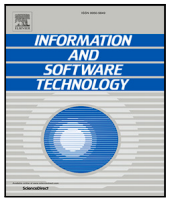
Information and Software Technology 158 (2023) 107187

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof



Fault-insertion and fault-fixing behavioural patterns in Apache Software Foundation Projects

Marco Ortu [a], Giuseppe Destefanis [b,*], Tracy Hall [c], David Bowes [c]

[a] University of Cagliari, Italy
[b] Brunel University London, UK
[c] Lancaster University, UK

ARTICLE INFO

Keywords:
Faults analysis
LDA
Mining software repositories

ABSTRACT

Background: 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.

Aim: 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.

Methods: We perform a time-based analysis of developer activity on twelve Apache projects using Latent Dirichlet Allocation (LDA), Network Analysis and Topic Modelling. We also build three models (using Petri-net, Markov Chain and Hawkes Processes) which describe and simulate developers' bug-introduction and fixing behaviour.

Results: 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.

Conclusions: We recommend that projects identify the code topic expertise of developers and use expertise information to inform the assignment of project work.

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. [1]), to identifying who should review code (e.g. [2]) as well as to enable effective bug triaging (e.g. [3,4]).

Developer experience has been reported as related to fault insertion [5] but measuring developer experience is not straightforward, with conflicting reports of whether time spent coding is a valuable experience metric (e.g., [6,7]). Eyolfson et al. [5], for example, considered experience as the amount of time since a developer's first commit to a project. Eyolfson et al. [5] reported that experience is related to fault insertion. 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 previously studied in software engineering with Baltes & Diehl [9] recently developing a theory of expertise in software development. The impact of developer code ownership [10] 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. [10,11]). Most previous studies consider only snapshots of developer fault insertion and fixing. Very few studies account for the impact project experience over time is likely to have on developer fault inserting and fixing. Kini & Tosun [12] is an exception to this, using developer experience metrics over time to improve defect prediction models.

* Corresponding author.
E-mail addresses: marco.ortu@unica.it (M. Ortu), giuseppe.destefanis@brunel.ac.uk (G. Destefanis), tracy.hall@lancaster.ac.uk (T. Hall), d.h.bowes@lancaster.ac.uk (D. Bowes).

https://doi.org/10.1016/j.infsof.2023.107187
Received 16 March 2022; Received in revised form 5 January 2023; Accepted 20 February 2023
Available online 24 February 2023

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.

This paper is an extended version of earlier work by the same authors [13]. We added six new systems to the original corpus analysed in [13] and one new research question (RQ4).

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.

Our study analyses the repositories of twelve Apache projects with data extracted from Github and Jira. We extracted fault fix and fault insertion git commits and collected information about code authors and committers. With the aim of investigating how code complexity changes over time, we also calculated the cyclomatic complexity for all fault insertion and fault commits and linked this information to the developers who authored the code changes. To represent the fault insertion and fault fixing process, we used a network analysis technique to build developer network graphs, and we used topic modelling to identify code topics from the issues submitted to each project's repository (for example, the topic 'Files and Resources' emerged from the Camel project) and understand the expertise of the developers in relation to those code topics. Finally, we built and evaluated three Stochastic models (Petri Nets, Markov chains and Hawkes models) to provide an overview of the activities performed by the developers.

Stochastic models are a way of representing a phenomenon which varies over time in a random manner. For example, the number of people crossing a bridge, number of faults fixed, the position of an aircraft during a flight, the number of packets travelling in a network. A stochastic process can be described by a set of random variables in which the sequence of values is a consequence of random factors. In building stochastic models it is necessary to start specifying the random variables and their associated probability density. Stochastic models describe the probabilistic law associated with the evolution over time of a physical phenomenon. In this work, we build stochastic models of the insertion and fixing behaviour of developers in our subset of Apache projects (simply speaking, our models provide a picture showing how a team is currently working). The goal of building such models is to understand the way developers behave, and to simulate their development activity over time. Having an understanding of, for example, what the probability is of a fault being fixed by a certain developer, can provide crucial information on the workload distribution among developers, and could help managers in taking informed decisions related to both the composition of a team and its productivity (for example redistributing more active developers into different teams).

We provide a replications package of our analysis containing a full set of scripts and raw data.[1]

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

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.

RQ4: Can we model developers' activities with Stochastic models? To try and understand the implications of our findings from RQ1, RQ2 and RQ3 for the future activities of projects we built a range of stochastic models. We built three stochastic models using Petri-nets, Markov chains and Hawkes processes so that we could understand project activities from a variety of points of view and to identify whether any of these three modelling techniques were particularly useful to gaining an understanding of project activities. All three models model developers' commit activities and make predictions on project behaviour of the developers in the future.

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. Section 5 discusses our results, while Section 6 outlines the threats to validity of our study. Section 7 concludes the paper and suggests future work.

2. Background

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 [14] 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. [10] 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 [11] replication study and Foucault et al.'s [16] larger study of code authorship in open source systems. Businge et al. [17] 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 [18] 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.

Rahman et al. [19] studied ownership and experience at the level of files and modules and found that implicated code is more strongly associated with a single developer's contribution. The results of the study also indicate that an author's specialised experience in the target file is more important than general experience. The authors suggested that quality control efforts could be profitably targeted at changes made by single developers with limited prior experience on that file.

Hokka et al. [20] analysed 500 C++ repositories to study correlations between developer experience and lambda use. The goal of the study is to understand whether the usage of lambdas correlates with programming experience. The results suggest that the developer experience positively correlates with lambda usage. Zhu et al. [21] studied how the authorship of code affects bug-fixing commits using the SStuBs dataset, a collection of single-statement bug fix changes in popular Java Maven projects. The authors studied the differences in characteristics between simple bug fixes by the original author (the developer who submitted the bug-inducing commit), and by different developers, i.e., non-authors. The results shows that 44.3% of simple bugs are fixed by a different developer. Fixes by the original author and by different developers differed qualitatively and quantitatively. The authors found that bug-fixing commits by authors tended to be larger in size and scope, and address multiple issues, whereas bug-fixing commits by other developers tended to be smaller and more focused on the bug itself.

More recently Wang et al.'s [22] preliminary work used Latent Dirichlet Allocation (LDA) modelling [23] 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 [14]. Previous studies have resorted to a range of diverse indirect measures of developer expertise. Constantine and Kapitsaki [24] 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. [25]). Recent studies suggest that the impact that length of project participation has on productivity and quality [8] and expertise [9] is not conclusive.

A more sophisticated understanding of developer expertise now seems to be emerging with Baltes & Diehl [9] 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. [25] 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. [26] 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 [16] 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. [27] further confirms the relationship between high turnover and lower code knowledge. Foucault et al. [16] 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 twelve 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, and also because we were interested in analysing **"The Apache Way"**,[2] defined as *the interpretation of one's experience with the Apache community-led development process*. As stated on the Apache website, *Apache projects and their communities are unique, diverse, and focused on the activities needed at a particular stage of the project's lifetime, including nurturing communities, developing great code, and building awareness.* The key elements of The Apache Way are the following:

- Earned Authority;
- Community of Peers;
- Open Communications;
- Consensus Decision Making;
- Responsible Oversight;
- Independence;
- Community Over Code;

In addition, the projects use the *git* versioning system and *JIRA* fault database which are linked using the *JIRA* ID. This allowed us to extract faults in code. Bissyandé et al. [28] have raised concerns about the quality of the process for linking bug reports and code changes since the links are missing for many software projects as the bug tracking and version control systems are often maintained separately. In order

[2] https://www.apache.org/theapacheway/index.html.

Table 1
Summary of the twelve *Apache* projects.

Project	# commits	# FI	# FF	# bugs	# committers	# authors	First commit	Last commit
hadoop hdfs	1 134	856	927	817	25	25	2009-05-19	2011-06-12
camel	44 020	19 607	12 446	6 623	213	673	2007-03-19	2019-12-18
derby	8 269	4 235	5 889	3 267	37	37	2005-01-24	2019-08-18
hadoop common	10 509	5 314	2 408	2 070	83	83	2009-05-19	2014-08-22
hive	14 247	11 060	13 104	12 290	120	324	2008-09-09	2020-01-14
hbase	17 424	12 580	15 023	22 133	138	458	2007-04-19	2020-01-11
Commons CLI	1 005	15	30	32	47	57	2002-06-10	2021-04-19
Commons CSV	1 556	33	62	63	30	43	2005-12-17	2021-04-19
Commons Math	6 622	25	60	92	69	79	2003-05-12	2021-04-13
Commons Codec	2 117	13	28	47	44	49	2003-04-25	2021-04-22
Commons JXPath	601	3	6	14	25	27	2001-08-23	2020-05-26
Commons Collections	3 569	54	137	66	68	95	2001-04-14	2021-04-26

to avoid possible issues with the linkage, we manually analysed 500 random commits from the projects, verifying that each commit message contains the issue key (e.g., "MATH-1541: Loop early exit.") which allows us to verify that the association issue/commit is correct.

We extracted data for each project from the beginning of the repository found on Github until April 2021.

The first column of Table 1 is the project name. The *# bugs* represents the reported bugs. 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.

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 BugVis [29] to extract fault fix and fault insertion *git* commits from the twelve systems. BugVis implements the SZZ algorithm [30] for initially linking fault reports to fault fixes and backtracking to identify the fault insertion commits. 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.[31]). The tool we used improved the original SZZ algorithm [30] by linking deleted as well as modified lines and using advanced diff commands which follow blocks of code being moved. These changes improved linkage from fault fix to insertion point [32]. BugVis is an interactive tool which allows developers/researchers to select individual lines and see where they came from and where they go to. The interactive ability of the tool allows us to investigate code which may be surrounding fault changes. 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 the cyclomatic complexity metrics for all fault insertion and fault fix commits. We used JHawk[3] 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.

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 addresses. We then ensured that all data was anonymised.

Alias resolution (developer matching and identity resolution) is a non-trivial problem well-studied in the literature [33–35]. In our study, we measured the percentage of aliases counting those developers with same name and surname and different email address. We found that the percentage of aliases was, on average, 8.86%, taking into account all the projects. We decided to merge those aliases considering that the real error would be lower than 8%, because some of the aliases are actually different people.

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[4] package to perform the LDA analysis with Python. The `gephi` tool[5] 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 twelve systems we analysed. These graphs represent the team of developers working on

[3] http://www.virtualmachinery.com/jhawkprod.htm.
[4] https://radimrehurek.com/gensim/.
[5] https://gephi.org version 0.9.2.

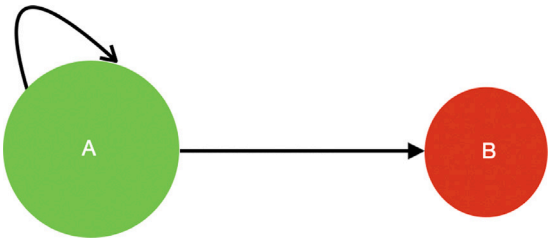


Fig. 1. Example of network graph.

a system and the connections between developers. Such network graph structures have previously been used in various analysis of open source projects (e.g. [36,37]).

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 visualisation and exploration tool. Fig. 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 Fig. 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 [38]. A node with higher Betweenness Centrality has an important role over the network, since more information will pass through that node.

3.5. Topic modelling

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 [23], 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 $N \sim Poisson(\epsilon)$
2: Choose $\theta \sim \text{Dir}(\alpha)$
for each: $N \in$ Words W_n
3: Choose a topic $\sim$ 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 modelling [39,40] applied to social content and user generated content. We aggregated issues by assignee (namely the developer assigned to the issue) allowing us to create a corpus of documents (where each document represents an issue) and to apply LDA to obtain the high level topics on which developers are grouped based on the dominant topic of issues assigned to developers. We chose 10 as the number of topics after applying a coherence model (using the U-Mass metric) [41] to all projects and obtaining 10 Topics as the best fit. The final value of 10 topics is a trade-off between the goodness of the coherence model and the interpretability of the topics. Higher number of topics would lead to less interpretable topics, while lower number would lead to overlapping topics.

3.6. Petri-nets

Petri Nets allow the modelling of dynamic systems and have been widely used in, for example, the modelling of manufacturing systems. The use of Petri Nets allows us to formally model the behaviour of individual developers' fault insertion and fixing in the wider context of the project in which they are working. Furthermore, Petri Nets can simulate the behaviour of the system, predicting expected behaviour in the long term (usually called Steady State Analysis) which allows us to understand, given the current state, development behaviour in the future.

A Petri-net PT (Place/Transition) is an oriented graph with two types of nodes, **places** and **transitions**, connected by direct arcs. The places are represented graphically by circles and the transitions by rectangles.

An arc can only join nodes of different types, so there can be arcs between places and transitions, but not between places or between transitions. A place from which an arc starts (to end to a transition) is called the transition **input place**; a place where an arc arrives (from a transition) is called the transition **output place**.

Places can contain several tokens. A distribution of tokens over all the places in the network is called **marking**. Transitions act on incoming tokens according to a rule, called **firing rule**. A transition is enabled if it can be triggered (e.g., if there are tokens in each input place). When a transition fires, it consumes tokens from its input places, performs tasks, and places a specified number of tokens in each of its output places. This happens automatically, for example, in a single non-preemptive step.

The execution of Petri nets is non-deterministic:

- if multiple transitions are enabled at the same time, any of them can be triggered;
- it is not guaranteed that an enabled transition is going to trigger. An enabled transition can be triggered immediately, after any waiting time (as long as it remains enabled), or not be triggered at all.

Since the triggering of a transition is not predictable *a priori*, Petri nets are suitable for modelling a concurrent system's behaviour. Petri nets have the following characteristics:

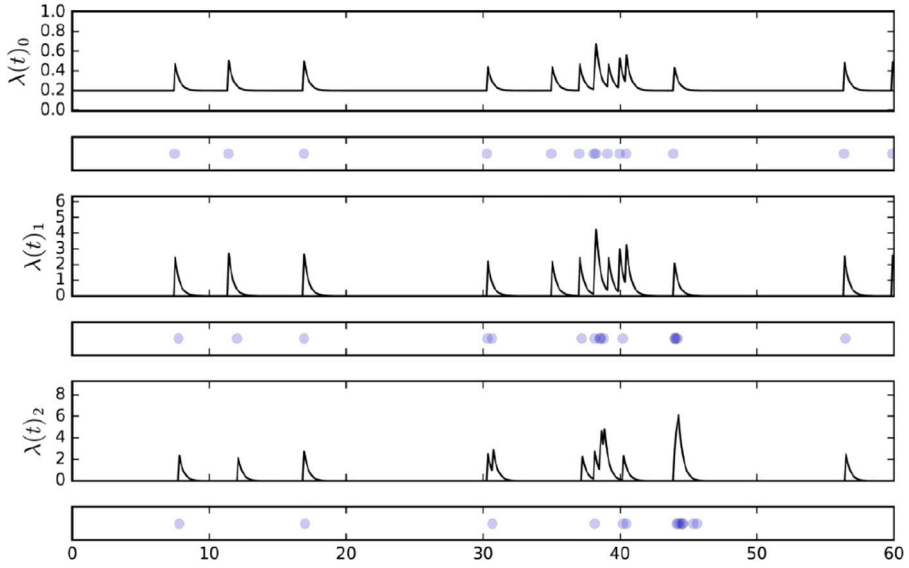


Fig. 2. Example of multivariate Hawkes process.

- they are easily modifiable (addition of other variables, modification of the set of values that can be assumed by one or more variables), without the need to start over and without "explosion" of complexity;
- they are modular, buildable by "assembling" submodels related to parts of the system;
- easy to interpret in terms of evolution of the state of the individual parts of the system (the state has a local meaning, being distributed in the network);
- they can represent infinite-state systems with a finite number of nodes of a graph.

Stochastic Petri nets are a form of Petri net where the transitions fire after a probabilistic delay determined by a random variable. We modelled the random variable that fires the transitions using the exponential random variable, which is modelled using the commits' activity of projects in our experiments. The probability density function (pdf) of an exponential distribution is showed in Eq. (1).

$$f(x;\lambda) = \lambda e^{-(\lambda x)} \tag{1}$$

A stochastic Petri net is a five-tuple $SPN = (P, T, F, M_0, \Delta)$ where:

- P is a set of states, called places.
- T is a set of transitions.
- F where $F \subset (P \times T) \bigcup (T \times P)$ is a set of flow relations called "arcs" between places and transitions (and between transitions and places).
- M_0 is the initial marking.
- Δ is the array of firing rates λ associated with the transitions. The firing rate, a random variable, can also be a function $\lambda(M)$ of the current marking.

3.7. Hawkes models

The Hawkes process is a point process class [42], also known as a self-exciting counting process, in which the impulse response function explicitly depends on past events [43]. In this type of process, the observation of an event causes the increase of the process impulse function. From a mathematical point of view, a point process is a Hawkes process if the impulse function $\lambda(t|H_t)$ of the process takes the form of (2).

$$\lambda(t|H_t) = \lambda_0(t) + \sum_{i:t_i<t} \phi(t - t_i) \tag{2}$$

In Eq. (2) H_t represents the history of given past events, $\lambda_0(t)$ is a positive function that determines the basic intensity of the process and ϕ is another positive function known as *memory kernel*, since it depends on past events occurred before time t. Hawkes models can be used to identify the dynamics of interactions between a group of K processes. The occurrence of an event on a particular process can cause an impulse response on that process (self-excitation), determining an increase of the likelihood of further events, and on other processes (mutual-excitation). Given a set of events occurring on a number of processes, a Hawkes model can be used to quantify previously hidden connections between the processes.

Fig. 2 illustrates an explanatory example of a multivariate Hawkes process with two flows of events: $\lambda(t)_0$ and $\lambda(t)_1$. In this example, the event flows have been constructed so that $\lambda(t)_0$ is not influenced by other flows, but only by events that happen in its own flow (self-exciting effect) otherwise, events in the same $\lambda(t)_0$ can have effects in $\lambda(t)_1$ (mutual-exciting effect).

3.8. Markov chain model

A Markov chain consists of X states and is a discrete-time stochastic process, a process that occurs in a series of time-steps in each of which a random choice is made. A Markov chain can be represented with graphs and/or matrices. The states can be represented with circles (nodes or vertices) and directed edges (links) connecting node i and node j if $p_{ij} > 0$.

From each node (or vertice) there is at least an out-coming edge: some nodes have links connecting to themselves, some nodes cannot be connected with each other, while some can be connected through bidirectional links. A probability p_{ij} is associated to each connection $i \rightarrow j$, defined as the transition probability from state i to j. The following properties need to be satisfied for p_{ij} values:

$$p_{ij} \in [0, 1], \qquad \sum_j p_{ij} = 1 \ \ \forall i$$

A (square) transition matrix **P** contains only positive elements, with sum equal to 1 for each row.

p_{11} indicates the probability of staying in the state 1, p_{12} indicates the probability of moving from state 1 to state 2 and p_{1n} indicates the probability of moving from state 1 to state n.

We built a MC modelling three different states:

- Development commit
- Fault Introduction commit
- Fault Fix commit

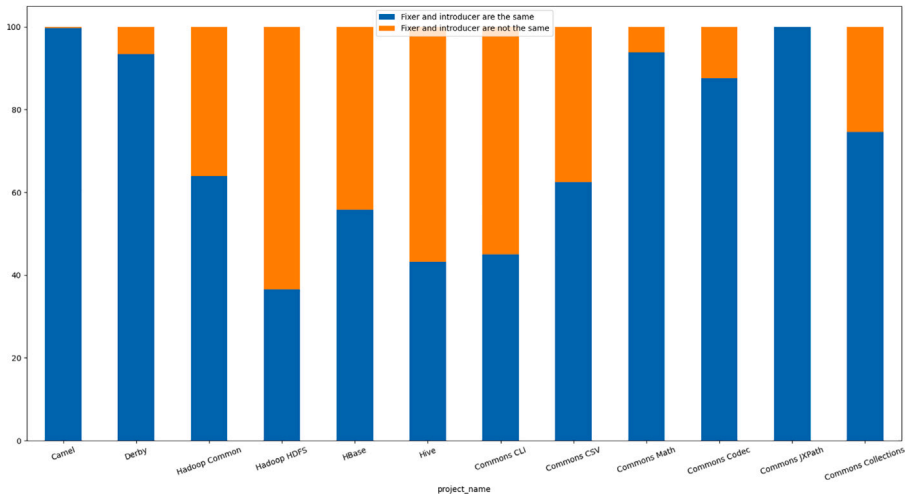


Fig. 3. Fault Insertion Vs. Fault fixing. The orange bar represents the percentage of bugs where the developer that introduced the bug is not the same developer that fixed the issue and blue otherwise.

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. Fig. 3 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. Fig. 3 shows that for six of the twelve projects between 40%–60% of faults are fixed by developers who did not insert the fault. The Camel and Commons JXPaths projects are outliers 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.

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.

Figs. 4–6 present directed graphs showing developers who are most actively inserting and fixing faults. Developer activity levels are proportionate to the size of nodes in the graphs. Nodes with bigger size 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. The colour of the nodes is related to the value of Betweenness Centrality, the darker the colour, the higher the value. 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 is represented by the biggest node for Hadoop HDFS in Fig. 4 showing that this developer performed the most fixes. Tables 2 and 3 show that for Hadoop HDFS and Derby developer 1 also has the highest Betweenness Centrality value resulting in Node 1 shown as darkest in both Figs. 4 and 5.

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 project networks shown in Figs. 4–6 suggest that some developers predominately fix faults inserted by other developers. Figs. 4–6 also

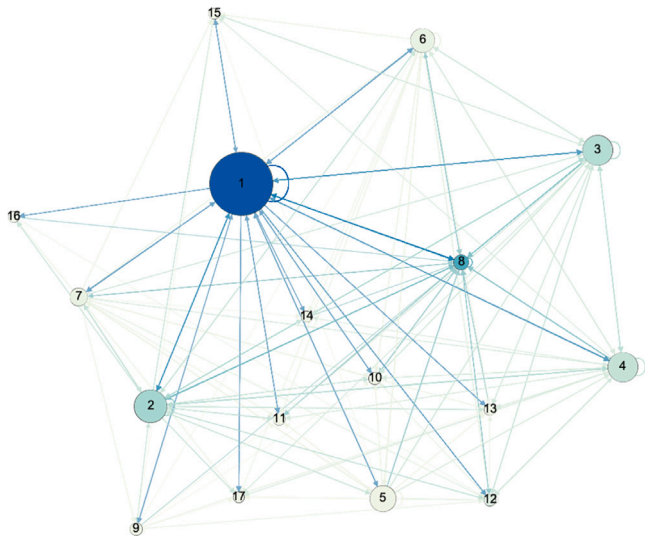


Fig. 4. Hadoop HDFS network.

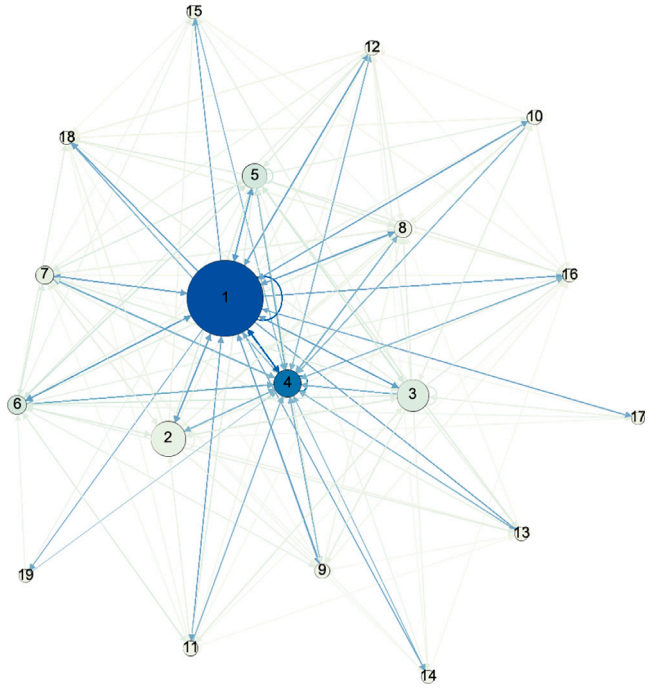


Fig. 5. Derby network.

show 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 Fig. 3. To explore further the activities of these self-fixing developers we built the networks both with and without self-loops and found minimal change in the network. Fig. 7 shows the network built for Derby without self-loops. Compared with Fig. 5 (containing self-loops) the structure of the Derby network remains the same, but the size of the nodes is reduced (as a smaller number of out-links are included). This suggests that the larger nodes in each network represents the most active developers in projects. Each project has a small number of highly active developers who 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 twelve 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 twelve projects. Whereas a small number of super-developers are active in all twelve 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. Fig. 8 shows an example of fixing in Derby. As the figure shows, the fix is not complicated, but in this case was performed by a super-developer. Having the possibility of understanding how to allocate the required fixes, would improve the productivity of the teams. The structure of networks vary across the twelve projects. This variation suggests slightly different 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?

Fig. 9 presents the topic modelling clusters, considering all kind of issues (e.g. bugs, enhancements etc.), for the twelve 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). We used the concept of document, which consists of a mixture of topics in different percentages. The notion of dominant topic is the following: ''the topic with the highest percentage for a given document''. Code topics are obtained aggregating together all issues assigned to a developer, in order to obtain a single text document. Therefore, topics are identified based on the types of issues that developers are assigned to. We then apply the LDA to all documents (one document per developer with all issues assigned to them) and we assign a dominant topic to each document (where a document represents a single developer) which is the *code topic*. We used the *t-distributed stochastic neighbour embedding* (T-SNE) to reduce the dimensionality of the topics and to be able to represent the clusters in two dimensions, the algorithm models the points so that close objects in the original space released close in the reduced-dimensional space, and distant objects far away, trying to preserve the local structure. 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.

Fig. 10 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 Fig. 10 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,

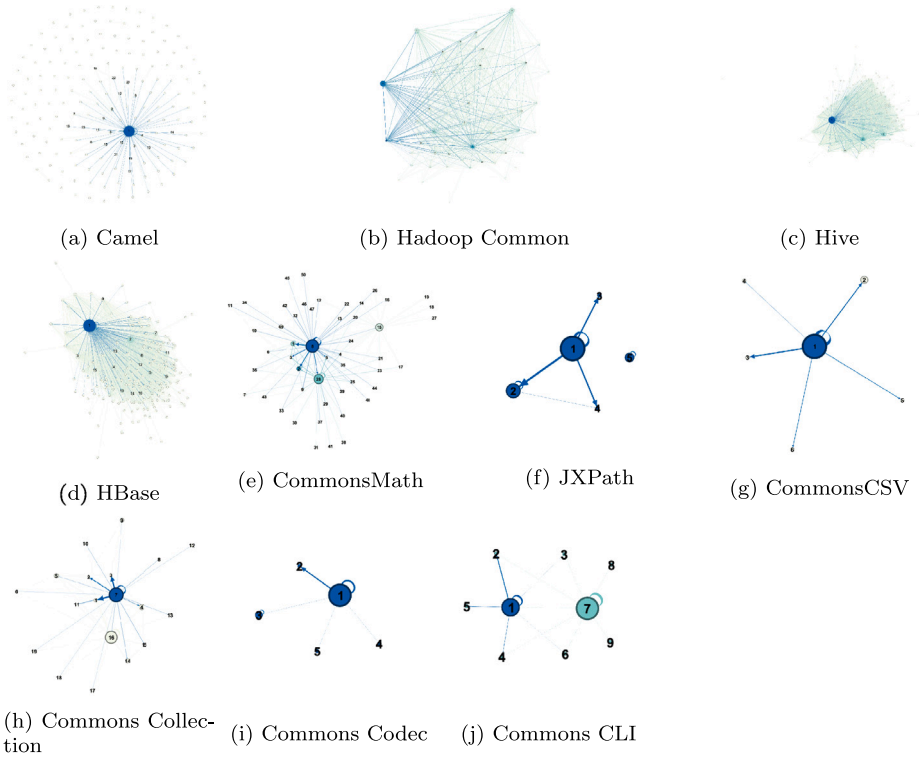


Fig. 6. Networks for all the remaining projects.

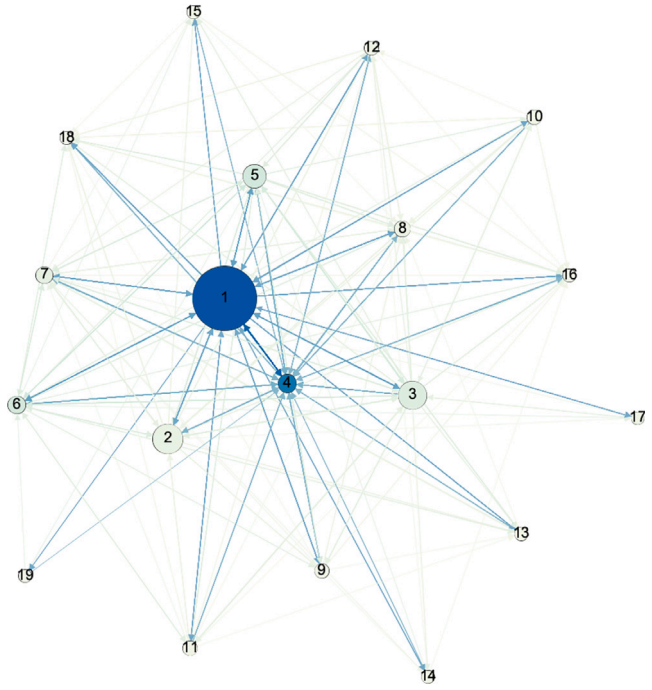


Fig. 7. Derby network (without self-loops).

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). Fig. 11 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 ten out of 12 projects, in most cases the developer who inserted the fault is not an expert in the fault topic (27% 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 and Commons JXPath seem to be such outlier projects, and to investigate further whether this result could be related to the relatively high number of authors and committers, all of whom seem to fix their own faults.

Fig. 11 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

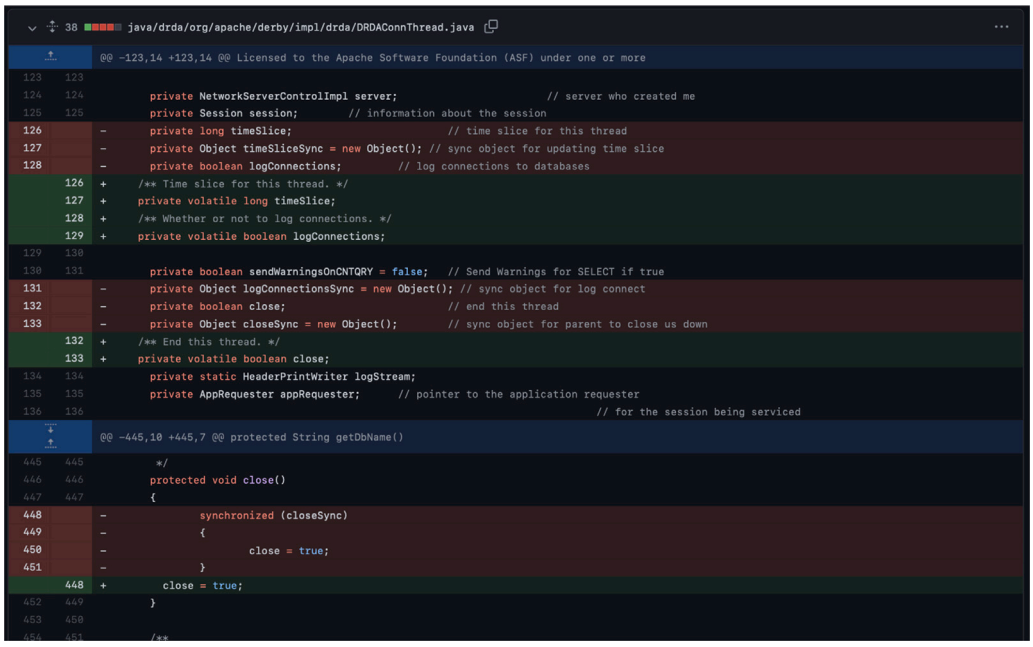


Fig. 8. Example of fix in Derby.

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. Fig. 12 plots the cyclomatic complexity of the files changed during a fault insertion or fix in each project over time by individual developers.

Fig. 12 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. [44]) 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. Fig. 12 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 Fig. 12 provides some insight).

Fig. 12 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 (Fig. 12(e)) 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).

Fig. 13 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 fault insertion activity appears before fault fixes. Fig. 13 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. Fig. 13 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.

RQ4: Can we model developers' activities with stochastic models?

We analysed fault insertion/fix behaviour during the development process using three different statistical models:

- Petri Nets
- Markov Chains
- Hawkes Processes

In the next subsection we provide the formal definition of the three models followed by a validation section where the models are applied to three different time windows. To validate our models, we considered three time windows of three months, each time window is further divided in two months for training our models and the last month of each time window is used for the validation of the models' prediction.

4.1. Petri nets

We modelled developers' activities using Stochastic Petri nets as shown in Fig. 14. This model represents a particular state of a developer's activity such as bug fixing or normal development activity and transitions are triggered based on commit activities.

- $P = [p_0, p_1, p_2, p_3]$.
- $T = [t_0, t_1, t_2, t_3, t_4, t_5, t_6]$.
- $M_0 = [1, 0, 0, 0]$ is the initial marking
- $\Delta = [\lambda_0, \lambda_1, \lambda_2, \lambda_3, \lambda_4, \lambda_5]$

The meaning of the places is:

- P_0: Represents the *Idle* state, the developer is waiting for a new activity.

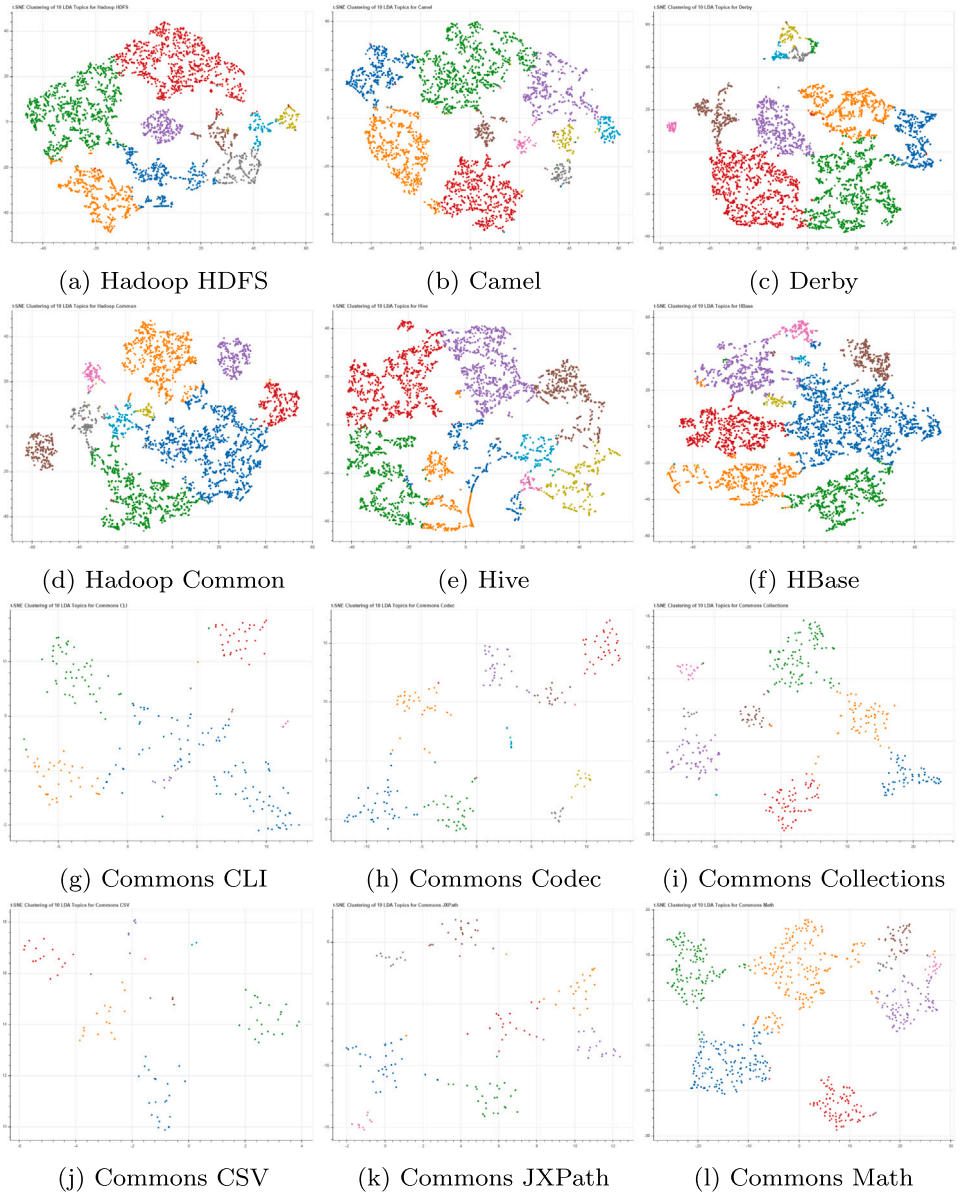


Fig. 9. Topic clusters for issues.

- P_1: Represents the *Development* state, the developer is working on a normal development activity.
- P_2: Represents the *Fault-Fixing* state, the developer is fixing a fault.
- P_3: Represents the *Fault-Insertion* state, the developer during normal development activity or fault-fixing activity has a probability to insert a new fault.

The transitions meaning derives directly from the places definition. In particular the transitions sequence $[t_0, t_1]$ represents a normal development activity. The transitions sequence $[t_2, t_3]$ represents the fault-fixing activity while the sequences $[t_0, t_4]$ and $[t_2, t_5]$ represents a fault insertion occurring during normal development activity or during a fault fix. Finally transition t_6 is fired right after t_4 or t_5 to go in *Idle* state where the developer is ready for a new activity.

We modelled each developer through the Δ array of λ parameters. These λ parameters have been empirically calculated from the commit activities. Table 4 shows the λ parameters evaluated for all 12 projects as an average of the developers' λ parameters. In particular, we considered t_6 as *direct transitions* to model the fact that a fault insertion is something that happens during a generic development activity, transaction λ_1 and λ_3 are respectively the complementary of λ_4, λ_3 (λ_4 = 1-λ_1 and λ_5 = 1-λ_3) as from place p_1 and p_2 we modelled the developer behaviour to be either going back to *idle* state (p_0) or introduce a bug (p_1).

Table 5 shows the result of the steady state analysis of the Petri's Nets for all the projects. We simulate the project's Petri Nets in order to obtain the steady state probabilities of the places which can be interpreted and the *average* time spent in a particular state over time.

Table 5 shows that the *idle* state (p_0) has the highest probability, this does not mean that most of the time developers are in an idle state, but reflect instead the behavioural model behind the Petri Net. In fact, transitions are modelled based on commit activities over times, and we are interested in capturing the different behaviours in terms of fault introduction/fixing against normal development activities showed by the steady state probabilities of *Fault-Fixing* state (p_2 place) and *Development* state (p_1 place). The probability of fault insertion and fault fix (p_2 and p_3) can be directly interpreted as an estimation of the fault fix effort, these probability values show a plethora of different behaviours across different projects. Table 5 refers to the whole project history

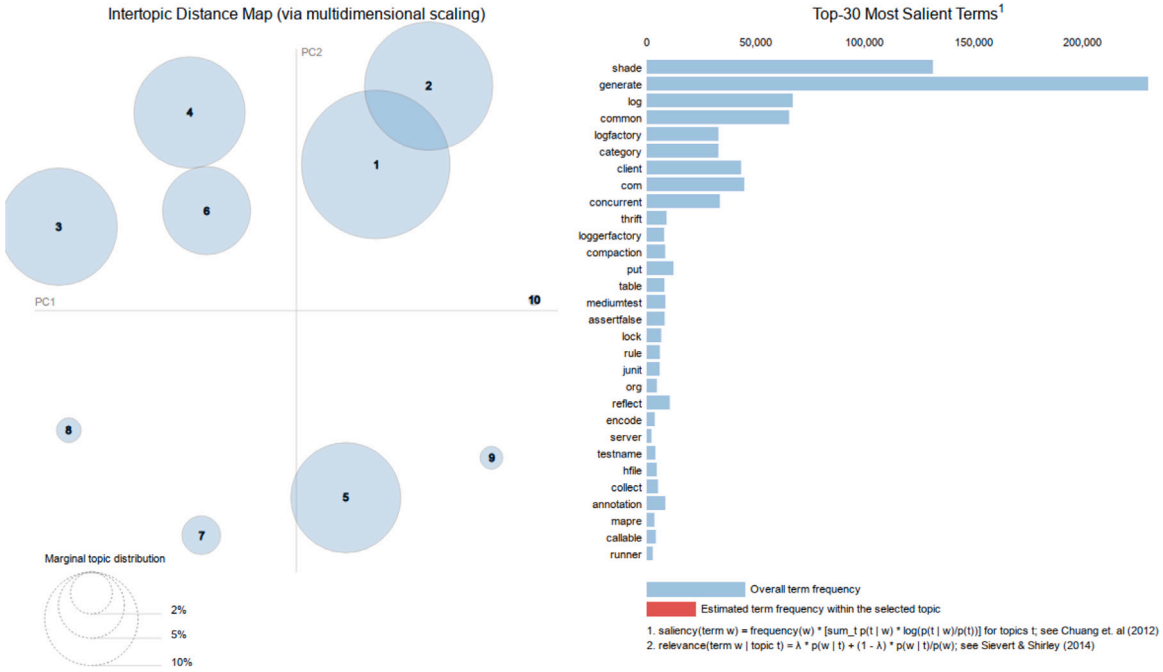


Fig. 10. HBase topic clusters for issues and top 30 keywords.

Table 4
Lambdas parameters for all 12 Apache projects.

Project	λ_0	$\lambda_4 - \lambda_5$	λ_2
Camel	0.479	0.354	0.167
Derby	0.133	0.364	0.503
Hadoop Common	0.348	0.428	0.224
Hadoop HDFS	0.003	0.496	0.500
HBase	0.064	0.299	0.637
Hive	0.068	0.344	0.589
Commons CLI	0.812	0.040	0.147
Commons CSV	0.675	0.073	0.252
Commons Math	0.901	0.016	0.083
Commons Codec	0.863	0.026	0.111
Commons JXPath	0.946	0.010	0.044
Commons Collections	0.676	0.084	0.240

Table 5
Places probabilities for all the projects.

Project	P_0	P_1	P_2	P_3
Camel	0.533	0.256	0.089	0.122
Derby	0.535	0.071	0.269	0.124
Hadoop Common	0.55	0.192	0.123	0.135
Hadoop HDFS	0.571	0.002	0.285	0.142
HBase	0.523	0.033	0.333	0.11
Hive	0.531	0.036	0.313	0.12
Commons CLI	0.501	0.407	0.074	0.019
Commons CSV	0.501	0.338	0.126	0.034
Commons Math	0.5	0.451	0.042	0.008
Commons Codec	0.5	0.432	0.056	0.013
Commons JXPath	0.5	0.437	0.022	0.005
Commons Collections	0.502	0.339	0.012	0.039

but in general the same analysis can be conducted at different levels, for example at single developer level or at a particular time window (such as pre-release or post-release) providing an estimation of fault introduction/fixing behaviour in the near future of the development system evolution.

The Petri Net model could be used by managers to evaluate the impact of the developers' activity while introducing and fixing faults, in particular giving an estimation of the percentage of time spent in fixing and introducing faults.

4.2. Markov Chain model

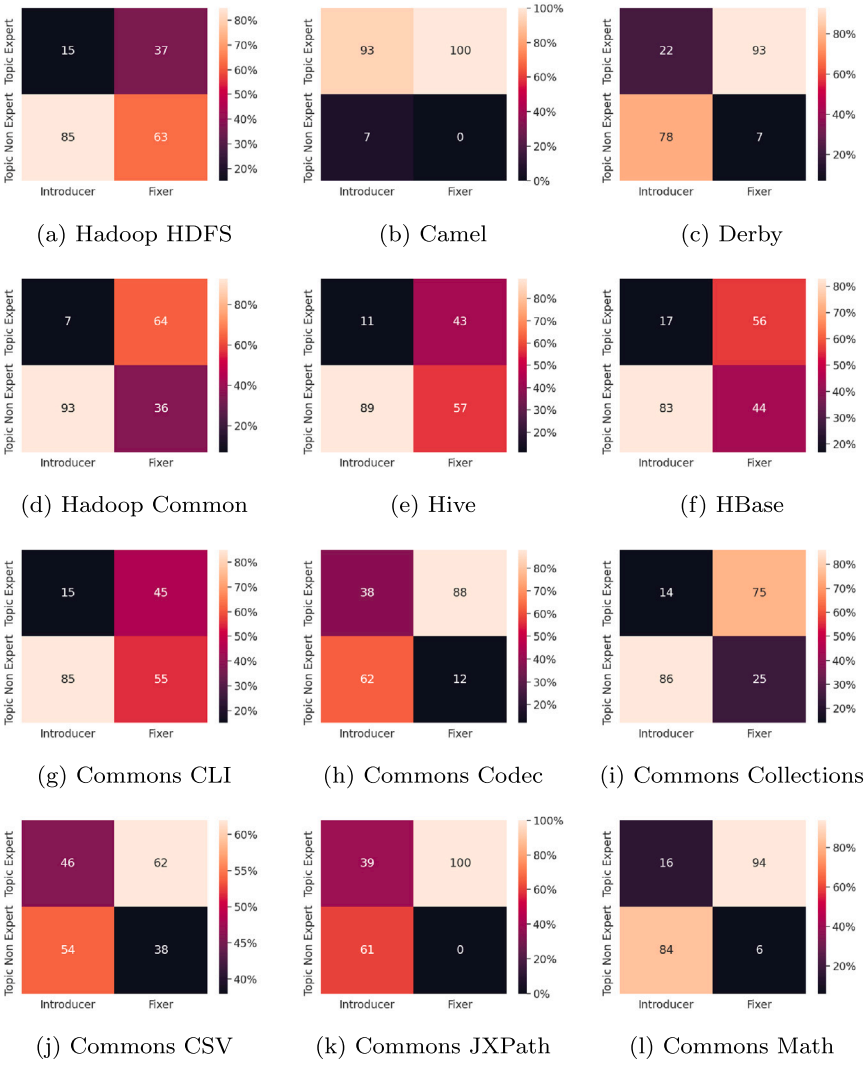
Markov Chains (MC) have been used to model behavioural aspects in social sciences [45,46]. Markov Chains provide a unique representation where the transitions between the different states of Sentiment, Politeness and Emotion, extracted from developers comments, can be modelled; it implicitly assumes "memorylessness" (through the Markov property) and this simplifies our analysis. Furthermore the models allow a straightforward determination of probability transitions between development, fault insertion and fix activities; this is crucial in determining how developer behaviour is influenced by other developers. We used Discrete Time Markov Chains considering the time series of commits, categorising each commit as:

- Normal Development Commit
- Fault Insertion Commit
- Fault Fix Commit

We then modelled the Markov Chain with three states: (i) Development (DEV) (ii) Fault Insertion (FI) (iii) Fault Fix (FF). The transition probability from one state to another, i.e. $DEV \rightarrow FI$, represents the probability that the next commit is a *Fault Insertion Commit* given that the current commit is a *Normal Development Commit*. Given the sequence of commits we evaluated the adjacency matrix for each project, this matrix can be directly represented by a directed graph. Fig. 15 represents the MC graphs for all considered project where the states are represented by ovals and transition by arrows.

These MCs are directly obtained using all data and represents the final model for each project and can be used to simulate the percentage of development, fault introduction and fix commits in the next n discrete events, which can be interpreted as an estimation of the development, fault introduction and fix effort.

In Section 4.4 we describe a real case scenario where these MCs are exploited to estimate the number of development, fault insertion and fix commits in the next n commits (which covers approximately a month of activities) and can be directly interpreted by project managers for decision support.



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

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

(g) Commons CLI (h) Commons Codec (i) Commons Collections

(j) Commons CSV (k) Commons JXPath (l) Commons Math

Fig. 11. Fault insertion vs. fault fixing when *inserter* and *fixer* are topic experts or not.

4.3. Hawkes Model

In order to build the Hawkes Model used in this study, we considered the following flows of events:

- $\lambda(t)_0$: as the number of faults inserted/fixed by top 5 developers.
- $\lambda(t)_1$: as the number of faults inserted/fixed by the rest of the developers.

To explain how we interpreted the model we introduce the following example which helps to understand how the Hakews' processes are capable of modelling the many events happening during the development process.

In our case we considered the interaction and relationship among top-5 fault inserters, top-5 fault fixers and the rest of developers. Hawkes processes help decipher the role of top-n fault inserters/fixers with respect to the other developers.

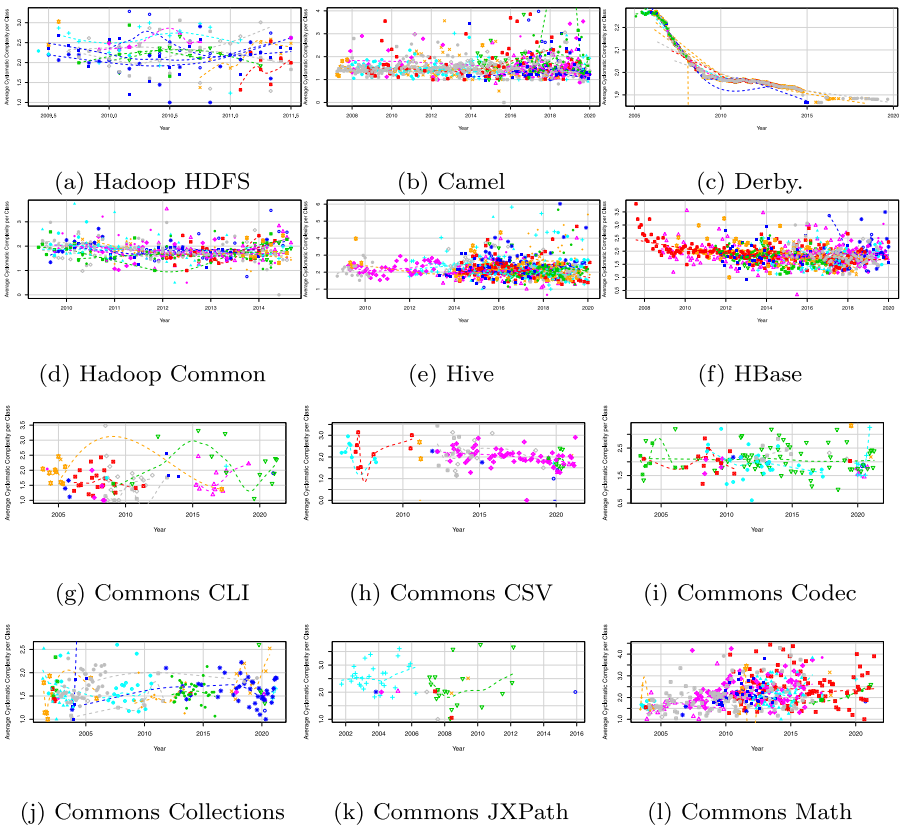
Once the Hawkes model is fitted on data, it will contain some weights (in a matrix fashion) representing the directional strength of any interaction between processes interpreted as the expected number of events on a specific process resulting from an event on another process.

Fig. 16 represents the Hawkes coefficient matrix, fitted with the hypothetical data from the our example. This matrix represents the coefficients of the fitted model: along the diagonal we have the coefficients for the *self-excitation* and the coefficients outside the diagonal represent the *mutual-excitation*. We used the right arrow symbol "↪" to highlight the *direction* of the relationship, i.e. $\lambda(t)_1 \rightarrow \lambda(t)_2$ coefficient represents the strength of the relationship of $\lambda(t)_1$ events on $\lambda(t)_2$ and $\lambda(t)_2 \rightarrow \lambda(t)_1$ viceversa.

Figs. 17 and 18 represents the fitted model for all projects considering the fault insertion in Fig. 17 and fault fixes in Fig. 18. We can see a variety of different behaviours, let us consider for example Fig. 18(a). This Hawkes coefficient matrix highlights a general *self-excitement* relationship of fault insertion considering the developers (non top-5 fault inserters, right bottom corner) meaning that, considering the whole time of analysis, *super fault-inserters* do not influence the fault insertion behaviour of other developers. On the other hand for the Derby project, left top corner of Fig. 17(e) shows, on the contrary, that there is a *self-excitement* of fault insertion among *super fault-inserters*.

4.4. Validation

We validate the three models proposed with a real case based on the Camel project. We selected this project as it is a long running project providing enough data to train our models. We considered a training period of two months to build the three stochastic models: (i) Petri net (ii) Markov Chain and (iii) Hawkes Model, then we used the trained model to predict the next month of activities to evaluate the accuracy of



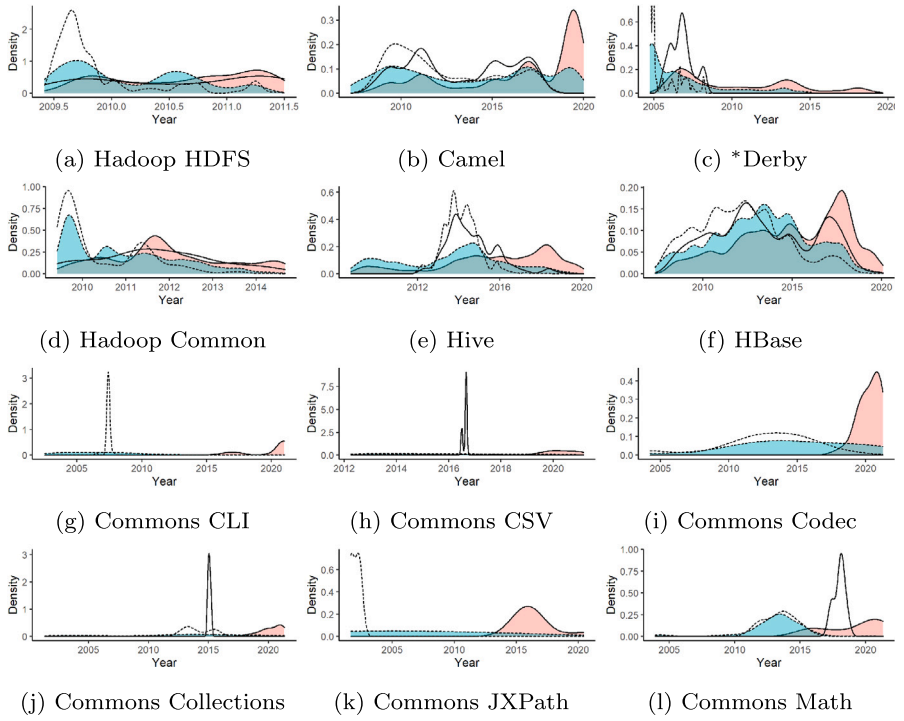
(a) Hadoop HDFS (b) Camel (c) Derby.

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

(g) Commons CLI (h) Commons CSV (i) Commons Codec

(j) Commons Collections (k) Commons JXPath (l) Commons Math

Fig. 12. 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

(g) Commons CLI (h) Commons CSV (i) Commons Codec

(j) Commons Collections (k) Commons JXPath (l) Commons Math

Fig. 13. 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 colour 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.

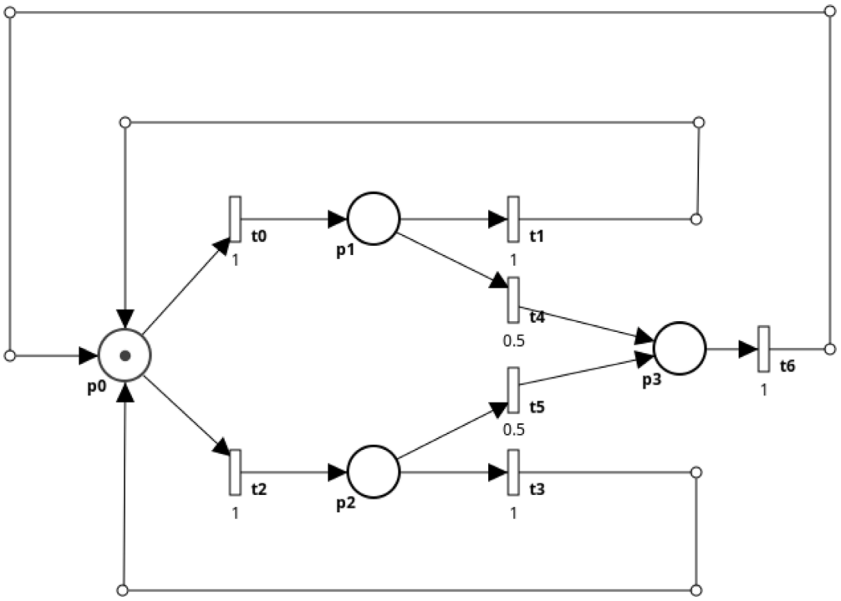


Fig. 14. Development commits activity model using stochastic Petri Nets.

Table 6
Lambda parameters for Camel.

Training period	λ_0	$\lambda_4 - \lambda_5$	λ_2
2016-03-01–2016-04-30	0.213	0.59	0.197
2016-06-01–2016-07-31	0.109	0.656	0.235
2016-09-01–2016-10-31	0.1	0.59	0.31

the predictions. All the three models proposed provide different insights and different points of view regarding fault insertion/fix behaviour.

4.4.1. Petri Nets

Table 6 shows the lambda parameters for the Petri Net model estimated for the first two months of each validation period. These lambda parameters are then used for the steady state analysis of the last month.

Table 7 shows the transient analysis for the three periods considered which gives the corresponding probabilities of being in a particular state for the last month of each period, these probabilities are then compared with the actual percentages, columns *Predicted* and *Actual* respectively in Table 7.

We can see that the predictions are close to the actual values, suggesting that in the short term these predictions can be used by managers for decision support.

4.4.2. Markov Chains

We again considered the same three periods of validation in building Markov Chains. In this case we considered all commits in the first two months of each period to build the MC and then we simulated the probabilities of being in each state after one month. Fig. 19 shows the results of the simulation for each period.

We used the two months trained model (first row of Fig. 19) to simulate the last month of each period (second row of Fig. 19) and compare the probabilities of being in a particular state with the actual percentages (last row of Fig. 19) of development, fault insertion and fix commits.

In this case we can see that the predictions are less accurate (looking at the difference between actual and predicted) compared to the Petri Net model. This is probably due to the fact that MC are a simpler model compared to Petri Net, nevertheless they provide an estimation of the actual percentages that can help decision support and the monitoring of the development process.

4.4.3. Hawkes Model

We evaluated the Hawkes model considering the whole simulation period. In this case we fitted the model using all the data to highlight relationships between top-5 fault inserter/fixer developers and other developers. Fig. 20 shows the coefficients provided by the fitted model. We can see that the insertion/fix of faults by top-5 inserters/fixers has a negative effect on insertion/introduction of bugs due to other developers (top right on coefficient in Fig. 20).

The fault insertion by other developers has a negative self-excitement (bottom left coefficient Fig. 20(a)) meaning that an insertion of fault from the other developers is likely to decrease the number of future fault insertion by other developers. 20(b) shows the four Hawkes coefficients for the fault fix, along the diagonal we can see that there is self-excitements effect on both the fix of fault by top developers and other developers. While there is no mutual-excitement between top 5 developers and the other developers (left bottom corner of 20(b)), we can clearly see that there is a significant negative effect on fault fix caused by top 5 developers on the other developers (top right corner of 20(b)), namely the fault fix of top 5 developers is likely to decrease the fault fix of other developers (vice versa is not true).

5. Discussion

Software development is usually done in complex socio-technical teams [47]. Several types of software development teams exist ranging from the historical Chief Programmer teams [48] to the agile teams [49] that are now very common. Costa et al.'s [50] recent systematic literature review suggests that, regardless of team type, the technical attributes of an individual developer are most commonly used to select team members. Costa et al.'s [50] results confirm the importance of individual expertise in software development teams. In addition, Faraj and Sproull's [51] analysis of 69 software development teams suggests that: (1) it is important to have a range of expertise on teams (2) there is knowledge within the team of the available expertise and (3) expertise in the team should to be effectively coordinated. It is clear that expertise in software development teams is an important factor.

The aim of our study was to understand better the role of expertise in the fault insertion and fault fixing behaviour of developers. Such improved understanding could enable development activities to be planned and managed more effectively both in terms of resource

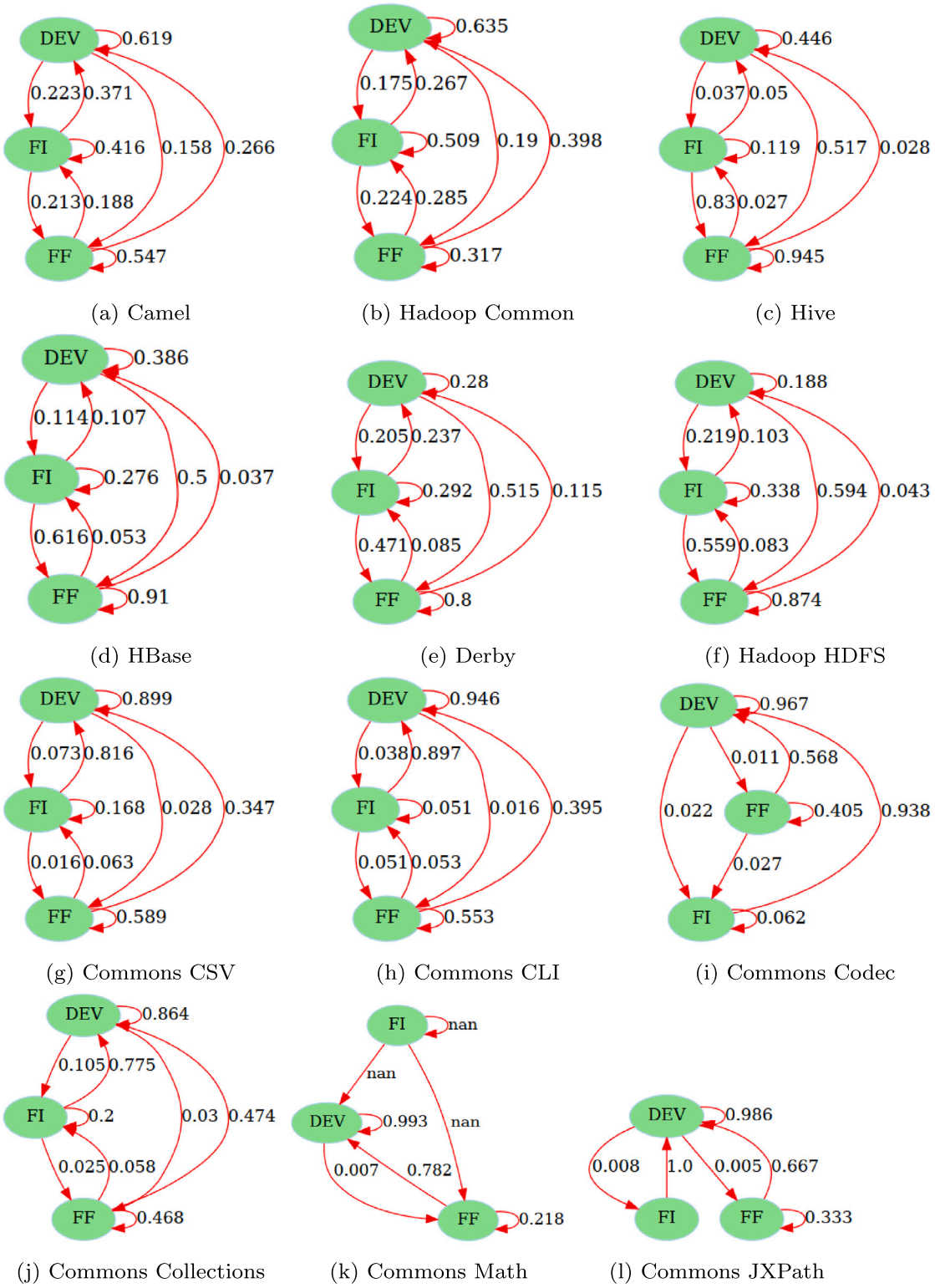


Fig. 15. Markov chains.

Table 7
Probabilities for the three periods considered.

Testing period	P_0		P_1		P_2		P_3	
	Predicted	Actual	Predicted	Actual	Predicted	Actual	Predicted	Actual
2016-05-01–2016-05-31	0.635	0.608	0.074	0.132	0.147	0.114	0.144	0.146
2016-08-01–2016-08-31	0.579	0.603	0.131	0.115	0.145	0.135	0.144	0.146
2016-11-01–2016-11-30	0.611	0.615	0.102	0.056	0.141	0.183	0.146	0.146

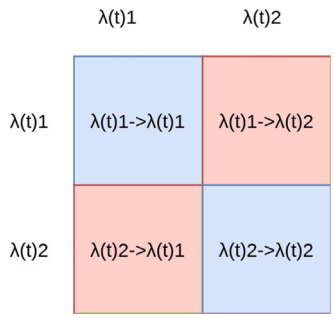


Fig. 16. Example of Hawkes coefficient matrix.

planning and reduced faults in delivered software. Poor planning and latent faults in delivered software continue to be at the root of many software development failures.[6] All 70 failed software projects analysed by Verna et al. [52] suffered from poor project management. Given the scale of the problem, any small improvements that our approach makes towards enabling better project management, could make a significant overall difference to the success of delivered software systems.

Understanding fault fixing and fault insertion behaviour could enable teams to be composed more effectively (with the right developers at the right time in the development process). In addition, the overall expertise relevant to the needs of the project could also be optimised. Sablis et al.'s [53] analysis of developer opinions in nine teams suggests that coordination is needed between teams to ensure the right mix of expertise is available at any point in time for a specific project. However, there are many variations of what constitutes an effective team. Sawyer [47] suggests a variety of hybrid variations on team types. Project managers need to understand the nature of their team so that developer expertise information can be used effectively to enable improved software development outcomes. Expertise is not the only factor that should be taken into account when forming a team. For example, the diversity of the team is important [25,54] as is the effectiveness of team communication [55].

We tried to understand whether it is important that developers have expertise in the code that they touch. We measure expertise using topic analysis, which helps to summarise the description of the issues written by developers. Clustering issues into topics allowed us to assign a dominant topic to each developer. We have been able to identify the most frequent topic in the issues that each developer has worked on. Our findings suggest 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. Our results provide quantitative support for Sablis et al.'s [53] findings on the importance of coordinating expertise within project teams.

We also considered that experience changes over time: both the experience of a single developer and the general experience of a team (e.g., new developers could join, and even with a high level of expertise, they will still have to understand what the team is doing and what the problems are). Since developers' experience changes over time, it is likely that developers' fault insertion and fixing also changes as time goes on. We think that discovering how experience changes over time and the impact on fault insertion and fixing patterns could help organise the team and the tasks in relation to developers on the project.

We built models of developers' activities using three different Stochastic Models: (i) Petri Net, (ii) Markov Chains and (iii) Hawkes Models. Each of these three modelling approaches offers different insights into the behaviour of developers. Petri Nets enabled us to simulate a team of developers (and their fault fixing/insertion patterns). These simulations could allow project managers to predict the outcome of changes introduced in the team configuration (e.g., new members joining, different task allocations, etc.). Using Petri Nets in this way could offer project managers additional insight to complement the defect prediction models previously presented (e.g. by [12]). Using Markov Chains it was possible to simulate and predict the percentages of fault insertion and fixing commits, from which we were able to estimate short term fault fixing behaviour. Finally using the Hawkes Model we were able to highlight hidden relationships in fault insertion/fixing behaviour. For example, between the *top-n* fault inserters and fixers and the other developers to identify the role of the super-introducers and super-fixers during the development process. In the future we plan to evaluate the insight offered by our suite of models with project managers.

To answer our initial Research Questions:

RQ1: Can we identify those developers most likely to insert and fix faults in code? Yes, using network analysis techniques it is possible to characterise developers' behaviour and identify those most likely to insert and fix faults. We hope our results can help managers to better distribute the workload in a team and potentially identify weaknesses in the development process.

RQ2: Does expertise impact developers' fault insertion and fixing? Yes, the expertise of fault fixers is better aligned to the topic of the fault, and in most cases the developer who inserts the fault is not an expert in the fault topic (27% to 93% of faults are introduced by developers whose dominant expertise is not in the topic).

RQ3: Does experience over time on projects impact developers' fault insertion and fixing? Our results show that the most active committers 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. Specific patterns of activity vary from one project to another.

RQ4: Can we model developers' activities with Stochastic models? Yes, we modelled developers activities in terms of the fault insertion, fault fixing and normal development activities using three stochastic models: (i) Petri Net (ii) Markov Chains and (iii) Hawkes models. Using these models, we were able to estimate development effort.

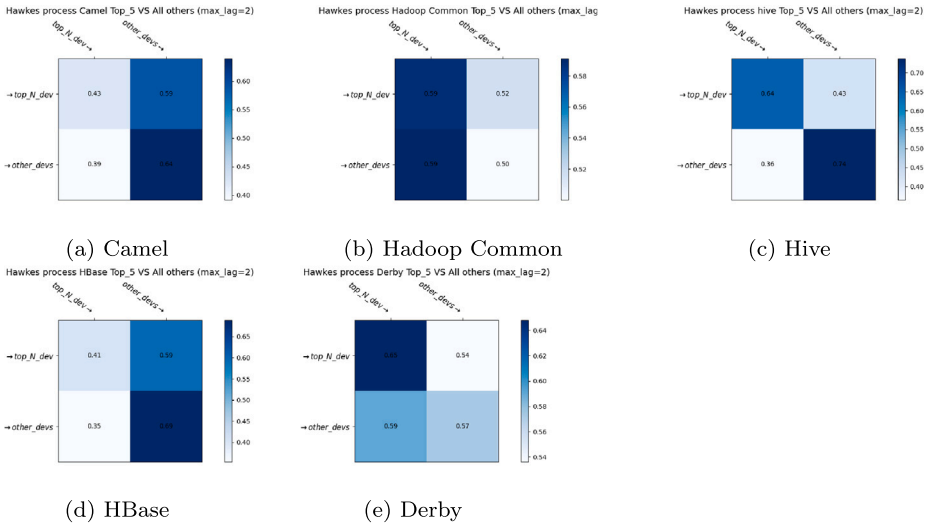
Our analysis is based on the historical analysis of developer behaviour in code bases. As with all such historical analysis it is increasingly obvious that care must be taken in the interpretation of the results to prevent counterproductive actions, this is especially important given our work is focused on individual developers and their expertise. Tozun et al. [56] present a compelling analysis of the dangers that blindly relying only on historical data can bring. For example, in our data, project managers must be aware that an analysis of historical behaviour by individuals may mask recent skills development that would result in different future behaviours. Similarly, project managers must also be mindful of measuring the performance of individual developers via their expertise to avoid the age-old dangers previously associated with measuring developer productivity without the full context in which the developer is working [57].

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 twelve Apache projects from GitHub repositories. As the results of the *Camel*

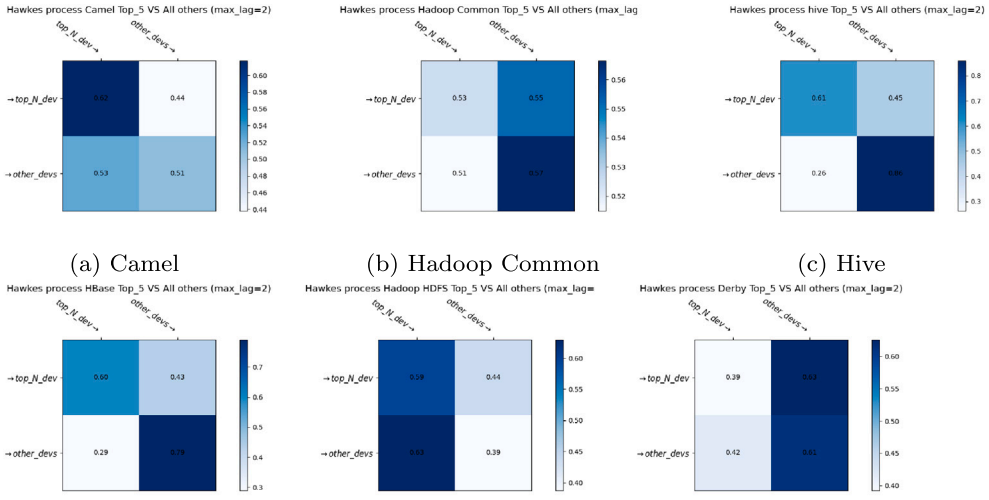
[6] https://www.computerworld.com/article/3412197/top-software-failures-in-recent-history.html.



(a) Camel (b) Hadoop Common (c) Hive

(d) HBase (e) Derby

Fig. 17. Hawkes process for fault inserters.



(a) Camel (b) Hadoop Common (c) Hive

(d) HBase (e) Hadoop HDFS (f) Derby

Fig. 18. Hawkes process for fault fix.

and Commons JXPath projects 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

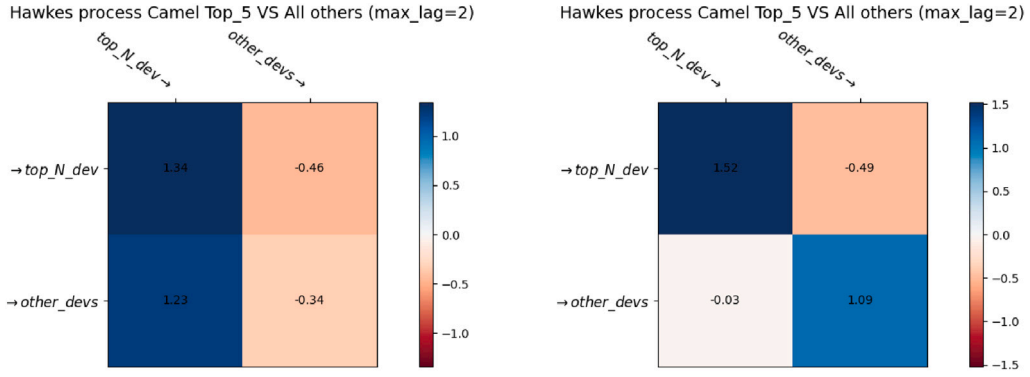
We performed a multi-dimensional study to identify patterns of developer activity over time. We considered the fault insertion and fixing activity of developers, the familiarity of developers with code topics, alongside code complexity. Our time-based analysis was performed throughout the history of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

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 as 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 plan to extend our analysis to more Open Source projects as well as to validate our findings on closed source systems. We intend to implement a dashboard to provide an accessible tool to aid the management of development activities for the reduction of faults.

(a) Camel Validation From 2016-03-01 To 2016-05-31 (b) Camel Validation From 2016-06-01 To 2016-08-31 (c) Camel Validation From 2016-09-01 To 2016-11-30

Fig. 19. Validation simulation.



(a) Fault Insertions. (b) Fault Fix.

Fig. 20. Hawkes model coefficients from 2016-04-01 To 2016-10-31.

Future analysis will be focused also in better understanding why certain developers contribute to fault fixing (and fault insertion) more in certain project than others, and studying the change in requirements to analyse if there are relationship with faults introduction.

CRediT authorship contribution statement

Marco Ortu: Conceptualization, Methodology, Software, Data curation, Writing – original draft. **Giuseppe Destefanis:** Conceptualization, Methodology, Software, Data curation, Writing – original draft. **Tracy Hall:** Conceptualization, Methodology, Software, Data curation, Writing – original draft. **David Bowes:** Conceptualization, Methodology, Software, Data curation, Writing – original draft.

Declaration of competing interest

One or more of the authors of this paper have disclosed potential or pertinent conflicts of interest, which may include receipt of payment, either direct or indirect, institutional support, or association with an entity in the biomedical field which may be perceived to have potential

conflict of interest with this work. For full disclosure statements refer to https://doi.org/10.1016/j.infsof.2023.107187. Tracy Hall, David Bowes reports that this work is partly funded by grants from the UK's Engineering and Physical Sciences Research Council (EP/S005730/1 and EP/S005749/2).

Data availability

Data are available.

Acknowledgements

This work is partly funded by grants from the UK's Engineering and Physical Sciences Research Council (EP/S005730/1 and EP/S00 5749/2).

References

[1] C. Hauff, G. Gousios, Matching GitHub developer profiles to job advertisements, in: 2015 IEEE/ACM 12th Working Conference on Mining Software Repositories, IEEE, Florence, 2015, pp. 362–366.

[2] P. Thongtanunam, C. Tantithamthavorn, R.G. Kula, N. Yoshida, H. Iida, K. Matsumoto, 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, 2015, pp. 141–150.

[3] A. Yadav, S.K. Singh, J.S. Suri, Ranking of software developers based on expertise score for bug triaging, Inf. Softw. Technol. 112 (2019) 1–17.

[4] G. Catolino, F. Palomba, A. Zaidman, F. Ferrucci, Not all bugs are the same: Understanding, characterizing, and classifying bug types, J. Syst. Softw. 152 (2019) 165–181.

[5] J. Eyolfson, L. Tan, P. Lam, 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, 2011, pp. 153–162, http://dx.doi.org/10.1145/1985441.1985464.

[6] Y. Qiu, W. Zhang, W. Zou, J. Liu, Q. Liu, An empirical study of developer quality, in: 2015 IEEE International Conference on Software Quality, Reliability and Security-Companion, IEEE, 2015, pp. 202–209.

[7] M. Zhou, A. Mockus, Developer fluency: Achieving true mastery in software projects, in: Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2010, pp. 137–146.

[8] O. Dieste, A.M. Aranda, F. Uyaguari, B. Turhan, A. Tosun, D. Fucci, M. Oivo, N. Juristo, Empirical evaluation of the effects of experience on code quality and programmer productivity: an exploratory study, Empir. Softw. Eng. 22 (5) (2017) 2457–2542.

[9] S. Baltes, S. Diehl, 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, 2018, pp. 187–200.

[10] C. Bird, N. Nagappan, B. Murphy, H. Gall, P. Devanbu, 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, 2011, pp. 4–14.

[11] M. Greiler, K. Herzig, J. Czerwonka, Code ownership and software quality: A replication study, in: Proceedings of the 12th Working Conference on Mining Software Repositories, IEEE, Florence, 2015, pp. 2–12.

[12] S. Ozcan Kini, A. Tosun, Periodic developer metrics in software defect prediction, in: 2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation, IEEE, Madrid, 2018, pp. 72–81.

[13] D. Bowes, G. Destefanis, T. Hall, J. Petric, M. Ortu, 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, 2020, pp. 41–50.

[14] A. Mockus, J.D. Herbsleb, Expertise browser: a quantitative approach to identifying expertise, in: Proceedings of the 24th International Conference on Software Engineering., IEEE, Orlando, 2002, pp. 503–512.

[15] S. Matsumoto, Y. Kamei, A. Monden, K.-i. Matsumoto, M. Nakamura, An analysis of developer metrics for fault prediction, in: Proceedings of the 6th International Conference on Predictive Models in Software Engineering, ACM, NY, USA, 2010.

[16] M. Foucault, M. Palyart, X. Blanc, G.C. Murphy, J.-R. Falleri, 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, 2015, pp. 829–841.

[17] J. Businge, S. Kawuma, E. Bainomugisha, F. Khomh, E. Nabaasa, 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, 2017, pp. 33–42.

[18] T. Fritz, G.C. Murphy, E. Hill, 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, 2007, pp. 341–350.

[19] F. Rahman, P. Devanbu, Ownership, experience and defects: a fine-grained study of authorship, in: Proceedings of the 33rd International Conference on Software Engineering, 2011, pp. 491–500.

[20] H. Hokka, F. Dobslaw, J. Bengtsson, Linking developer experience to coding style in open-source repositories, in: 2021 IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER, IEEE, 2021, pp. 516–520.

[21] W. Zhu, M.W. Godfrey, Mea culpa: How developers fix their own simple bugs differently from other developers, in: 2021 2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR), MSR, IEEE Computer Society, Los Alamitos, CA, USA, 2021, pp. 515–519, http://dx.doi.org/10.1109/MSR52588.2021.00065, https://doi.ieeecomputersociety.org/10.1109/MSR52588.2021.00065.

[22] J. Wang, X. Meng, H. Wang, H. Sun, An online developer profiling tool based on analysis of GitLab repositories, in: Y. Sun, T. Lu, Z. Yu, H. Fan, L. Gao (Eds.), Computer Supported Cooperative Work and Social Computing, Springer Singapore, Singapore, 2019, pp. 408–417.

[23] D.M. Blei, A.Y. Ng, M.I. Jordan, Latent dirichlet allocation, J. Mach. Learn. Res. 3 (Jan) (2003) 993–1022.

[24] E. Constantinou, G.M. Kapitsaki, Identifying developers' expertise in social coding platforms, in: 2016 42th Euromicro Conference on Software Engineering and Advanced Applications, IEEE, Limassol, 2016, pp. 63–67.

[25] B. Vasilescu, D. Posnett, B. Ray, M.G. van den Brand, A. Serebrenik, P. Devanbu, V. Filkov, Gender and tenure diversity in GitHub teams, in: Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems, ACM, NY, USA, 2015, pp. 3789–3798.

[26] J.E. Montandon, L.L. Silva, M.T. Valente, Identifying experts in software libraries and frameworks among GitHub users, in: 2019 IEEE/ACM 16th International Conference on Mining Software Repositories, IEEE, Montreal, 2019, pp. 276–287.

[27] P.C. Rigby, Y.C. Zhu, S.M. Donadelli, A. Mockus, 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, 2016, pp. 1006–1016.

[28] T.F. Bissyandé, F. Thung, S. Wang, D. Lo, L. Jiang, L. Reveillere, Empirical evaluation of bug linking, in: 2013 17th European Conference on Software Maintenance and Reengineering, IEEE, 2013, pp. 89–98.

[29] D. Bowes, J. Petrić, T. Hall, BugVis: Commit slicing for fault visualisation, in: Proceedings of the 28th International Conference on Program Comprehension, 2020, pp. 436–440.

[30] J. Sliwerski, T. Zimmermann, A. Zeller, When do changes induce fixes? in: Proceedings of the 2005 International Workshop on Mining Software Repositories, ACM, NY, USA, 2005, pp. 1–5.

[31] G. Rodríguez-Pérez, G. Robles, J.M. González-Barahona, Reproducibility and credibility in empirical software engineering: A case study based on a systematic literature review of the use of the SZZ algorithm, Inf. Softw. Technol. 99 (2018) 164–176.

[32] D. Bowes, S. Counsell, T. Hall, J. Petric, T. Shippey, Getting defect prediction into industrial practice: the elff tool, in: 2017 IEEE International Symposium on Software Reliability Engineering Workshops, ISSREW, IEEE, 2017, pp. 44–47.

[33] E. Kouters, B. Vasilescu, A. Serebrenik, M.G. Van Den Brand, Who's who in Gnome: Using LSA to merge software repository identities, in: 2012 28th IEEE International Conference on Software Maintenance, ICSM, IEEE, 2012, pp. 592–595.

[34] T. Fry, T. Dey, A. Karnauch, A. Mockus, A dataset and an approach for identity resolution of 38 million author ids extracted from 2b git commits, in: Proceedings of the 17th International Conference on Mining Software Repositories, 2020, pp. 518–522.

[35] I.S. Wiese, J.T. Da Silva, I. Steinmacher, C. Treude, M.A. Gerosa, Who is who in the mailing list? Comparing six disambiguation heuristics to identify multiple addresses of a participant, in: 2016 IEEE International Conference on Software Maintenance and Evolution, ICSME, IEEE, 2016, pp. 345–355.

[36] A. Meneely, L. Williams, W. Snipes, J. Osborne, 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, 2008, pp. 13–23.

[37] F. Thung, T.F. Bissyande, D. Lo, L. Jiang, Network structure of social coding in github, in: 2013 17th European Conference on Software Maintenance and Reengineering, IEEE, Genova, 2013, pp. 323–326.

[38] U. Brandes, A faster algorithm for betweenness centrality, J. Math. Sociol. 25 (2) (2001) 163–177.

[39] D.M. Blei, J.D. Lafferty, Dynamic topic models, in: Proceedings of the 23rd International Conference on Machine Learning, ACM, NY, USA, 2006, pp. 113–120.

[40] Y.B. Kim, J. Lee, N. Park, J. Choo, J.-H. Kim, C.H. Kim, 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.
[41] M. Röder, A. Both, A. Hinneburg, Exploring the space of topic coherence measures, in: Proceedings of the Eighth ACM International Conference on Web Search and Data Mining, 2015, pp. 399–408.
[42] M.F. Neuts, A versatile Markovian point process, J. Appl. Probab. (1979) 764–779.
[43] A. Hawkes, Spectra of some self-exciting and mutually exciting point processes, Biometrika (1971).
[44] T. Shippey, D. Bowes, T. Hall, Automatically identifying code features for software defect prediction: Using AST N-grams, Inf. Softw. Technol. 106 (2019) 142–160.
[45] M.I. Jordan, Learning in Graphical Models:[Proceedings of the NATO Advanced Study Institute: Ettore Mairona Center, Erice, Italy, September 27-October 7, 1996], Vol. 89, Springer Science & Business Media, 1998.
[46] T.A. Snijders, The statistical evaluation of social network dynamics, Sociol. Methodol. 31 (1) (2001) 361–395.
[47] S. Sawyer, Software development teams, Commun. ACM 47 (12) (2004) 95–99.
[48] F.T. Baker, Chief programmer team management of production programming, in: Classics in Software Engineering, Yourdon Press, USA, 1979, pp. 63–82.
[49] A. Cockburn, J. Highsmith, Agile software development, the people factor, Computer 34 (11) (2001) 131–133.
[50] A. Costa, F. Ramos, M. Perkusich, E. Dantas, E. Dilorenzo, F. Chagas, A. Meireles, D. Albuquerque, L. Silva, H. Almeida, A. Perkusich, Team formation in software engineering: A systematic mapping study, IEEE Access 8 (2020) 145687–145712, http://dx.doi.org/10.1109/ACCESS.2020.3015017.
[51] S. Faraj, L. Sproull, Coordinating expertise in software development teams, Manage. Sci. 46 (12) (2000) 1554–1568.
[52] J. Verner, J. Sampson, N. Cerpa, What factors lead to software project failure? in: 2008 Second International Conference on Research Challenges in Information Science, 2008, pp. 71–80, http://dx.doi.org/10.1109/RCIS.2008.4632095.
[53] A. Sablis, D. Smite, N. Moe, Team-external coordination in large-scale software development projects, J. Softw.: Evol. Process 33 (3) (2021) e2297, http://dx.doi.org/10.1002/smr.2297, https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.2297, e2297 smr.2297.
[54] M. Ortu, G. Destefanis, S. Counsell, S. Swift, R. Tonelli, M. Marchesi, How diverse is your team? Investigating gender and nationality diversity in GitHub teams, J. Softw. Eng. Res. Dev. 5 (1) (2017) 1–18.
[55] E. Bjarnason, B. Gislason Bern, L. Svedberg, Inter-team communication in large-scale co-located software engineering: a case study, Empir. Softw. Eng. 27 (2) (2022) 1–43.
[56] E. Tüzün, H. Erdogmus, M.T. Baldassarre, M. Felderer, R. Feldt, B. Turhan, Ground truth deficiencies in software engineering: when codifying the past can be counterproductive, IEEE Softw. (2021).
[57] C. Jaspan, C. Sadowski, No single metric captures productivity, in: Rethinking Productivity in Software Engineering, Springer, 2019, pp. 13–20.