Fault-Insertion and Fault-Fixing: Analysing Developer Activity over Time

David Bowes
Lancaster University, UK
d.h.bowes@lancaster.ac.uk

Giuseppe Destefanis
Brunel University, UK
giuseppe.destefanis@brunel.ac.uk

Tracy Hall
Lancaster University, UK
tracy.hall@lancaster.ac.uk

Jean Petric
Lancaster University, UK
j.petric@lancaster.ac.uk

Marco Ortu
University of Cagliari, Italy
marco.ortu@diee.unica.it

ABSTRACT

Developers inevitably make human errors while coding. These errors can lead to faults in code, some of which may result in system failures. It is important to reduce the faults inserted by developers as well as fix any that slip through. To investigate the fault insertion and fault fixing activities of developers. We identify developers who insert and fix faults, ask whether code topic 'experts' insert fewer faults, and experts fix more faults and whether patterns of insertion and fixing change over time. We perform a time-based analysis of developer activity on six Apache projects using Latent Dirichlet Allocation (LDA), Network Analysis and Topic Modelling. We show that: the majority of the projects we analysed have developers who dominate in the insertion and fixing of faults; Faults are less likely to be inserted by developers with code topic expertise; Different projects have different patterns of fault inserting and fixing over time. We recommend that projects identify the code topic expertise of developers and use expertise information to inform the assignment of project work. We propose a preliminary analytics dashboard of data to enable projects to track fault insertion and fixing over time. This dashboard should help projects to identify any anomalous insertion and fixing activity.

CCS CONCEPTS

• **Software and its engineering → Programming teams**;

KEYWORDS

mining software repositories, faults fixing, networks, software development

ACM Reference Format:
David Bowes, Giuseppe Destefanis, Tracy Hall, Jean Petric, and Marco Ortu. 2020. Fault-Insertion and Fault-Fixing: Analysing Developer Activity over Time . In *Proceedings of the 16th ACM International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE '20), November 8–9, 2020, Virtual, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3416508.3417117

PROMISE '20, November 8–9, 2020, Virtual, USA

ACM ISBN 978-1-4503-8127-7/20/11…$15.00
https://doi.org/10.1145/3416508.3417117

1 INTRODUCTION

Software code remains predominantly a handmade product, produced by human developers, and as such, it is prone to error. The result of this developer error can be faults in code and as the world demands ever larger and more complex software systems, controlling faults in code becomes more difficult but increasingly necessary. Understanding fault insertion and fault fixing is crucial to enabling the effective reduction of faults in software systems.

Previous studies have looked at a variety of aspects of fault insertion and fixing, however, this previous work is fragmented, with individual studies looking at elements of insertion and fixing in isolation. Previous studies focus on analysing fault fixing for a variety of potential uses. Developer experience has previously been investigated with the aim of matching developers to job vacancies (e.g. [13]), to identifying who should review code (e.g. [24]) as well as to enable effective bug triaging (e.g. [28]).

Developer experience is reported to be related to fault insertion [9]. Measuring developer experience is not straightforward with conflicting reports of whether time spent coding is a valuable metric. Studies increasingly suggest that additional context information must be considered alongside time spent coding [8].

Expertise seems an important enabler to reduced fault insertion and improved fault fixing. Expertise has been studied in software engineering with Baltes & Diehl [1] recently developing a theory of expertise in software development.

The impact of developer code ownership [2] on fault insertion has been studied extensively. Low code ownership (i.e. code that has been touched by many different developers) is widely reported as more likely to be faulty than code with high ownership (e.g. [2, 12]).

Most previous studies consider only snapshots of developer fault insertion and fixing. Very few account for the impact project experience over time is likely to have on developer fault inserting and fixing. Kini & Tosun [19] is an exception to this, using developer experience metrics over time to improve defect prediction models.

In this study, we look across a variety of aspects of developer fault insertion and fixing in an attempt to identify specific patterns of activity for particular systems. We consider the intensity of fault insertion and fixing by developers to highlight influential fault inserters and fixers in projects. We simulate the coding expertise of developers by analysing the code topics with which developers have most experience. We investigate the relationship between these code topics and the fault insertions and fixes made by individual developers. We also analyse fault insertion and fixing within

the context of code complexity, over time, to establish whether developers' fault insertion and fixing changes as their experience over time increases.

Our study takes a multi dimensional approach to understanding (I) fault insertion and fixing by considering the impact of developer expertise via familiarity with code topics, (II) developer insertion and fixing over time and (III) the complexity of the code being touched by developers during fault insertion and fixing over time. Our study attempts to pull together and build on previous research to understand more comprehensively fault insertion and fault fixing. We move towards building a more comprehensive time-based understanding that could help to enable better organised software teams who are more able to effectively deploy developers to minimise shipped faults and also underpin the development of tools to support the minimisation of faults during development. We suggest a dashboard of data for projects which may help to identify anomalies to a project's normal fault insertion and fixing activities.

Our study analyses the repositories of six Apache Github projects. We provide a replication package of our analysis containing a full set of scripts and raw data[1]. We aim to understand fault insertion and fixing by answering the following research questions:

RQ1: Can we identify those developers most likely to insert and fix faults in code? If we can identify who is most likely to insert faults, it may become easier to manage the deployment of developers effectively. Similarly, if we can identify who is likely to fix faults, assigning tasks to developers could become easier. We find in each project examples of developers who are very active in all activities as well as developers who seem to predominately insert faults and also developers who predominately fix faults inserted by other developers.

RQ2: Does expertise impact developers' fault insertion and fixing? We try to understand whether it is important that developers have expertise in the code that they touch. We analyse whether developers with topic expertise insert and fix faults. We suspect that it is likely that developers with topic expertise insert fewer faults and make more fixes than developers without expertise in the code topic. We find that faults appear to be inserted by developers with low expertise in the code topic of the fault. We also find that fault fixers have slightly more expertise in the topic of the fault, but less expertise than we expected.

RQ3: Does experience over time on projects impact developers' fault insertion and fixing? Developers' experience changes over time and it is likely that developers' fault insertion and fixing also changes as time goes on. Understanding the relationship between experience over time on the project and fault insertion and fixing will help to deploy tasks to developers in line with their project experience. To mitigate the impact of increasing code complexity over time, we analyse the complexity of files touched by developers. We find that there is a complex pattern of developer activity over time with no clear patterns of fault insertion and fixing across the projects studied. Similarly the evolution of code complexity varies across projects.

The rest of the paper is structured as follows: Section 2 summarises previous related work on fault insertion and fixing. Section 3 details the methodology of our analyses and is followed by Section 4 which presents the results of our analyses in response to the research questions we pose. In Section 5 we describe a preliminary expertise dashboard for future development. Section 6 outlines the threats to validity of our study. Section 7 concludes the paper and suggests future work.

[1] https://bitbucket.org/giuseppedestefanis/promise2020

2 BACKGROUND & RELATED WORK

The fault insertion and fixing behaviours of developers have been investigated for a variety of purposes using a range of methods and measurements. We summarise this previous work.

Many previous studies use code ownership to describe the familiarity developers have with units of code. Code ownership is often used to indirectly measure developer expertise. Mockus and Herbsleb [17] were some of the first to measure the frequency with which developers work with specific pieces of code and to associate code expertise with this measure.

Code ownership has also been analysed in terms of faults inserted into code. Matsumoto et al. [15] reported that code touched by many different developers was more likely to be faulty. Bird et al. [2] used code authorship metrics to identify the developer who originated problems in code and also to identify developers to whom fault fixes should be assigned. Bird et al. divided developers into two groups: Minor Developers (those who have contributed less than 5% of code in a component) and Major Developers (those who have contributed more than 5% of code in a component). Bird et al. report that faulty code is more likely to have been written by Minor developers.

Bird et al.'s findings were further supported in Greiler et al.'s [12] replication study and Foucault et al.'s [10] larger study of code authorship in open source systems. Businge et al. [6] also report a similar relationship between authorship and faults in Android applications. Overall, there seems to be growing empirical evidence that authors who are actively involved with a piece of code insert fewer faults into that code. Fritz et al.'s [11] model of code base knowledge confirms the importance of code authorship. Fritz et al.'s experimental study suggests that developers have more knowledge about code that they author. Fritz et al. show a direct link between effort spent by developers on code and knowledge about that code.

More recently Wang et al.'s [27] preliminary work used Latent Dirichlet Allocation (LDA) modelling [4] to identify the expertise of developers. Wang et al. automatically measured developer expertise based on code quantity, code quality, skills and contribution; embedding this understanding in an on-line tool. Wang et al. are among the few previous studies that take into account a variety of factors when measuring the quality of code produced by developers, including faults inserted, vulnerabilities introduced, code complexity and code smells introduced. Wang et al.'s study is preliminary work based on a small sample of data.

Developer expertise is likely to be influential to fault insertion and fault fixing. Measuring expertise directly is challenging as it is an abstract and multi dimensional concept [17]. Previous studies have resorted to a range of diverse indirect measures of developer expertise. Constantine and Kapitsaki [7] proposed an approach to analysing development activity to identify developer expertise. Constantine and Kapitsaki tracked the continuity of code contributions made by approximately 150 active Github developers to

understand the development of programming language expertise across Github projects in relation to the size of projects. Length of project participation is the most common proxy for measuring expertise and is used particularly in studies of OSS (e.g. Vasilescu et al., [26]). Recent studies suggest that the impact that length of project participation has on productivity and quality [8] and expertise [1] is not conclusive.

A more sophisticated understanding of developer expertise now seems to be emerging with Baltes & Diehl [1] recently developing a theory of software development expertise. Specific aspects of expertise have also recently been investigated. Dieste et al. [8] investigated the relationship between years of programming experience and programmer performance. Their quasi experiments with 56 students and 70 professional developers revealed that years of industry experience did not directly influence programmer performance. Other task specific skills were more influential to programmer performance (e.g. skills in specific frameworks). Vasilescu et al. [26] report that a combination of knowledge, perspectives and experience are good predictors of productivity and project success.

Developer use of specific tools and techniques has also been reported as an indirect measure of expertise. Montandon et al. [18] analysed library and framework use by Github developers to identify evidence of expertise. Montandon et al. reported that expertise was related to intensity of coding activity (i.e. low expertise is related to few contributions to projects). Montandon et al. triangulated their findings using other sources (e.g. Linkedin) to identify Github developers with high levels of expertise.

High developer turnover is also reported to increase code faults. This is because overall knowledge of the code base diminishes as developers leave. Foucault et al.'s [10] study of five large projects suggested that new developers lack project expertise and have different activity levels because of their reduced code-base knowledge. Rigby et al. [20] further confirms the relationship between high turnover and lower code knowledge. Foucault et al. [10] report that projects with high developer turnover exhibit lower productivity levels and higher numbers of faults.

In this study we investigate coding authorship by developers over time, not only in terms of the intensity of fault insertion, but also fault fixing, taking into account their sustained contributions and topic expertise in a particular project. We contextualise this developer activity by considering the complexity of code that developers are working with. We go further than the snapshot analysis predominately used previously by analysing changes in developer fault insertion and fixing over time.

3 METHODS

3.1 Open Source Projects Analysed

We selected six open source systems detailed in Table 1. All projects were selected from the *Apache* community. We chose *Apache* projects as they follow the same development guidelines which reduces the variability that arises when analysing datasets from different sources. In addition, the projects use the *git* versioning system and *JIRA* fault database which are linked using the *JIRA* ID. This makes it possible to extract faults in code. We selected the projects with the highest number of commits, comments, and issues. We extracted data for each project from the beginning of the repository found on Github until February 2020.

The first column of Table 1 is the project name. The *# commits* column shows the total number of commits for the project at the time we collected the data. The *#FI* and *#FF* columns represent the total number of fault insertion and fault fix commits, respectively. The same fault can be fixed in multiple commits which is why some projects have more fault fix than fault insertion commits. The last two columns represent the total number of project contributors throughout the project's history. In *derby*, *hadoop hdfs* and *hadoop common* the number of authors and committers is the same suggesting that all authors are allowed to contribute to the project.

Table 1: Summary of the six *Apache* projects

Project	# commits	# FI	# FF	# bugs	# commiters	# authors	first commit	last commit
hadoop hdfs	1134	856	927	817	25	25	2009-05-19	2011-06-12
camel	44020	19607	12446	6623	213	673	2007-03-19	2019-12-18
derby	8269	4235	5889	3267	37	37	2005-01-24	2019-08-18
hadoop common	10509	5314	2408	2070	83	83	2009-05-19	2014-08-22
hive	14247	11060	13104	12290	120	324	2008-09-09	2020-01-14
hbase	17424	12580	15023	22133	138	458	2007-04-19	2020-01-11

3.2 Data Extraction

We extracted a range of data from each project's *Github* and *JIRA* repositories. From *Github* we obtained the following information: *commit hash*, *commit author*, *commit date*. The data was collected for each commit on the *master* branch throughout each of the project's history. To collect the *Github* data we used a script which is provided in the replication package. From *JIRA* we obtained the following information: *fault ids*, *fault titles*, *fault description*, *fault comments* and *fault report dates*. Similar to *Github* data, we obtained all the fault reports throughout the projects' history. The script for collecting the *JIRA* data is available in the replication package.

We used the SZZ algorithm to extract fault fix and fault insertion *git* commits from the six systems [23]. The SZZ algorithm finds a link between a fault report and the corresponding fix *git* commits in the version control system. From the fix commit, the SZZ algorithm then identifies which code snippets were faulty and tracks those back to their insertion points. The SZZ algorithm has been widely used in previous studies (e.g.[21]). From fault fix and insertion data we extracted the *developer*, *git hashes* and *files* involved in the fault fix/insertion. In addition, information about *author* and *committer* for each git commit was collected. We used *author*/*committer* information to attribute each change to the *author* of a commit, rather than the *committer*. This is because only *committers* have the right to commit changes but may do this on behalf of other authors.

Finally, we collected cyclomatic complexity metrics for all fault insertion and fault fix commits. We used JHawk[2] to collect this data. Metrics were collected at class level. We recorded the *commit hash* next to each class for which the metrics were collected. We then linked these *commit hashes* to the developers who authored the code changes. This data enabled us to investigate how code complexity changed over time for individual developers.

The data collected enables us to identify who inserted a fault, who fixed that fault and the complexity of the files being changed at insertion and fix.

[2] http://www.virtualmachinery.com/jhawkprod.htm

3.3 Data Cleaning and Analysis

We performed a series of cleaning steps on our data. First, we merged all developer activity representing the same developer into one set. As several developers were contributing to a project using slightly different names or email address. We then ensured that all data was anonymised. All identifiable information about developers such as their names and emails were replaced. We also associated the numbers of fault insertion and fault fix commits to each developer. Finally, we used the data about *authors* and *committers* of *git commits* to correctly attribute changes made by developers. For all of our extracted data we ensured that the metrics are mapped to the *author* of a commit, rather than the *committer*.

We used the following techniques to analyse our data. Latent Dirichlet Allocation (LDA) was used for cluster analysis of issue topics as described in Section 3.5. We used the Gensim[3] package to perform the LDA analysis with Python. The `gephi` tool[4] was used for network analysis to identify the contributions of individual developers on a project and to visualise developer activity in terms of who introduces and fixes faults (as described in Section 3.4). Finally, we used static code metrics to demonstrate how the complexity of code that developers touch changes over time.

3.4 Network Analysis

We built developer network graphs for the six systems we analysed. These graphs represent the team of developers working on a system and the connections between developers. Such network graph structures have previously been used in various analysis of open source projects (e.g. [16] [25]).

For each system we generated a direct network graph showing insertion and fixing activity of developers. We built our developer network graph using Gephi, an interactive network visualization and exploration tool. Figure 1 provides an example network graph. Each developer is represented by a node and an edge between two nodes means that the two developers are interacting. If developer A fixes an issue generated by developer B, there is a direct link between node A and node B, with an in-link from A to B. The size of the nodes are proportional to the number of out-links (which represents the number of fixed issues by a developer) in the graphs showing fixing activities. In Figure 1 node A has two out-links, while node B has no out-links. The size of node A is bigger than the size of node B. If there is a link going from node A to node B, this means that developer A fixed an issue introduced by developer B. The "self-link" exiting from node A and entering node A indicates that developer A both introduced and fixed a fault.

We also computed the Betweenness Centrality network metric to obtain a deeper understanding of the developer network structure, and the interactions of developers in the project. Betweenness Centrality is a statistical property of a network used to find influential people in a social network. The Betweenness Centrality of a node is an indicator of its importance in the network and is defined as the number of shortest paths that pass through the node [5]. A node with higher Betweenness Centrality has an important role over the network, since more information will pass through that node.

[3]https://radimrehurek.com/gensim/
[4]https://gephi.org version 0.9.2

Figure 1: Example of network graph

3.5 Topic Modeling

We use topic modelling to identify the code topics in projects and to understand the topic 'expertise' of developers. Topic modelling involves using statistical models to automatically discover themes occurring within a corpus of text documents. The aim of topic modelling is to find a distribution of words in each topic and the distribution of topics in each document. A topic can be considered as a probability distribution over a collection of words, e.g. a topic relating to football is more likely to contain the words goal and offside than a topic relating to cricket. Since its introduction in 2003 [4], LDA has become a popular unsupervised learning technique for topic modelling. LDA assumes each document contains multiple topics to different extents. The generative process by which LDA assumes each document originates is described below:

Algorithm 1 LDA's Algorithm

Require:
1: Choose N $Poisson(\epsilon)$
2: Choose θ Dir(α)
for each: $N \in$ Words W_n
3: Choose a topic Multinomial(θ).
4: Choose a word W_n from $p(W_n|Z_n, \beta)$, a multinominal probability conditioned on the topic Z_n.

Considering each document, the number N of words to generate is chosen (1). The algorithm randomly chooses a distribution of words over the topics, σ (2). For each word to be generated in the document, the algorithm randomly chooses a topic, from the distribution of topics (3), and then, from the topic selected, chooses a word using the distribution of words in the topic (4). The algorithm focuses on the distribution of topic in document and the distribution of words in topic as variables. The aim is to find latent (hidden) parameters that can be estimated via inference for retrieval of per-document topic distributions and per-topic word distributions. We applied LDA to model developers' topic 'expertise' in issues (faults) considering the title and description as a textual representation of the issue - which is common in topic modeling [3, 14] applied to social content and user generated content. We aggregated issues by assignee (namely the developer assigned to the issue) allowing us to created a corpus of documents (where each document represents an issue) and applied LDA to obtain the high level topics on which developers are grouped based on the dominant topic of issues assigned to developers.

4 RESULTS

RQ1: Can we identify those developers most likely to insert and fix faults in code?

To put fault insertion and fault fixing into context we first looked at whether developers fixed faults that they themselves or other developers had inserted into the code. Figure 2 shows in blue the percentage of faults where the developer who inserted the fault and the developer who fixed the fault are the same, in orange the percentage of faults where the inserter and the fixer are different developers. Figure 2 shows that for five of the six projects between 40-60% of faults are fixed by developers who did not insert the fault. The Camel project is an outlier in much of our analysis, which we nevertheless include throughout to avoid appearing to cherry pick results, as well as to show that there are always a range of heterogeneous projects, each with their own proclivities.

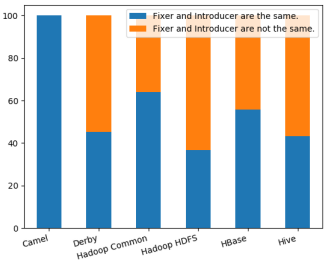


Figure 2: Fault Insertion Vs Fault Fixing. The orange bar represents the percentage of bugs where the developer that introduced the bug is not the same developer that fixed the issue and blue otherwise

To provide a more detailed understanding of the dynamics within each project's development community, and to identify any developers who are most actively inserting and fixing faults, we built a network of insertion and fixing activities and apply network analysis (as described in the previous section).

A developer active in a project, from the point of view of the network graphs we built, can perform one or more of the following actions over time:

- fix a previous fault they inserted;
- fix a fault inserted by an other developer.

Developer activity levels are proportionate to the number of out-degree of a node in the network. Nodes with higher value of out-degree indicate increased developer activity. Where a developer fixes a fault they inserted, a self-loop is added to the node in the network which represents that developer. When a developer fixes a fault inserted by another developer, a direct link will connect the two nodes representing the developers, with the arrow pointing at the developer who inserted the fault. Tables 2 and 3 provide examples of the underlying data related to the Hadoop HDFS and Derby directed graphs. Table 2 shows that developer 1 has the highest out-degree value (449) in the project. This means that developer 1 represents the biggest node for Hadoop HDFS (and it highlights that this developer performed the most fixes). Table 2 and Table 3 show that for Hadoop HDFS and Derby developer 1 also has the highest Betweenness Centrality value resulting in Node 1.

Table 2: Network Analysis (Hadoop HDFS)

Dev Id	Out-degree	In-degree	Degree	B-cent
1	449	176	625	56.25
2	188	116	304	15.46
3	167	158	325	11.85
4	165	112	277	8.83
5	133	20	153	0.03
6	118	60	178	1.15
7	61	32	93	1.06
8	38	313	351	34.4
9	15	12	27	0
10	14	65	79	0.29
11	12	56	68	0
12	9	83	92	3.69
13	7	71	78	0
14	4	13	17	0
15	3	39	42	0
16	3	18	21	0
17	0	42	42	0

Table 3: Network Analysis (Derby)

Dev Id	Out-degree	In-degree	Degree	B-cent
1	1469	492	1961	85.19
2	512	241	753	1.91
3	441	203	644	5.67
4	331	1307	1638	72.6
5	276	166	442	8.76
6	138	176	314	8.61
7	118	183	301	3.01
8	111	264	375	1.09
9	55	82	137	0.03
10	48	30	78	0
11	46	27	73	0
12	37	86	123	0.11
13	21	45	66	0
14	7	9	16	0.003
15	7	19	26	0
16	5	113	118	0.01
17	3	12	15	0
18	2	170	172	0
19	2	4	6	0

The networks data suggest that some developers predominately fix faults inserted by other developers. Data also shows that the most active fault fixing developers have a high number of self-loops, meaning that they introduce and fix their own faults. We previously identified a relatively high level of self-fixing in Figure 2. Each project has a small number of highly active developers who insert and fix many of their own and other developers' faults.

The results obtained from the directed graphs show that it is possible to identify specific types of developers in most of the six projects. We describe these types of developers as:

- Super-developers: most active in the project who insert and fix their own faults and those of other developers;
- Fixers: less active in the project who predominately fix faults inserted by other developers;
- Inserters: less active in the project who predominately fix their own faults.

A relatively large number of inserters and fixers seem active in most of the six projects. Whereas a small number of super-developers are active in all six projects. Each type of developer is important to identify as each type is likely to impact differently on project success. More stringent reviews of code contributed by inserters would probably benefit projects. Whereas more active use of fixers would also probably benefit projects. Super-developers are likely to have excellent knowledge of the project and could be deployed to more difficult tasks with less stringent code review. The structure of networks vary across the six projects, as it is possible to see in the network graphs provided in the repository containing the replication package[5]. This variation suggests slightly different

[5] https://bitbucket.org/giuseppedestefanis/promise2020

insertion and fixing activity across projects. Such variability is to be expected as most projects have specific ways of working and it is important to understand the normal patterns for each project so that anomalies can be identified quickly.

The analysis we provide in response to Research Question 1 is an aggregated picture of the entire time-frame we analysed for each project. It is likely that developer activity evolves over time as new developers join a project and gain experience. To investigate this evolution we present a time-based analysis of developer activity in response to Research Question 3.

RQ2: Does expertise impact developers' fault insertion and fixing?

Figure 3 presents the topic modelling clusters, considering all kind of issues (e.g. bugs, enhancements etc.), for the six projects and shows that for all projects there are distinct topic clusters. These are clusters of issues sharing similar content produced using LDA (as described in the previous section). This suggests that the issues on which developers work cover a range of different topics and that some of these topics are likely to benefit from specific expertise during development activities.

Figure 4 looks in more detail at the topics for the HBase project (similar detail for all projects is available in our replication package). The figure on the left represents the size and dimensional spacing among the ten topics, the dimension of the circle represents the number of issues belonging to that cluster. These circles show that there is little overlap between topics meaning that, in general, the topics are well defined for HBase. On the right of Figure 4 are the top 30 most important keywords of the first topic (represented by circle 1). We manually analysed each topic for each project to confirm that these topics make sense, representing for example topics such as "compute, thread, save etc" for concurrent issues or "connection, pool, jdbc, driver" for database related issues.

Clustering issues into topics allowed us to assign a *dominant topic* to each developer, i.e. to identify the most frequent topic in the issues that each developer has worked on. For example, if developer A worked most often on issues related to concurrency, developer A's dominant topic would be the topic containing keywords representing concurrency. In the remainder of this study we considered only issues related to fault (e.g. bugs) and link topics to faults by assigning a topic to the issue report for that fault. We then compute the percentage of issues where the fixer is an *expert*, i.e. the *dominant topic* of the fixer and the fault are the same. For example, a fault related to "concurrency" is reported and an issue assigned to a developer. The topic "concurrency" is assigned to the fault, we then compare the topic with the dominant topics of the fixer.

We analysed the expertise of developers who insert and fix faults (i.e. whether a developer's dominant topic matches that of the fault at insertion and fix). Figure 5 shows this expertise analysis for each project broken down into four quadrants (Q1: the fault inserter is an expert; Q2: the fault inserter is not an expert; Q3: the fault fixer is an expert; Q4: the fault fixer is not an expert). For four out of six projects, in most cases the developer who inserted the fault is not an expert in the fault topic (78% to 93% of faults are introduced by developers whose dominant expertise is not in the topic). This is an important finding as lack of developer expertise could be the cause of some of these faults. Camel has a different profile with only 6.5% faults introduced by non-experts. More research is needed to understand why Camel seems to be such an outlier project, but this may be related to the relatively high number of authors and committers, all of whom seem to fix their own faults.

Figure 5 also shows that the expertise of fault fixers is slightly better aligned to the topic of the fault. This alignment is not as strong as we might expect, but may be mitigated by the presence of 'super-developers'. Such developers are likely to have wide expertise of the project and so are able to tackle a range of fault topics rather than faults only related to their dominant topic.

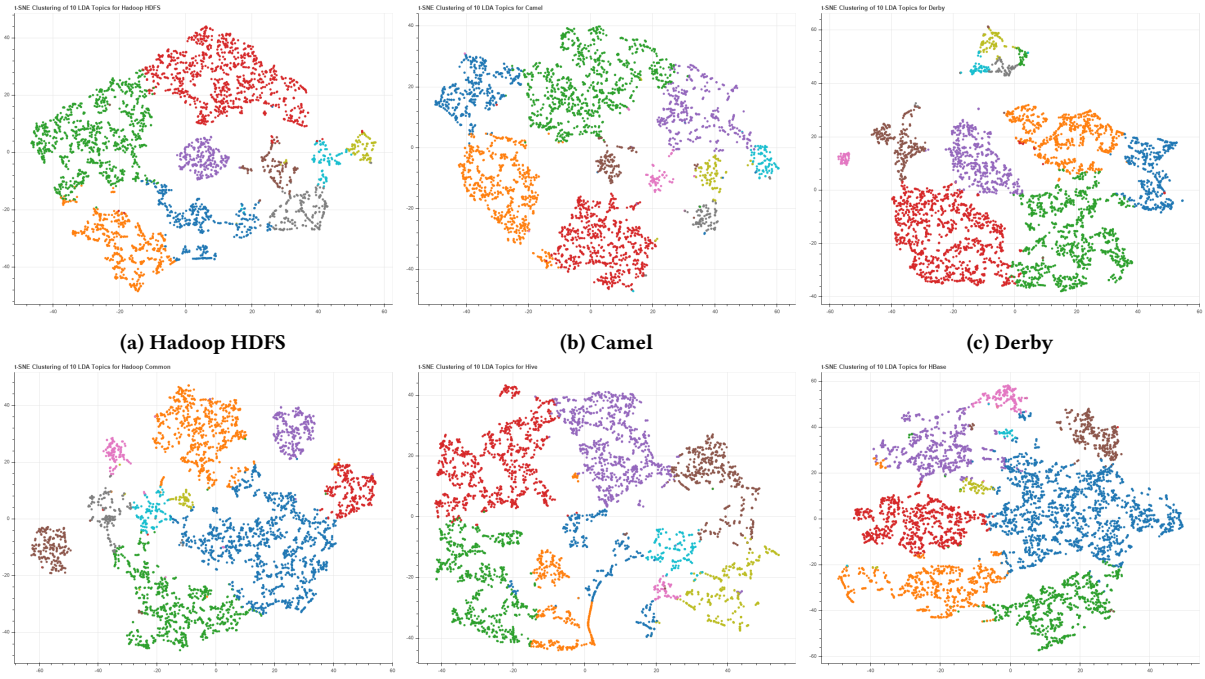
RQ3: Does experience over time on projects impact developers' fault insertion and fixing?

We analyse the impact of time on fault insertion and fixing. We investigate whether developers introduce fewer faults and fix more faults as time goes by. We also investigate whether the code tackled by developers gets more complex over time and developers gain more project experience and how code expertise changes during the lifetime of a project. We first analysed the activity levels of developers in relation to the complexity of files touched by developers over time. This analysis allowed us to identify how the complexity of the code developers were working on evolved in the context of activity intensity. Figure 7 plots the cyclomatic complexity of the files changed during a fault insertion or fix in each project over time by individual developers.

Figure 7 shows that each project has a unique pattern of developer activity over time. Such differences in projects are commonly reported in empirical investigations (e.g. [22]) and very few empirical studies are able to convincingly report findings that hold across all projects, even when the projects appear to have much in common. Figure 7 suggests that developer contributions over time vary between projects, with sustained project contributions from some developers and bursts of intense contributions from other developers. Some projects (e.g. HBase) seem to have many developers who come and go from the project. Other projects (e.g. Derby) seem to have fewer but more long lasting project developers. Patterns of developer retention and contribution intensity are likely to affect developer expertise and underpin patterns of fault insertion and fault fixing. However the exact relationship is difficult to understand given the complexity of activities across projects (in which Figure 7 provides some insight).

Figure 7 also shows that projects vary in terms of the evolution of code complexity. Some projects seem to increase in complexity as time goes on, e.g. Hive (figure 7e) appears to be stable for about 4 years during which time there are relatively few developers working on the system, after 2014, the number of developers increases and the files worked on become more complex. In other projects complexity seems to remain fairly stable, e.g. Hadoop Common. Whereas some projects start highly complex but steeply reduce in complexity over time (e.g. Derby).

Figure 8 shows the fault fixing and fault insertion activities by the 20 most active developers over time. The blue area shows the density of fault inserting changes and the pink shows the density of fault fixing changes. It is not possible for a fault fix to occur before a fault insertion, therefore it is expected that the density of



(a) Hadoop HDFS (b) Camel (c) Derby

(d) Hadoop Common (e) Hive (f) HBase

Figure 3: Topic Clusters for Issues

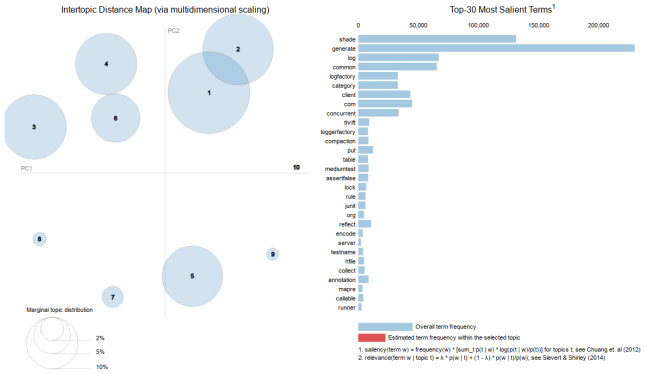


Figure 4: HBase Topic Clusters for Issues and Top 30 Keywords

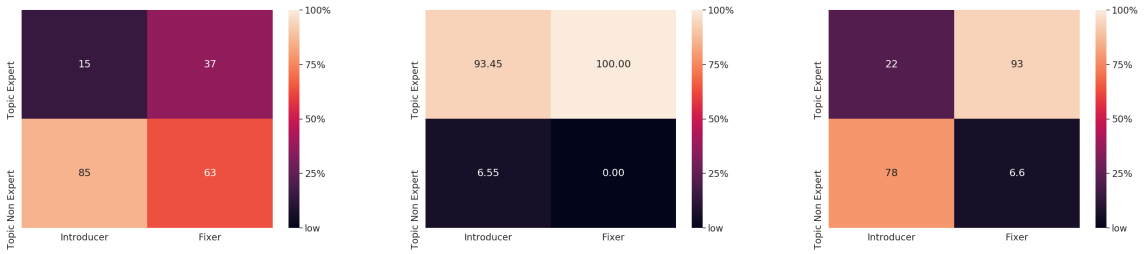
fault insertion activity appears before fault fixes. Figure 8 shows that peaks of fault insertion are followed by peaks of sustained fault fixing activity. For Derby and the Hadoop projects, there is a peak of fault insertion near the start of the project. Figure 8 also shows the activity of the most frequent committer to each project. The most active committers also show periods of fault insertion followed by fault fixing. The density of fault fixing for frequent committers seems more sustained over time compared to other developers. In most projects the introduction of faults by the most frequent committer drops as a proportion of faults fixed over time.

We also track expertise over time in relation to fault insertion and fixing. Figure 6 shows in red the number of commits, per month, from developers who inserted faults with a dominant topic different from the topic of the fault. In green is shown the number of commits, per month, of developers who fixed faults with a dominant topic matching the topic of the fault. Figure 6 suggests a pattern of fault insertion and fixing where bursts of non-expert activity occur throughout the lifetime of the project. Figure 6 also suggests that in most projects there is relatively high activity in non-expert fault insertion compared to fault fixing. Again, Camel is an outlier in this analysis showing a very different pattern of activity over time.

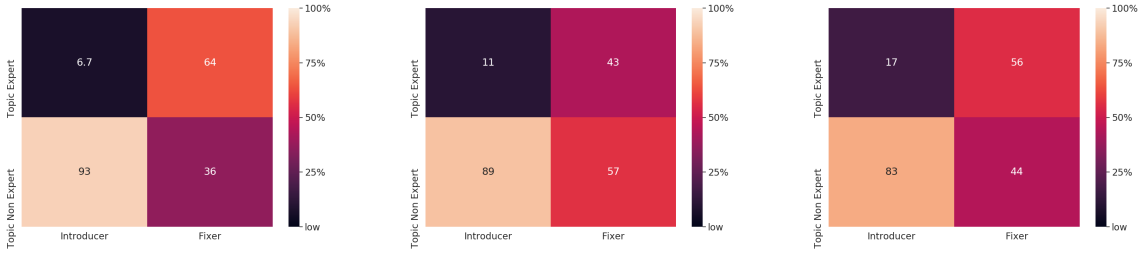
Monitoring fault insertion and fixing trends over time seems to provide useful insights into how projects are changing and when anomalous and potentially problematic activity is occurring. Monitoring 'curves' over time should allow projects to recognise potential problems and adapt their activities to ensure faults are reduced.

5 ANALYTICS DASHBOARD

Figure 9 presents a mock-up for a potential project dashboard which aggregates all the *fault analytics* presented in this study. The dashboard consists of two sections: *Topic Dashboard* and *Fault-Fix Dashboard*. The first section visualises the *dominant topic* of the issues that developers are working on. For example showing the number of commits over time whose author is an expert in the topic versus the number of commits over time whose author is **not** expert in the topic. Monitoring this data may help projects to quickly identify a situation where non-expert developers are *touching* code they probably should not.

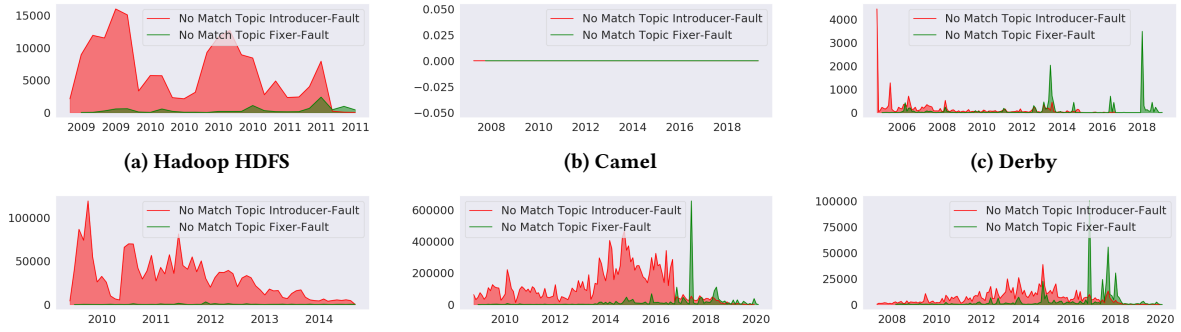


(a) Hadoop HDFS (b) Camel (c) Derby



(d) Hadoop Common (e) Hive (f) HBase

Figure 5: Fault Insertion Vs Fault Fixing when *inserter* and *fixer* are topic experts or not



(a) Hadoop HDFS (b) Camel (c) Derby

(d) Hadoop Common (e) Hive (f) HBase

Figure 6: Number of commits per month where *inserter* is not an expert in the fault topic (red) and the number of commits where the *fixer* is not an expert in the fault topic (green)

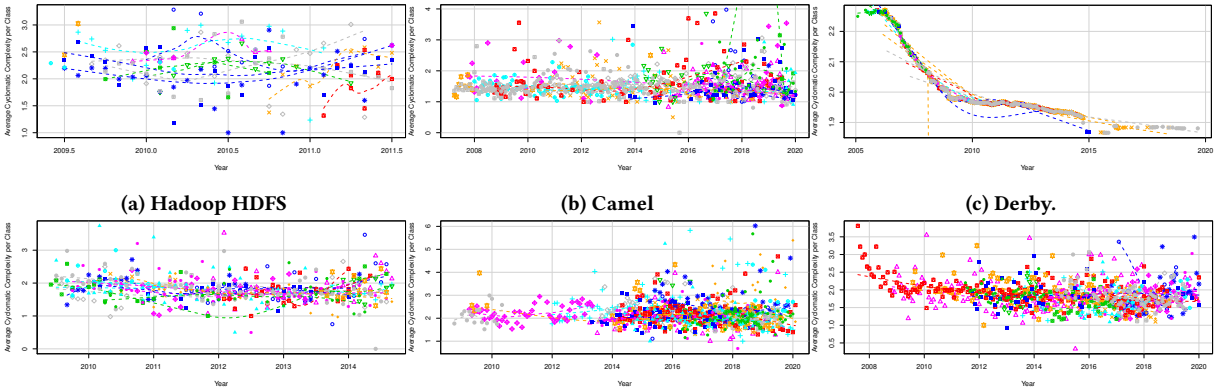
The second section of the *analytics dashboard* summarises for projects the information provided in this study about fault insertion and fixing over time. In this section is, for example, a time-series of fault insertion commits as shown in Figure 6. This data could enable project managers to visualise situations where, for instance, there is a growing number of fault insertions from developers with low expertise of the code topic. Observing such a situation could trigger, for example, stronger issue triaging in the project.

Overall such a dashboard could be useful for identifying systemic fault insertion and fixing problems in projects. Improved processes and tools can then be identified and the impact of their implementation tracked using the data in the dashboard. In addition, training and support requirements for individual developers could also be identified. Though clearly there are potential management challenges in using such a dashboard for the management of individuals. These management challenges go beyond the scope of this study.

6 THREATS TO VALIDITY & RELIABILITY

Internal Validity. Threats to internal validity concern confounding factors that can influence the obtained results. Based on empirical evidence, we assume a causal relationship between the topic modelling of developers and what they write in their discussion.

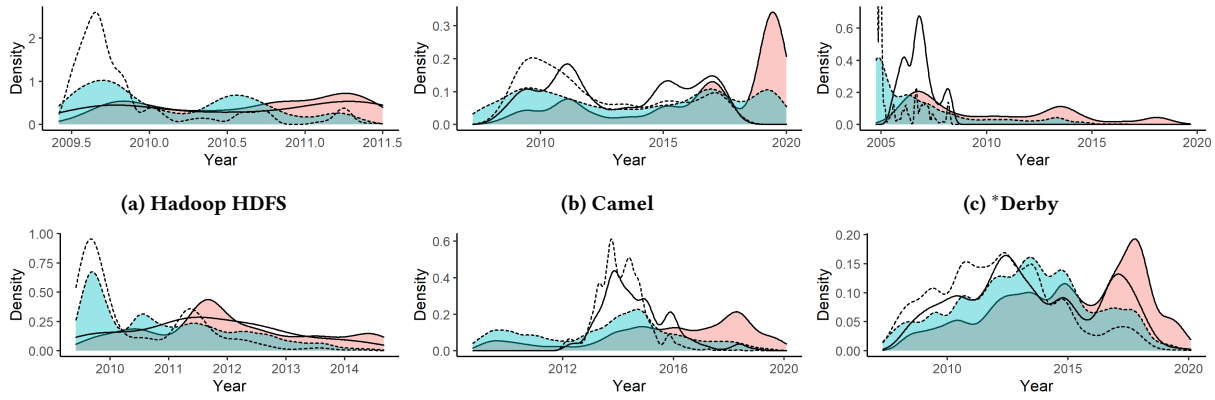
External Validity. Threats to external validity correspond to the generalisation of experimental results. In this study, we used several empirical approaches to evaluate the collaboration network of six projects from GitHub repositories. As the results of the *camel*



(a) Hadoop HDFS (b) Camel (c) Derby.

(d) Hadoop Common (e) Hive (f) HBase

Figure 7: Scatter-plots of average cyclomatic complexity of files touched. Each symbol represents a different developer. Each point represents the average complexity of a file being changed



(a) Hadoop HDFS (b) Camel (c) *Derby

(d) Hadoop Common (e) Hive (f) HBase

Figure 8: Plots of fault inserting and fixing activity of the 20 most active committers in each project. Dotted blue is for fault insertion and solid pink is for fault fixing (including the most active committer). Lines with no color below them are the density of the activity for the most active committer during the project. *Derby has a very active start, the most frequent committer reaches a peak density of 6 at the start of the project

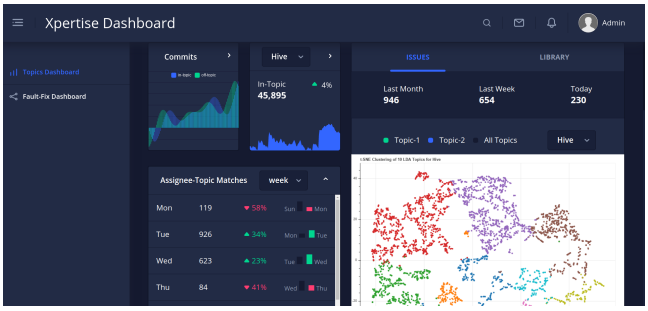


Figure 9: Analytics dashboard mock-up

project suggest, our approach may not always hold across all open source or close source projects. Projects from other open source communities may demonstrate different behaviours. Replications on commercial and other open source projects are needed to confirm or extend our results. We provide all scripts and data for other research to replicate this work.

Construct Validity. Developer expertise is a multi faceted challenging concept to measure. Numerous factors, such as project participation and use of libraries and frameworks, can proxy developer expertise. It is unlikely that a single factor exists to measure developer expertise. We considered the fault insertions and fixes of developers over time, as well as the complexity of the code they touch. We enhanced this data by considering experience working on specific code topics. We believe that, together, this data gives a reasonable approximation of developer expertise.

After visual inspection we assumed that the network graphs used for analysing developer activities are not random. This assumption was on the basis that all the nodes representing the developers in

networks are not fixing all the faults introduced by all the other nodes in the network.

7 CONCLUSIONS AND FUTURE WORK

We performed a multi–dimensional study to identify patterns of developer activity over time. We considered the fault insertion and fixing activity of developers, the familiarity of developers with code topics, alongside code complexity. Our time-based analysis was performed throughout the history of six projects on Github.

Our findings suggest that analysing developer activity over time provides insightful information about fault insertion and fixing. In each of the analysed projects we identify patterns that suggest that certain developers insert and fix more faults than others as well a developers who are highly active across the project. Our results also imply that developers who lack topic expertise are likely to insert more faults compared to those with more code topic expertise. Our results also suggest that developers who fix a fault have only marginally more expertise in the topic of the fault. The impact of code complexity on fault insertions and fixes over time is not clear.

We propose a project analytics dashboard to visualise and understand anomalies to a project's normal fault insertion and fixing activity. Such a dashboard could help to enable better organised teams who are more able to deploy developers to minimise faults and also underpin the deployment of tools to support the minimisation of faults during development activities.

We plan to extend our analysis to more Open source projects as well as to valid our findings on closed source systems. We intend to enhance our dashboard to provide an accessible tool to managing development activities for the reduction of faults.

ACKNOWLEDGEMENTS

This work is partly funded by grants from the UK's Engineering and Physical Sciences Research Council (EP/S005730/1 and EP/S005749/2).

REFERENCES

[1] Sebastian Baltes and Stephan Diehl. 2018. Towards a Theory of Software Development Expertise. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, NY, USA, 187–200.

[2] Christian Bird, Nachiappan Nagappan, Brendan Murphy, Harald Gall, and Premkumar Devanbu. 2011. Don't Touch My Code! Examining the Effects of Ownership on Software Quality. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*. ACM, NY, USA, 4–14.

[3] David M. Blei and John D. Lafferty. 2006. Dynamic Topic Models. In *Proceedings of the 23rd International Conference on Machine Learning*. ACM, NY, USA, 113–120.

[4] David M Blei, Andrew Y Ng, and Michael I Jordan. 2003. Latent dirichlet allocation. *Journal of machine Learning research* 3, Jan (2003), 993–1022.

[5] Ulrik Brandes. 2001. A faster algorithm for betweenness centrality. *Journal of mathematical sociology* 25, 2 (2001), 163–177.

[6] John Businge, Simon Kawuma, Engineer Bainomugisha, Foutse Khomh, and Evarist Nabaasa. 2017. Code Authorship and Fault-Proneness of Open-Source Android Applications: An Empirical Study. In *Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering*. ACM, NY, USA, 33–42.

[7] E. Constantinou and G. M. Kapitsaki. 2016. Identifying Developers' Expertise in Social Coding Platforms. In *2016 42th Euromicro Conference on Software Engineering and Advanced Applications*. IEEE, Limassol, 63–67.

[8] Oscar Dieste, Alejandrina M Aranda, Fernando Uyaguari, Burak Turhan, Ayse Tosun, Davide Fucci, Markku Oivo, and Natalia Juristo. 2017. Empirical evaluation of the effects of experience on code quality and programmer productivity: an exploratory study. *Empirical Software Engineering* 22, 5 (2017), 2457–2542.

[9] Jon Eyolfson, Lin Tan, and Patrick Lam. 2011. Do Time of Day and Developer Experience Affect Commit Bugginess?. In *Proceedings of the 8th Working Conference on Mining Software Repositories (MSR '11)*. Association for Computing Machinery, New York, NY, USA, 153–162. https://doi.org/10.1145/1985441.1985464

[10] Matthieu Foucault, Marc Palyart, Xavier Blanc, Gail C. Murphy, and Jean-Rémy Falleri. 2015. Impact of Developer Turnover on Quality in Open-Source Software. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, NY, USA, 829–841.

[11] Thomas Fritz, Gail C. Murphy, and Emily Hill. 2007. Does a Programmer's Activity Indicate Knowledge of Code?. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*. ACM, NY, USA, 341–350.

[12] Michaela Greiler, Kim Herzig, and Jacek Czerwonka. 2015. Code Ownership and Software Quality: A Replication Study. In *Proceedings of the 12th Working Conference on Mining Software Repositories*. IEEE, Florence, 2–12.

[13] Claudia Hauff and Georgios Gousios. 2015. Matching GitHub developer profiles to job advertisements. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. IEEE, Florence, 362–366.

[14] Young Bin Kim, Jurim Lee, Nuri Park, Jaegul Choo, Jong-Hyun Kim, and Chang Hun Kim. 2017. When Bitcoin encounters information in an online forum: Using text mining to analyse user opinions and predict value fluctuation. *PloS one* 12, 5 (2017), 1 –14.

[15] Shinsuke Matsumoto, Yasutaka Kamei, Akito Monden, Ken-ichi Matsumoto, and Masahide Nakamura. 2010. An Analysis of Developer Metrics for Fault Prediction. In *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*. ACM, NY, USA, 9.

[16] Andrew Meneely, Laurie Williams, Will Snipes, and Jason Osborne. 2008. Predicting failures with developer networks and social network analysis. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*. ACM, Atlanta, 13–23.

[17] Audris Mockus and James D Herbsleb. 2002. Expertise browser: a quantitative approach to identifying expertise. In *Proceedings of the 24th International Conference on Software Engineering*. IEEE, Orlando, 503–512.

[18] Joao Eduardo Montandon, Luciana Lourdes Silva, and Marco Tulio Valente. 2019. Identifying experts in software libraries and frameworks among GitHub users. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories*. IEEE, IEEE, Montreal, 276–287.

[19] S. Ozcan Kini and A. Tosun. 2018. Periodic Developer Metrics in Software Defect Prediction. In *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation*. IEEE, Madrid, 72–81.

[20] P. C. Rigby, Y. C. Zhu, S. M. Donadelli, and A. Mockus. 2016. Quantifying and Mitigating Turnover-Induced Knowledge Loss: Case Studies of Chrome and a Project at Avaya. In *2016 IEEE/ACM 38th International Conference on Software Engineering*. IEEE, Austin, 1006–1016.

[21] Gema Rodríguez-Pérez, Gregorio Robles, and Jesús M. González-Barahona. 2018. Reproducibility and credibility in empirical software engineering: A case study based on a systematic literature review of the use of the SZZ algorithm. *Information and Software Technology* 99 (2018), 164 – 176.

[22] Thomas Shippey, David Bowes, and Tracy Hall. 2019. Automatically identifying code features for software defect prediction: Using AST N-grams. *Information and Software Technology* 106 (2019), 142 – 160.

[23] Jacek Sliwerski, Thomas Zimmermann, and Andreas Zeller. 2005. When Do Changes Induce Fixes?. In *Proceedings of the 2005 International Workshop on Mining Software Repositories*. ACM, NY, USA, 1–5.

[24] P. Thongtanunam, C. Tantithamthavorn, R. G. Kula, N. Yoshida, H. Iida, and K. Matsumoto. 2015. Who should review my code? A file location-based code-reviewer recommendation approach for Modern Code Review. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, Montreal, 141–150.

[25] Ferdian Thung, Tegawende F Bissyande, David Lo, and Lingxiao Jiang. 2013. Network structure of social coding in github. In *2013 17th European conference on software maintenance and reengineering*. IEEE, Genova, 323–326.

[26] Bogdan Vasilescu, Daryl Posnett, Baishakhi Ray, Mark G.J. van den Brand, Alexander Serebrenik, Premkumar Devanbu, and Vladimir Filkov. 2015. Gender and Tenure Diversity in GitHub Teams. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*. ACM, NY, USA, 3789–3798.

[27] Jing Wang, Xiangxin Meng, Huimin Wang, and Hailong Sun. 2019. An Online Developer Profiling Tool Based on Analysis of GitLab Repositories. In *Computer Supported Cooperative Work and Social Computing*, Yuqing Sun, Tun Lu, Zhengtao Yu, Hongfei Fan, and Liping Gao (Eds.). Springer Singapore, Singapore, 408–417.

[28] Asmita Yadav, Sandeep Kumar Singh, and Jasjit S. Suri. 2019. Ranking of software developers based on expertise score for bug triaging. *Information and Software Technology* 112 (2019), 1 – 17.