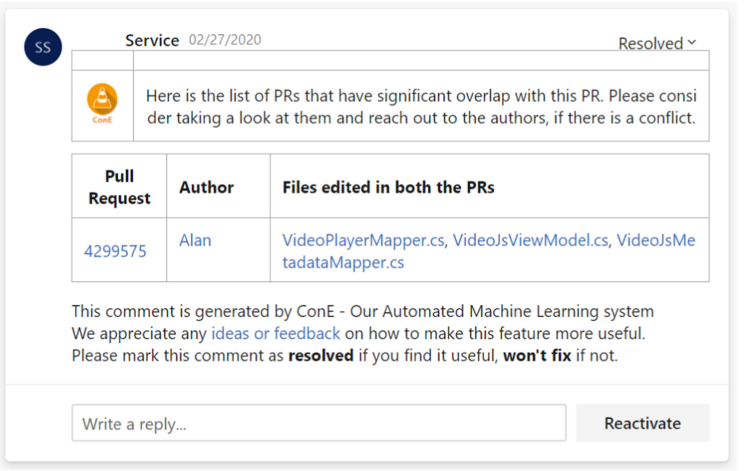

Fig. 5. ConE notification that a pull request has significant overlap with another pull request.

went through 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, and so on. We experimented with tuning our parameters (explained in Section 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 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.

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 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 that 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.

5.4 Scale

The ConE system has been deployed on 234 repositories in Microsoft. The repositories have been picked based on maximizing the diversity and variety of the repositories in various dimensions as explained in Section 5.2. Since enabled in March 2020, until September 2020 (when we pulled the telemetry data) ConE evaluated 26,000 pull requests that 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 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.

6 EVALUATION: DEVELOPERS PERCEPTIONS ABOUT CONE'S USEFULNESS

Of the 26,000 pull requests under analysis during ConE's six month deployment (Section 5.4), ConE's filtering algorithm (Section 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 6, the notification volume varies between repositories.

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. The notification itself also contains instructions, as

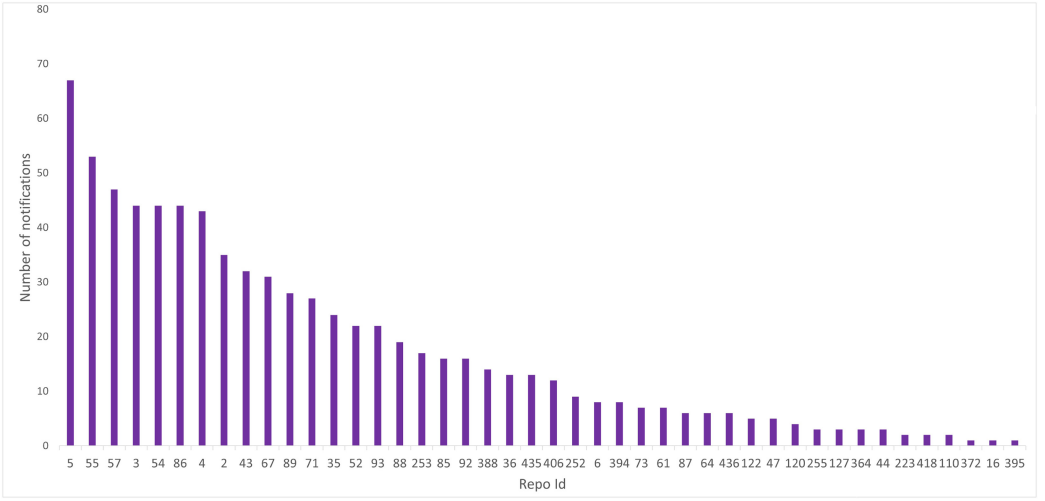

Fig. 6. Distribution of notifications per repository.

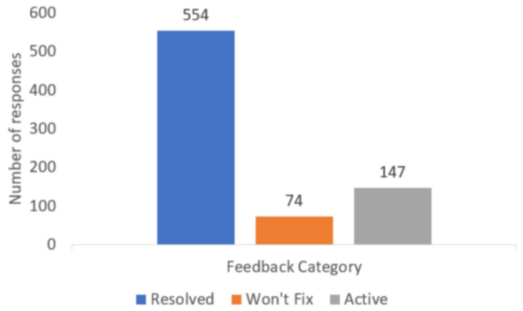

Fig. 7. Number of positive (Resolved), negative (Won't Fix), and neutral (Active) responses to ConE notifications.

shown in Figure 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 7 shows the distribution of the feedback received. The vast majority (554 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 [35, 43]. 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 Section 6.4.

6.2 Extent of Interaction

As discussed in Section 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

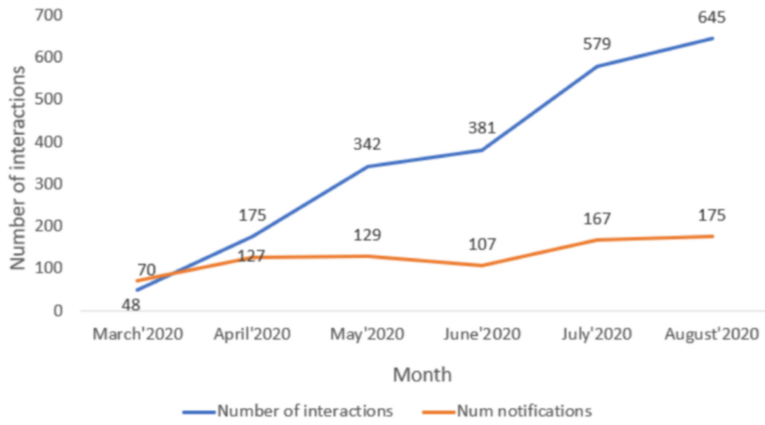

Fig. 8. Number of notifications (orange), and number of interactions (blue) with those notfications, per month. As developers become more familiar with ConE, they increasingly interact with its notifications.

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, then that action is also logged as telemetry (in Azure AppInsights).

From March to September of 2020, we logged 2,170 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 8, the number of interactions and the "clicks per notification" are clearly increasing as more and more people are getting 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.

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, then roughly how much effort do you estimate was saved as a result of finding out about the overlapping PRs? If not, then 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?
(4) What action (if any) will you likely take now that you know about the overlapping PRs?

Table 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%)

(5) Would you be interested in keeping using ConE that 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 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 6 shows the distribution of the feedback: 93.75% of the respondents indicated their interest and willingness to use ConE.

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 the other two 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."*

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.

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 toward 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 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

Table 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%)

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.

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.

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 4.2) and conditions to flag conflicting changes that causes unintended consequences. We offer three configuration parameters (see Section 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.

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 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.

8 CONCLUSION AND FUTURE WORK

In this article, 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.

REFERENCES

[1] Paola Accioly, Paulo Borba, and Guilherme Cavalcanti. 2018. Understanding semi-structured merge conflict characteristics in open-source Java projects. *Empir. Softw. Eng.* 23, 4 (Aug. 2018), 2051–2085. https://doi.org/10.1007/s10664-017-9586-1

[2] Iftekhar Ahmed, Caius Brindescu, Umme Ayda Mannan, Carlos Jensen, and Anita Sarma. 2017. An empirical examination of the relationship between code smells and merge conflicts. In *Proceedings of the ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM'17)*. 58–67. https://doi.org/10.1109/ESEM.2017.12

[3] Usman Ashraf, Christoph Mayr-Dorn, and Alexander Egyed. 2019. Mining cross-task artifact dependencies from developer interactions. In *Proceedings of the IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER'19)*. 186–196. https://doi.org/10.1109/SANER.2019.8667990

[4] Azure Batch Accessed 2020. Azure Batch. Retrieved from https://azure.microsoft.com/en-us/services/batch/.

[5] Azure Devops Accessed 2020. Azure DevOps. Retrieved from https://azure.microsoft.com/en-us/services/devops/?nav=min.

[6] Steve Berczuk and Brad Appleton. 2002. *Software Configuration Management Patterns: Effective Teamwork, Practical Integration*. Addison-Wesley, Boston.

[7] Jacob T. Biehl, Mary Czerwinski, Greg Smith, and George G. Robertson. 2007. FASTDash: A visual dashboard for fostering awareness in software teams. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI'07)*. ACM, New York, NY, 1313–1322. https://doi.org/10.1145/1240624.1240823

[8] Christian Bird, Peter C. Rigby, Earl T. Barr, David J. Hamilton, Daniel M. German, and Prem Devanbu. 2009. The promises and perils of mining git. In *Proceedings of the 6th IEEE International Working Conference on Mining Software Repositories*. 1–10. https://doi.org/10.1109/MSR.2009.5069475

[9] Christian Bird and Thomas Zimmermann. 2012. Assessing the value of branches with what-if analysis. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE'12)*. ACM, New York, NY, Article 45, 11 pages. https://doi.org/10.1145/2393596.2393648

[10] Caius Brindescu, Iftekhar Ahmed, Carlos Jensen, and Anita Sarma. 2019. An empirical investigation into merge conflicts and their effect on software quality. *Empir. Softw. Eng.* 25 (Sep. 2019). https://doi.org/10.1007/s10664-019-09735-4

[11] Caius Brindescu, Iftekhar Ahmed, Rafael Leano, and Anita Sarma. 2020. Planning for untangling: Predicting the difficulty of merge conflicts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE'20)*. 801–811. https://doi.org/10.1145/3377811.3380344

[12] Yuriy Brun, Reid Holmes, Michael Ernst, and David Notkin. 2011. Proactive detection of collaboration conflicts. In *Proceedings of the 19th ACM SIGSOFT Symposium on Foundations of Software Engineering (SIGSOFT/FSE'11)*. 168–178. https://doi.org/10.1145/2025113.2025139

[13] Yuriy Brun, Reid Holmes, Michael D. Ernst, and David Notkin. 2011. Proactive detection of collaboration conflicts. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE'11)*. ACM, New York, NY, 168–178. https://doi.org/10.1145/2025113.2025139

[14] Yuriy Brun, Reid Holmes, Michael D. Ernst, and David Notkin. 2011. Proactive detection of collaboration conflicts. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE'11)*. ACM, New York, NY, 168–178. https://doi.org/10.1145/2025113.2025139

[15] Yuriy Brun, Reid Holmes, Michael D. Ernst, and David Notkin. 2013. Early detection of collaboration conflicts and risks. *IEEE Trans. Softw. Eng.* 39, 10 (Oct. 2013), 1358–1375. https://doi.org/10.1109/TSE.2013.28

[16] Marcelo Cataldo, Patrick A. Wagstrom, James D. Herbsleb, and Kathleen M. Carley. 2006. Identification of coordination requirements: Implications for the design of collaboration and awareness tools. In *Proceedings of the 20th Anniversary Conference on Computer Supported Cooperative Work (CSCW'06)*. ACM, New York, NY, 353–362. https://doi.org/10.1145/1180875.1180929

[17] Catarina Costa, Jose Figueiredo, Gleiph Ghiotto lima de Menezes, and Leonardo Murta. 2014. Characterizing the problem of developers' assignment for merging branches. *Int. J. Softw. Eng. Knowl. Eng.* 24 (Dec. 2014), 1489–1508. https://doi.org/10.1142/S0218194014400166

[18] Catarina Costa, Jair Figueiredo, Leonardo Murta, and Anita Sarma. 2016. TIPMerge: Recommending experts for integrating changes across branches. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'16)*. ACM, New York, NY, 523–534. https://doi.org/10.1145/2950290.2950339

[19] Cleidson R. B. de Souza, David Redmiles, and Paul Dourish. 2003. "Breaking the Code": Moving between private and public work in collaborative software development. In *Proceedings of the International ACM SIGGROUP Conference on Supporting Group Work (GROUP'03)*. ACM, New York, NY, 105–114. https://doi.org/10.1145/958160.958177

[20] Cleidson R. B. de Souza, David Redmiles, and Paul Dourish. 2003. "Breaking the Code": Moving between private and public work in collaborative software development. In *Proceedings of the 2003 International ACM SIGGROUP Conference on Supporting Group Work (GROUP'03)*. ACM, New York, NY, 105–114. https://doi.org/10.1145/958160.958177

[21] Klissiomara Dias, Paulo Borba, and Marcos Barreto. 2020. Understanding predictive factors for merge conflicts. *Info. Softw. Technol.* 121 (May 2020), 106256. https://doi.org/10.1016/j.infsof.2020.106256

[22] H. Christian Estler, Martin Nordio, Carlo A. Furia, and Bertrand Meyer. 2014. Awareness and merge conflicts in distributed software development. In *Proceedings of the IEEE 9th International Conference on Global Software Engineering*. 26–35. https://doi.org/10.1109/ICGSE.2014.17

[23] Martin Fowler. Accessed 2020. Semantic Conflict. Retrieved from https://martinfowler.com/bliki/SemanticConflict.html.

[24] GitHub Accessed 2020. GitHub. Retrieved from https://github.com/about.

[25] Georgios Gousios, Martin Pinzger, and Arie van Deursen. 2014. An exploratory study of the pull-based software development model. In *Proceedings of the International Conference on Software Engineering (ICSE'14)*, Pankaj Jalote, Lionel C. Briand, and André van der Hoek (Eds.). ACM, 345–355. Retrieved from http://dblp.uni-trier.de/db/conf/icse/icse2014.html#GousiosPD14.

[26] Rebecca E. Grinter. 1995. Using a configuration management tool to coordinate software development. In *Proceedings of Conference on Organizational Computing Systems (COCS'95)*. ACM, New York, NY, 168–177. https://doi.org/10.1145/224019.224036

[27] Mário Luís Guimarães and António Rito Silva. 2012. Improving early detection of software merge conflicts. In *Proceedings of the 34th International Conference on Software Engineering (ICSE'12)*. 342–352. https://doi.org/10.1109/ICSE.2012.6227180

[28] Anja Guzzi, Alberto Bacchelli, Yann Riche, and Arie van Deursen. 2015. Supporting Developers' Coordination in the IDE. In *Proceedings of the 18th ACM Conference on Computer Supported Cooperative Work and Social Computing (CSCW'15)*. ACM, New York, NY, 518–532. https://doi.org/10.1145/2675133.2675177

[29] Susan Horwitz, Jan Prins, and Thomas Reps. 1989. Integrating noninterfering versions of programs. *ACM Trans. Program. Lang. Syst.* 11, 3 (July 1989), 345–387. https://doi.org/10.1145/65979.65980

[30] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M. German, and Daniela Damian. 2014. The promises and perils of mining github. In *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR'14)*. ACM, New York, NY, 92–101. https://doi.org/10.1145/2597073.2597074

[31] Bakhtiar Khan Kasi and Anita Sarma. 2013. Cassandra: Proactive conflict minimization through optimized task scheduling. In *Proceedings of the International Conference on Software Engineering (ICSE'13)*. IEEE Press, 732–741.

[32] Bakhtiar Khan Kasi and Anita Sarma. 2013. Cassandra: Proactive conflict minimization through optimized task scheduling. In *Proceedings of the 35th International Conference on Software Engineering (ICSE'13)*. 732–741. https://doi.org/10.1109/ICSE.2013.6606619

[33] Sunghun Kim, Thomas Zimmermann, Kai Pan, and E. James Jr. Whitehead. 2006. Automatic identification of bug-introducing changes. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*. 81–90. https://doi.org/10.1109/ASE.2006.23

[34] Michele Lanza, Marco D'Ambros, Alberto Bacchelli, Lile Hattori, and Francesco Rigotti. 2013. Manhattan: Supporting real-time visual team activity awareness. In *Proceedings of the 21st International Conference on Program Comprehension (ICPC'13)*. 207–210. https://doi.org/10.1109/ICPC.2013.6613849

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

[36] Shane McKee, Nicholas Nelson, Anita Sarma, and Danny Dig. 2017. Software practitioner perspectives on merge conflicts and resolutions. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME'17)*. 467–478. https://doi.org/10.1109/ICSME.2017.53

[37] Microsoft. 2020. Microsoft Facts. Retrieved from https://news.microsoft.com/facts-about-microsoft/#EmploymentInfo.

[38] Nicholas Nelson, Caius Brindescu, Shane McKee, Anita Sarma, and Danny Dig. 2019. The life-cycle of merge conflicts: Processes, barriers, and strategies. *Empir. Softw. Eng.* 24 (Feb. 2019). https://doi.org/10.1007/s10664-018-9674-x

[39] Antti Nieminen. 2012. Real-time collaborative resolving of merge conflicts. In *Proceedings of the 8th International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom'12)*. 540–543. https://doi.org/10.4108/icst.collaboratecom.2012.250435

[40] Moein Owhadi-Kareshk, Sarah Nadi, and Julia Rubin. 2019. Predicting merge conflicts in collaborative software development. In *Proceedings of the ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM'19)*. 1–11. https://doi.org/10.1109/ESEM.2019.8870173

[41] Marc J. Rochkind. 1975. The source code control system. *IEEE Trans. Softw. Eng.* 1, 4 (Mar. 1975), 364–370. https://doi.org/10.1109/TSE.1975.6312866

[42] Anita Sarma, Zahra Noroozi, and Andre van der Hoek. 2003. Palantír: Raising awareness among configuration management workspaces. In *Proceedings of the 25th ACM/IEEE International Conference on Software Engineering (ICSE'03)*. IEEE, 444–454. https://doi.org/10.1109/ICSE.2003.1201222

[43] Harald Steck. 2011. Item popularity and recommendation accuracy. In *Proceedings of the 5th ACM Conference on Recommender Systems (RecSys'11)*. ACM, New York, NY, 125–132. https://doi.org/10.1145/2043932.2043957

[44] Song Wang, Chetan Bansal, Nachiappan Nagappan, and Adithya Abraham Philip. 2019. Leveraging change intents for characterizing and identifying large-review-effort changes. In *Proceedings of the 15th International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE'19)*. ACM, New York, NY, 46–55. https://doi.org/10.1145/3345629.3345635

[45] Chadd Williams and Jaime Spacco. 2008. SZZ revisited: Verifying when changes induce fixes. In *Proceedings of the Workshop on Defects in Large Software Systems (DEFECTS'08)*. ACM, New York, NY, 32–36. https://doi.org/10.1145/1390817.1390826

[46] Titus Winters, Tom Manshreck, and Hyrum Wright. 2020. *Software Engineering at Google: Lessons Learned from Programming Over Time*. O'Reilly Media.

[47] Feng Zhang, Foutse Khomh, Ying Zou, and Ahmed E. Hassan. 2012. An empirical study of the effect of file editing patterns on software quality. In *Proceedings of the 19th Working Conference on Reverse Engineering*. 456–465. https://doi.org/10.1109/WCRE.2012.55

Received January 2021; revised May 2021; accepted July 2021