Are Fix-Inducing Changes a Moving Target?

A Longitudinal Case Study of Just-In-Time Defect Prediction

Shane McIntosh, *Member, IEEE* and Yasutaka Kamei, *Member, IEEE*

Abstract—Just-In-Time (JIT) models identify fix-inducing code changes. JIT models are trained using techniques that assume that past fix-inducing changes are similar to future ones. However, this assumption may not hold, e.g., as system complexity tends to accrue, expertise may become more important as systems age.

In this paper, we study JIT models as systems evolve. Through a longitudinal case study of 37,524 changes from the rapidly evolving QT and OPENSTACK systems, we find that fluctuations in the properties of fix-inducing changes can impact the performance and interpretation of JIT models. More specifically: (a) the discriminatory power (AUC) and calibration (Brier) scores of JIT models drop considerably one year after being trained; (b) the role that code change properties (e.g., Size, Experience) play within JIT models fluctuates over time; and (c) those fluctuations yield over- and underestimates of the future impact of code change properties on the likelihood of inducing fixes. To avoid erroneous or misleading predictions, JIT models should be retrained using recently recorded data (within three months). Moreover, quality improvement plans should be informed by JIT models that are trained using six months (or more) of historical data, since they are more resilient to period-specific fluctuations in the importance of code change properties.

Index Terms—Just-In-Time prediction, defect prediction, mining software repositories

1 Introduction

THE limited Software Quality Assurance (SQA) resources of software organizations must focus on software modules that are likely to be defective in the future. To that end, *defect prediction models* are trained using historical data to identify defect-prone software modules (e.g., methods [9, 15], files [48], or subsystems [30]). After being trained using data from historical releases, defect prediction models can be used to prioritize SQA effort according the predicted defect proneness of the modules of a future release.

Change-level defect prediction [28], a.k.a., Just-In-Time (JIT) defect prediction [18], is an alternative to module-level defect prediction, which has several advantages [38]. First, since changes are often smaller than modules, JIT predictions are made at a finer granularity, which localizes the inspection process. Second, while modules have a group of authors, changes have only one, which makes triaging JIT predictions easier (i.e., predictions can be assigned to the author of the change). Finally, unlike module-level prediction, JIT models can scan changes as they are being produced, which means that problems can be inspected while design decisions are still fresh in the developers' minds.

Despite the advantages of JIT defect prediction, like all prediction models, they assume that the properties of past events (i.e., fix-inducing changes) are similar to the properties of future ones. Yet, the consistency of properties of fix-inducing changes remains largely unexplored. It may indeed be the case that the properties of fix-inducing changes in one development period are entirely different from the fix-inducing changes of another. For example, since complexity tends to accrue as systems age [3], expertise may grow more important as systems age. To that end, in this paper, we set out to address the following central question:

Do the important properties of fix-inducing changes remain consistent as systems evolve?

To address our central question, we train JIT models to identify fix-inducing changes using six families of code change properties, which are primarily derived from prior studies [18, 19, 23, 28]). These properties measure: (a) the magnitude of the change (*Size*); (b) the dispersion of the changes across modules (*Diffusion*); (c) the defect proneness of prior changes to the modified modules (*History*); (d) the experience of the author (*Author Experience*) and (e) code reviewer(s) (*Reviewer Experience*); and (f) the degree of participation in the code review of the change (*Review*). Through a longitudinal case study of the QT and OPENSTACK projects, we address the following research questions:

(RQ1) Do JIT models lose predictive power over time?
Motivation: If properties of fix-inducing changes do change, JIT models that were trained to identify fix-inducing changes of the past would quickly lose their predictive power. Hence, we are first interested in (a) whether JIT models lose predictive power over time, and (b) how quickly such losses would occur.
Results: After one year, our JIT models typically lose 11–22 and 14–34 percentage points of their discriminatory power (AUC) in the QT and OPENSTACK systems, respectively. Similarly, after one year, our JIT models lose up to 11 and 19 percentage points from their calibration (Brier) scores in the QT and OPENSTACK systems, respectively.

- *S. McIntosh is with the Department of Electrical and Computer Engineering, McGill University, Canada.*
Email: shane.mcintosh@mcgill.ca
- *Y. Kamei is with the Principles of Software Languages Group (POSL), Kyushu University, Japan.*
Email: kamei@ait.kyushu-u.ac.jp

(RQ2) Does the relationship between code change properties and the likelihood of inducing a fix evolve?
Motivation: Another symptom of changing properties of fix-inducing changes would be fluctuations in the impact that code change properties have on the explanatory power of JIT models. If properties of fix-inducing changes do indeed change, the prediction modelling assumption that properties of prior and future events are similar may not be satisfied. To cope with this, teams may need to refit their models. To better understand how volatile the properties of fix-inducing changes are, we set out to study trends in the importance scores of code change properties in our JIT models over time.
Results: The *Size* family of code change properties is a consistent contributor of large importance scores. However, the magnitude of these importance scores fluctuates considerably, ranging between 10%–43% and 3%–37% of the period-specific explanatory power of our QT and OPENSTACK JIT models, respectively.

(RQ3) How accurately do current importance scores of code change properties represent future ones?
Motivation: Fluctuations in the importance scores of code change properties may impact *quality improvement plans*, i.e., team initiatives to reduce the rate of defects, which are based on interpretation of JIT models. For example, if a quality improvement plan is formulated based on a large importance score of a property p, but p drops in importance in the following period, even a perfectly executed quality improvement plan will have a smaller impact than anticipated. On the other hand, if property p has a small importance score in the past, but grows more important in the following period, the quality improvement plan may omit important focal points. To better understand the impact that fluctuations have on quality improvement plans, we study the magnitude of these fluctuations in our JIT models.
Results: The stability of property importance scores in our JIT models is project-sensitive. The contributions of impactful property families like *Size* are both consistently: (a) overestimated in our QT JIT models by a median of 5% of the total explanatory power and (b) underestimated in our OPENSTACK JIT models by a median of 2% of the total explanatory power.

Our results lead us to conclude that properties of fix-inducing changes can fluctuate, which may impact both the performance and interpretation of JIT defect models. To mitigate the impact on model performance, researchers and practitioners should add recently accumulated data to the training set and retrain JIT models to contain fresh data from within the last six months. To better calibrate quality improvement plans (which are based on interpretation of the importance scores of code change properties), researchers and practitioners should put a greater emphasis on larger caches of data, which contain at least six months worth of data, to smooth the effect of spikes and troughs in the importance of properties of fix-inducing changes.

Paper Organization

The remainder of the paper is organized as follows. Section 2 discusses the related work. Section 3 describes the design of our case study, while Section 4 presents the results. Section 5 presents practical suggestions that are informed by our study results. Section 6 discloses the threats to the validity of our study. Finally, Section 7 draws conclusions.

2 RELATED WORK

In this section, we situate our work within the context of past studies on JIT modelling and its assumptions.

2.1 JIT Defect Modelling

Recent work has shown that JIT models have become sufficiently robust to be applied in practice. Indeed, JIT models are included in the development processes of large software systems at Avaya [28], Blackberry [38], and Cisco [40].

To achieve such robustness, a variety of code change properties need to be used to predict fix-inducing changes. For example, Mockus and Weiss [28] predicted fix-inducing changes using a set of code change properties that are primarily derived from the changes themselves. Kim *et al.* [19] and Kamei *et al.* [18] built upon the set of code change properties, reporting that the addition of a variety of properties that were extracted from the Version Control System (VCS) and the Issue Tracking System (ITS) helped to improve the prediction accuracy. Kononenko *et al.* [23] also found that the addition of code change properties that were extracted from code review databases contributed a significant amount of explanatory power to JIT models.

We derive the majority of our set of studied code change properties from those described in these prior studies. However, while these previous studies empirically evaluate the prediction performance of JIT models, they do not focus on the consistency of the properties of fix-inducing changes, which is the central thrust of this paper.

2.2 Assumptions of Defect Modelling

Past work has shown that if care is not taken when collecting data from software repositories, noise may impact defect models. Aranda and Venolia [2] found that VCSs and ITSs are noisy sources of data. For example, Antoniol *et al.* [1] found that issue reports are often mislabelled, i.e., reports that describe defects are labelled as enhancements, and vice versa. Herzig *et al.* [16] found that this issue report mislabelling can impact the ranking of modules that is produced by defect models. Furthermore, Bird *et al.* [4] found that characteristics of issue reports (e.g., severity) and issue reporters (e.g., experience) can influence the likelihood of linking related ITS and VCS records when it is necessary. Without these links, datasets that are constructed for defect prediction purposes may erroneously label defective modules as clean, or vice versa. To study the impact that missing links may have on defect prediction, Kim *et al.* [20] sever existing links to simulate a variety of rates of missing links.

On the other hand, recent work also shows that noise and bias may not be a severe impediment to defect modelling. For example, our prior work [41] showed that issue report mislabelling rarely impacts the precision of defect

TABLE 1
An overview of the studied systems. Those above the double line satisfy our criteria for analysis.

System	Timespan		Changes		Reviews		
	Start	**End**	**Total**	**Defective**	**Reviewed**	**Self reviewed**	**# Reviewers**
Qt	06/2011	03/2014	25,150	2,002 (8%)	23,821 (95%)	1,217 (5%)	$\bar{x} = 2.3$, $M = 1.0$, $sd = 2.8$
OpenStack	11/2011	02/2014	12,374	1,616 (13%)	12,041 (97%)	117 (<1%)	$\bar{x} = 4.3$, $M = 3.0$, $sd = 5.3$
ITK	08/2010	08/2014	3,347	-	-	-	-
VTK	08/2010	08/2014	7,723	-	4,237 (55%)	-	-
Android	08/2010	01/2013	61,789	-	2,771 (4%)	-	-
LibreOffice	03/2012	11/2013	11,988	-	1,679 (14%)	-	-

models or the interpretation of the top-ranked variables. Nguyen *et al.* [33] found that biases exist even in a "near-ideal" development setting, where links are automatically recorded, suggesting that linkage bias is a symptom of any development process. Moreover, Rahman *et al.* [35] found that the total number of modules has a larger impact on defect model performance than noise or biases do.

Even with data that is completely free of noise and bias, there are assumptions that must be satisfied in order to fit defect models. Turhan [44] argued that "dataset shifts" (i.e., dataset characteristics that violate modelling assumptions [27]) may influence predictive models in software engineering. In a similar vein, we study whether properties of fix-inducing changes are consistent enough to satisfy the assumption that past events are similar to future events.

Past studies also use historical data to improve the performance of defect prediction models [22, 32, 47]. For example, Nam *et al.* [32] mitigated dataset shifts by applying a transfer learning approach, which makes feature distributions in training and testing datasets more similar. Zimmermann *et al.* [47] showed how complexity, problem domain, and change rate can be used to predict defects in Microsoft and Eclipse systems. While these studies predict which modules are at-risk of containing defects, our study focuses on understanding the impact that various code change properties have on the risk that changes pose for inducing future fixes in a longitudinal setting.

Perhaps the most similar prior work is that of Ekanayake *et al.* [7], who studied the stability of module-level defect prediction models. They used trends in the predictive power of defect prediction models to identify periods of "concept drift," i.e., periods where historical trends do not aid in identifying defect-prone modules. Our work differs from that of Ekanayake *et al.* in two ways. First, we study JIT models, which are concerned with fix-inducing changes rather than defective modules. Second, while past work focuses on model performance, we study the impact that fluctuations in the importance of properties of fix-inducing changes have on both the performance (RQ1) and the interpretation (RQ2, RQ3) of JIT models.

3 Case Study Design

In this section, we describe our: (1) rationale for selecting our studied systems and (2) data extraction process. In addition, we describe our approaches to: (3) model construction and (4) model analysis.

3.1 Studied Systems

To address our research questions, we perform a longitudinal case study on successful and rapidly evolving open source systems. In selecting the subject systems, we identified three important criteria that needed to be satisfied:

- **Criterion 1: Traceability.** We need reliable data in order to produce healthy datasets (see Section 2.2). To that end, it should be reasonably straightforward to extract from the VCS of a studied system, a series of *code changes*, i.e., listings of changed lines to a set of code files that developers have submitted to the VCS together. It should also be straightforward to connect a large proportion of those code changes to issue reports, which are stored in ITSs, and code review data, which is stored in code review databases. Without a traceable process, our code change properties will be unreliable, and our JIT models may be misleading.
- **Criterion 2: Rapidly evolving.** In order for JIT models to yield the most benefit, our studied systems must undergo plenty of change on a continual basis.
- **Criterion 3: Code review policy.** Recent studies report that code review can have an impact on post-release software quality [26, 42] and defect proneness [23]. To control for this, we focus our study on systems where code reviewing is a common practice.

In order to address criterion 1, we study systems that adopt the Gerrit code review tool.[1] Gerrit tightly integrates with VCSs, automatically submitting commits after project-specific quality gates are passed (e.g., two reviewers vote in favour of acceptance of a change). These Gerrit-generated commits are accompanied by a structured message, which includes references to the review record (ChangeID) and any addressed issue reports (IssueIDs).

Table 1 shows that, similar to our prior work [26, 43], we begin with six candidate systems that adopt the Gerrit code review tool. In order to address criterion 2, we remove ITK from our set of candidate systems because only 3,347 changes appear in the ITK Gerrit instance over four years.

In order to address criterion 3, we ensure that the studied systems have high rates of review coverage. Since we find that only 55% of VTK, 4% of Android, and 14% of LibreOffice changes could be linked to reviews, we remove these three systems from our set of candidate systems.

Table 1 provides an overview of the two candidate systems that satisfy our selection criteria (Qt and OpenStack).

[1] https://code.google.com/p/gerrit/

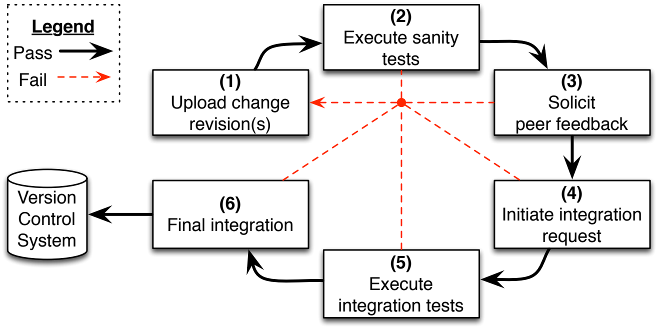


Fig. 1. An overview of the Gerrit-based code review process.

QT is a cross-platform application framework whose development is supported by the Digia corporation, however welcomes contributions from the community-at-large.[2] OPENSTACK is an open-source software platform for cloud computing that is developed by many well-known software organizations (e.g., IBM, VMware, NEC).[3] Although the cleaning process is described below (see Section 3.2), the clean QT dataset contains 25,150 code changes, 23,821 of which (95%) can be linked to code reviews. Similarly, the clean OPENSTACK dataset contains 12,374 code changes, 12,041 of which (97%) can be linked to code reviews.

3.1.1 Gerrit Code Review Process

Gerrit is a code review tool that enables a traceable code review process for git-based software projects. It tightly integrates with test automation and code integration tools, allowing users to codify code review and verification criteria that must be satisfied before a code change can be integrated into upstream git repositories.

Using Gerrit, both the QT and OPENSTACK projects implement similar workflows for managing code contributions. Figure 1 provides an overview of the process, which is made up of the following steps, checks, and quality gates.

(1) Upload change revision(s). An author of a code change uploads a new change or a change revision to a Gerrit instance and invites a set of reviewers to critique it by leaving comments for (a) the author to address; or (b) review participants to discuss.

(2) Execute sanity tests. Before reviewers examine the submitted changes, sanity tests verify that the changes are compliant with the coding style conventions, and does not introduce obvious regression in system behaviour (e.g., code does not compile). If the sanity tests report issues, the change is blocked from integration until the author uploads a revision of the change that addresses the issues. This step provides quick feedback and avoids wasting reviewer effort on finding style or low-level coding issues that can be automatically checked.

(3) Solicit peer feedback. After the submitted changes pass sanity testing, the author solicits reviewers to examine the change. Each reviewer is asked to provide feedback and a review score. In Gerrit, reviewers can provide one of five score values: "+2" indicates strong support for the change and approval for integration, "+1" indicates weak support for the change without approval for integration, "0" indicates abstention, "-2" indicates strong disagreement with the change and also blocks integration, and "-1" indicates weak disagreement with the change without blocking integration.

(4) Initiate integration request. Gerrit allows teams to codify code review and verification criteria that must be satisfied before changes can be integrated into the project git repositories. For example, the OPENSTACK integration policy specifies that an author needs to receive at least two +2 scores.[4] After satisfying the integration criteria, the author can initiate an integration request, which queues the change for integration.

(5) Execute integration tests. Code changes that are queued for integration are scanned by the integration testing system. The integration testing system runs a more rigourous set of tests than the sanity testing phase to ensure that changes that land in the project git repositories are clean. If the integration tests report failures, the change may not be integrated until the author uploads a revision of the change that addresses the failures.

(6) Final integration. Once the change passes integration testing, Gerrit automatically commits the change into the upstream (official) project git repositories.

Table 1 shows that the Gerrit-based code review processes of QT and OPENSTACK achieve high coverage rates of 95% and 97%, respectively. On occasion, the code review process is omitted due to a rush to integrate a critical change. However, the vast majority of changes undergo a code review. Moreover, changes are rarely approved for integration by only the author. Only 5% and <1% of changes were self-approved in QT and OPENSTACK, respectively.

Table 1 also shows that more people tend to be involved in the OPENSTACK review process than the QT one. The analyzed changes have a median of one reviewer in QT and a median of three reviewers in OPENSTACK.

3.2 Data Extraction

In order to conduct our case study, we extract data from the VCS, ITS, and code review databases of the studied systems. Figure 2 provides an overview of our data extraction approach. Below, we describe each step in our approach.

(DE1) Extract Issue Data

From each issue in the ITSs of the studied systems, we extract its unique identifier (IssueID), the timestamp from when the issue was reported (RepDate), and the issue type (Type, e.g., bug or enhancement). The IssueID is used to link issues to code changes, while RepDate and Type are used to detect false positive links in our datasets.

(DE2) Extract Code Properties

When extracting data from the VCS, we have three goals. First, we detect whether a change is potentially fix-inducing or not using the SZZ algorithm [39]. The SZZ algorithm identifies fix-inducing changes by: (a) identifying defect-fixing changes, (b) pinpointing the lines that are modified

[2] http://qt.digia.com/
[3] http://www.openstack.org/
[4] http://docs.openstack.org/infra/manual/developers.html#project-gating

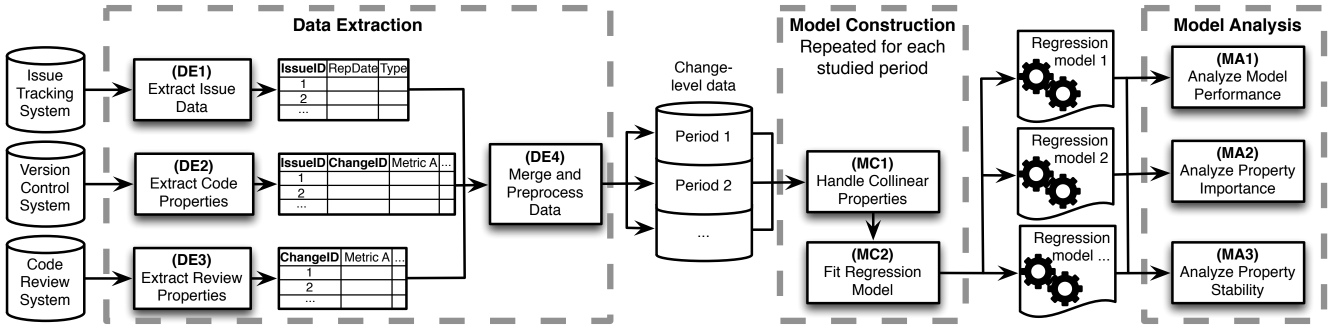

Data Extraction
Issue Tracking System
Version Control System
Code Review System
(DE1) Extract Issue Data
(DE2) Extract Code Properties
(DE3) Extract Review Properties
IssueID RepDate Type
1
2
...
IssueID ChangeID Metric A ...
ChangeID Metric A ...
(DE4) Merge and Preprocess Data
Change-level data
Period 1
Period 2
...
Model Construction
Repeated for each studied period
(MC1) Handle Collinear Properties
(MC2) Fit Regression Model
Regression model 1
Regression model 2
Regression model ...
Model Analysis
(MA1) Analyze Model Performance
(MA2) Analyze Property Importance
(MA3) Analyze Property Stability

Fig. 2. An overview of the design of our case study.

TABLE 2
A taxonomy of the studied families of code and review properties.

	Property	Description	Rationale
Size	Lines added	The number of lines added by a change.	The more code that is changed, the more likely that defects will be introduced [31].
	Lines deleted	The number of lines deleted by a change.	
Diffusion	Subsystems	The number of modified subsystems.	Scattered changes are riskier than focused ones because they require a broader spectrum of expertise [6, 14].
	Directories	The number of modified directories.	
	Files	The number of modified files.	
	Entropy	The spread of modified lines across files.	
History	Unique changes	The number of prior changes to the modified files.	More changes are likely more risky because developers will have to recall and track many previous changes [18].
	Developers	The number of developers who have changed the modified files in the past.	Files previously touched by more developers are likely more risky [24].
	Age	The time interval between the last and current changes.	More recently changed code is riskier than older code [10].
Author/Rev. Experience	Prior changes	The number of prior changes that an actor[1] has participated in.[2]	Changes that are produced by novices are likely to be more risky than changes produced by experienced developers [28].
	Recent changes	The number of prior changes that an actor has participated in weighted by the age of the changes (older changes are given less weight than recent ones).	
	Subsystem changes	The number of prior changes to the modified subsystem(s) that an actor has participated in.	
	Awareness[3]	The proportion of the prior changes to the modified subsystem(s) that an actor has participated in.	Changes that involve developers who are aware of the prior changes in the impacted subsystems are likely to be less risky than those that do not.
Review	Iterations	Number of times that a change was revised prior to integration.	The quality of a change likely improves with each iteration. Hence, changes that undergo plenty of iterations prior to integration may be less risky than those that undergo few [34, 42].
	Reviewers	Number of reviewers who have voted on whether a change should be integrated or abandoned.	Since more reviewers will likely raise more issues so that they may be addressed prior to integration, changes with many reviewers are likely to be less risky than those with fewer reviewers [36].
	Comments	The number of non-automated, non-owner comments posted during the review of a change.	Changes with short discussions may not be deriving value from the review process, and hence may be more risky [25, 26].
	Review window	The length of time between the creation of a review request and its final approval for integration.	Changes with shorter review windows may not have spent enough time carefully analyzing the implications of a change prior to integration, and hence may be more risky [34, 42].

[1] Either the author or reviewer of a change. [2] Either authored or reviewed. [3] New property proposed in this paper.

by defect-fixing changes using the `diff` command, and (c) traversing the version history to detect which change(s) introduced the modified lines using the `blame` command.

Our second goal is to extract the IssueIDs and ChangeIDs that are encoded in commit messages. The IssueIDs are used to link code changes to issue reports in the ITS, while the extracted ChangeIDs are used to link code changes to review records in the code review database. The merging and preprocessing steps are defined in more detail under Step DE4 below.

Our third goal is to compute, for each change, four families of code change properties that have been shown to share a relationship with the likelihood of inducing fixes in past work [17, 18, 19, 28]. We compute a broad range of code change properties that measure the change volume (Size), its dispersion across the codebase (Diffusion), the modified areas of the codebase (History), and the experience of the author (Author experience). Table 2 provides an overview of the studied properties. We describe each family below.

Size properties measure the volume of code that was modified within the change. For each change, we compute the number of *lines added* and *lines deleted*. These properties can be directly computed by analyzing the change itself.

Diffusion properties measure the dispersion of a change across a codebase. For each change, we compute the number of distinct names of modified: (1) *subsystems* (i.e., root directories), (2) directories (i.e., full directories within the codebase), and (3) files. To illustrate, consider the file `qtbase/src/dbus/qdbuserror.cpp`. The subsystem of this file is `qtbase` and the directory is `qtbase/src/dbus`.

History properties measure the past tendencies of modules that were involved with a change. For each change, we measure history properties using the changes that have: (a) modified the same files that are being modified by the change in question and (b) been recorded prior to the change in question. With this in mind, we compute the number of *unique changes* that have impacted the modified files in the past and the number of *developers* who have changed the modified files in the past. We also compute the *age*, which is the average of the time intervals between the last change that was made to each modified file and the change in question. Similar to prior studies [18, 28], we compute these history properties using all of the changes that were recorded prior to the change in question.

Files may be copied or renamed, which, if not handled carefully, may have an impact on our history properties. In this paper, we rely on the built-in capability of `git` to detect copied and renamed files. When a copied/renamed file is detected, we include the history of the source of the copy/rename operation in the analysis of the target file. Since `git` copy/rename detection is based on heuristics, the detection results are not without limitations. We discuss the potential impact that false positives and negatives may have on our history properties in Section 6.1.

Author experience properties estimate the expertise of the author of a change. Similar to the history properties, author experience properties are computed using past changes. The *experience* computes the number of past changes that an author has made to the codebase. *Recent experience* weighs the experience value of changes by their age. Similar to recent work [18], we compute recent experience by applying $\frac{1}{1+age}$, where *age* is measured in years. *Subsystem experience* is the number of past changes that an author has made to the subsystems that are being modified by the change in question. Finally, we propose author *awareness*—a new expertise property that measures the proportion of past changes that were made to a subsystem that the author of the change in question has authored or reviewed. Again, Section 6.1 discusses the impact that `git`'s copy/rename detection may have on our author experience measurements.

(DE3) Extract Review Properties

When extracting data from the review databases of our studied systems, we have two goals. First, we need to extract the ChangeIDs that are encoded in the review records. These IDs uniquely identify a review record, and can be used to link them with code changes in the VCSs. Next, we compute, for each review record, two families of properties that have been shown to share a relationship with the likelihood of inducing a fix [23, 42]. We compute code review properties that measure the experience of the reviewer(s) (Reviewer experience) and characteristics of the review process (Review). Table 2 provides an overview of the studied review properties. We describe each family below.

TABLE 3
The number of fix-inducing changes that survive each step of our filtering process.

#	Filter	QT			OPENSTACK		
		Total	%	Δ	Total	%	Δ
F_0	No filters	5,495	17%	-	4,423	16%	-
F_1	Code comments	5,407	17%	88	4,291	16%	132
F_2	Whitespace changes	5,133	16%	274	3,814	14%	477
F_3	Issue report date	4,158	13%	975	3,480	13%	334
F_4	Issue report type	3,242	10%	916	3,480	13%	0
F_{5a}	Too much churn	3,190	10%	52	3,474	13%	6
F_{5b}	Too many files	3,162	10%	28	3,461	13%	13
F_{5c}	No lines added	3,153	11%	9	3,450	14%	11
F_6	Period	2,891	11%	262	2,788	23%	662
F_7	Suspicious fixes	2,172	9%	719	1,830	15%	958
F_8	Suspicious inducing changes	2,002	8%	170	1,616	13%	214

Reviewer experience properties estimate the expertise of the reviewers of a change. Again, *experience* computes the number of past changes that a reviewer has reviewed, *recent experience* weighs experience by age, and *subsystem experience* focuses on the subset of past reviews that have changed the same subsystems as the change in question. Finally, *awareness* is the proportion of past changes that were made to a subsystem that the reviewer has authored or reviewed. Again, we refer readers to Section 6.1 for a discussion of the impact that our reliance on `git`'s built-in copy/rename detection may be having on our reviewer experience measurements.

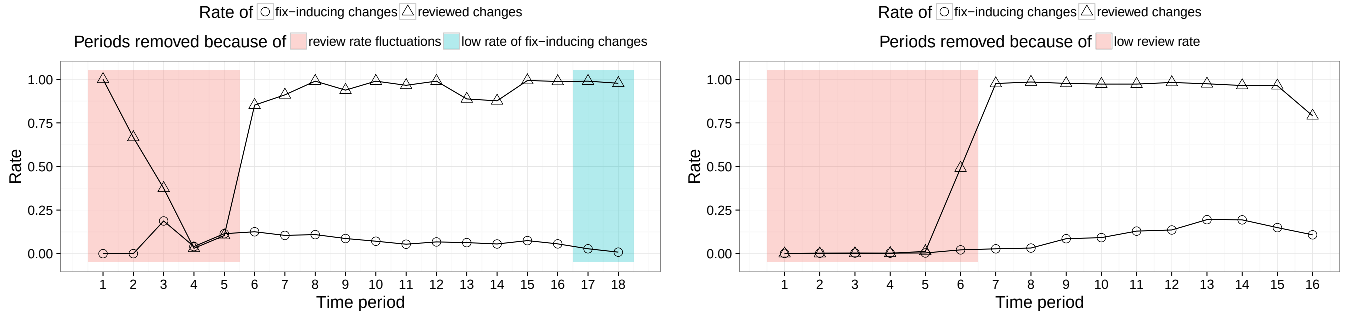
Review properties estimate the investment that developers have made in the code review process. *Iterations* counts the number of times that a code change was updated prior to integration. *Reviewers* counts the number of developers who approved a change for integration. *Comments* counts the number of reviewer comments that appear in a review record. *Review window* is the length of the time interval between when a review record was opened and when the changes were approved for integration.

(DE4) Merge and Preprocess Data

After extracting data from the VCSs, ITSs, and review databases of the studied systems, we merge them using the extracted identifiers (ChangeIDs and IssueIDs). This merging allows us to filter our data to mitigate false positives in our datasets. Table 3 shows the impact of applying each filter sequentially to our sets of fix-inducing changes.

First, as suggested by Kim *et al.* [21], we ignore potential fix-inducing changes that only update code comments (F_1) or whitespace (F_2). Next, we filter out potential fix-inducing changes that appear after the date that the implicated defect was reported (F_3) [39]. Then, we focus on only those defect-fixing changes where the issue type in the ITS is `bug` (F_4).

After merging the datasets and cleaning the fix-inducing changes, we preprocess our dataset to remove extremities. We ignore large commits—those that change at least 10,000 lines (F_{5a}) or at least 100 files (F_{5b})—because these commits are likely noise that is caused by routine maintenance (e.g., copyright updates). We also ignore changes that do not add any new lines (F_{5c}), since due to a limitation in the SZZ approach, only commits that introduce new lines have the potential to be flagged as fix-inducing.



(a) Qt

(b) OpenStack

Fig. 3. The rate of changes that are fix-inducing and the review coverage rate in the studied systems. The shaded areas are filtered out of our analysis due to large fluctuations in review coverage or drops in the rate of fix-inducing changes.

In order to study whether properties of fix-inducing changes are consistent, we stratify our data into time periods. We analyze period lengths of three and six months, since we find that at least three months are needed for our studied systems to accrue a substantial amount of data (i.e., 1,721–2,984 changes in Qt and 831–2,094 in OpenStack), while still yielding enough time periods to study trends (at least 18 periods in Qt and 16 periods in OpenStack). Although the primary goal of our paper is not to identify the optimal period length, we discuss the threat that our choice of period lengths imposes on our conclusions in Section 6.1.

Figure 3 shows the results of a preliminary analysis of the rates of (a) fix-inducing changes and (b) reviewed changes in each time period (F_6). We consider the rate of fix-inducing changes to counteract a limitation in the SZZ algorithm. The SZZ algorithm identifies defect-fixing changes, then traverses the version history to detect which change(s) had introduced the lines that were modified by the fix using the `blame` command. The SZZ algorithm needs future data to detect whether a change is fix-inducing or not. Hence, the later the period in the analyzed data, the lower the chances that a fix has been committed to address the problems in those commits. We do not filter out periods using a threshold value for the rate of fix-inducing changes in a period, but instead remove the latest periods where we begin to see a steady drop in the rate by analyzing Figure 3.

We also consider the rate of reviewed changes, since one of our criteria to select our subject systems is code review policy. However, even if the Qt and OpenStack projects have satisfied this criterion overall, the early periods of these projects may not. These early periods of adoption of code review are likely turbulent as changes are being made to development processes. To prevent the turbulent initial adoption period from impacting our reviewing measurements, we filter out periods where the review rate is low. Similar to the rate of fix-inducing changes, we do not select a threshold value for the rate of reviewed changes, but instead analyze Figure 3 in search of suspicious values.

Figure 3a shows that code review coverage was sporadic in the early periods of Qt development (periods 1–5). Furthermore, the rate of fix-inducing changes drops dramatically in the final two development periods on record (periods 17 and 18). Since this will introduce an additional confounding factor in our analysis, we elect to filter those periods out of our Qt dataset. Similarly, since Figure 3b shows that code review coverage was extremely low for the first six development periods of OpenStack (periods 1–6), we opt to filter those periods out of our OpenStack dataset.

Finally, our recent work proposes a framework for evaluating the results of SZZ-generated data [5]. We use the framework to highlight suspicious fixes (F_7), i.e., changes that fix more than the upper Median Absolute Deviation (MAD) of the number of fixed issues by a change for that project. Similarly, we use the framework to highlight suspicious fix-inducing changes as well (F_8), i.e., changes that induce more than the upper MAD of the number of fixes that were induced by a change for that project.

3.3 Model Construction

In this step, we use the preprocessed data to construct our JIT models. Figure 2 provides an overview of our model construction approach. We describe each step below.

(MC1) Handle Collinear Properties

Collinear code change properties will interfere with each other, distorting the modelled relationship between them and the likelihood of introducing defects. Thus, we remove collinear properties prior to constructing our JIT models.

Correlation analysis: We first check for code change properties that are highly correlated with one another using Spearman rank correlation tests (ρ). We choose a rank correlation instead of other types of correlation (e.g., Pearson) because rank correlation is resilient to data that is not normally distributed. We use a variable clustering analysis to construct a hierarchical overview of the correlation among the properties [37]. For sub-hierarchies of code change properties with correlation $|\rho| > 0.7$, we select only one property from the sub-hierarchy for inclusion in our models.

Redundancy analysis: In order to detect redundant code change properties, we fit preliminary models that explain each property using the others. We use the R^2 value of these models to measure how well each property is explained by the others. We use the implementation of this approach provided by the `redun` function in the `rms` R package, which iteratively drops the property that is most well-explained by the other properties until either: (1) no model achieves an $R^2 \geq 0.9$, or (2) removing a property would make a previously dropped property no longer explainable, i.e., its preliminary model will no longer achieve an $\mathrm{R}^2 \geq 0.9$.

(MC2) Fit Regression Model

We use a nonlinear variant of multiple regression modelling to fit our JIT models, which relaxes the assumption of a linear relationship between the likelihood of introducing defects and our code change properties. This relaxed fitting technique enables a more accurate fit of the data. We allocate a maximum of three degrees of freedom to each property (i.e., allowing the relationship to change directions once). Moreover, we fit our curves with restricted cubic splines, which fit smooth transitions at the points where curves change in direction (due to the curling nature of cubic curves). Finally, as suggested by Harrell Jr. *et al.* [12, 13], we ensure that we do not exceed a ratio of 15 events (i.e., fix-inducing changes) per degree of freedom spent, which mitigates the risk of *overfitting*, i.e., producing a model that is too specialized for the training dataset to apply to others.

The nonlinear variant of multiple regression modelling is often used in modelling of software engineering phenomena [26, 29, 46], especially for understanding the relationship between software development practices and software quality. However, using other techniques may lead to different conclusions. We discuss this threat to validity in Section 6.2.

3.4 Model Analysis

Next, we address our research questions by analyzing our JIT models. Figure 2 provides an overview of our model analysis approach. We describe each step below.

(MA1) Analyze Model Performance

To assess the accuracy of our JIT models, we compute threshold-independent measures of model performance. We avoid threshold-dependent measures like precision and recall, which depend on arbitrarily thresholds and are sensitive to imbalanced data.

The Area Under the receiver operator characteristics Curve (AUC) is a measure of a model's *discriminatory power*, i.e., its ability to differentiate between fix-inducing and clean changes. AUC is computed by measuring the area under the curve that plots the true positive rate against the false positive rate, while varying the threshold that is used to determine if a change is classified as fix-inducing or not. Values of AUC range between 0 (worst discrimination), 0.5 (random guessing), and 1 (perfect discrimination).

In addition to being a measure of discriminatory power, the Brier score is also a measure of a model's *calibration*, i.e., its absolute predictive accuracy. The Brier score is computed as $\mathit{Brier} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y_i})^2$, where N is the total number of changes; $y_i = 1$ if the i^{th} change is fix-inducing, $y_i = 0$ otherwise; and $\hat{y_i}$ is the probability of the i^{th} change being fix-inducing according to the JIT model under analysis. It is important to note that low Brier scores are desirable. Indeed, $\mathit{Brier} = 0$ indicates perfect calibration, while $\mathit{Brier} = 1$ indicates the worst possible calibration.

(MA2) Analyze Property Importance

We estimate the impact that each family of code change properties has on the explanatory power of our JIT models.

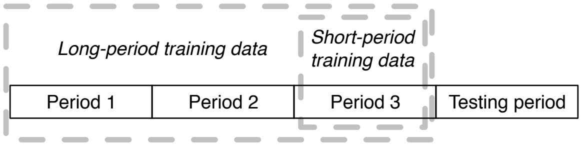


Fig. 4. An illustrative example of the types of the JIT model types.

In addition to each family being composed of several properties, each property has been allocated several degrees of freedom due to our nonlinear model construction approach (see Section 3.3). Each degree of freedom is represented with a model term. Hence, to control for the effect of multiple properties (and multiple terms), we jointly test the set of model terms for each family using Wald χ^2 maximum likelihood (a.k.a., "chunk") tests [11]. In order to make the Wald χ^2 values of multiple models comparable, we normalize them by the total Wald χ^2 score of the JIT model from which they originate. The larger the normalized Wald χ^2 score, the larger the impact that a particular family of properties has on the explanatory power of the JIT model under analysis.

(MA3) Analyze Property Stability

To assess the stability of the importance scores for a family of code change properties f over time, we compute the difference between the importance scores of f in a model that is trained using time period p and a future model that is trained using time period $p + x$, where $x > 0$.

4 Case Study Results

In this section, we present the results of our case study with respect to our research questions. For each research question, we present our approach and discuss the results.

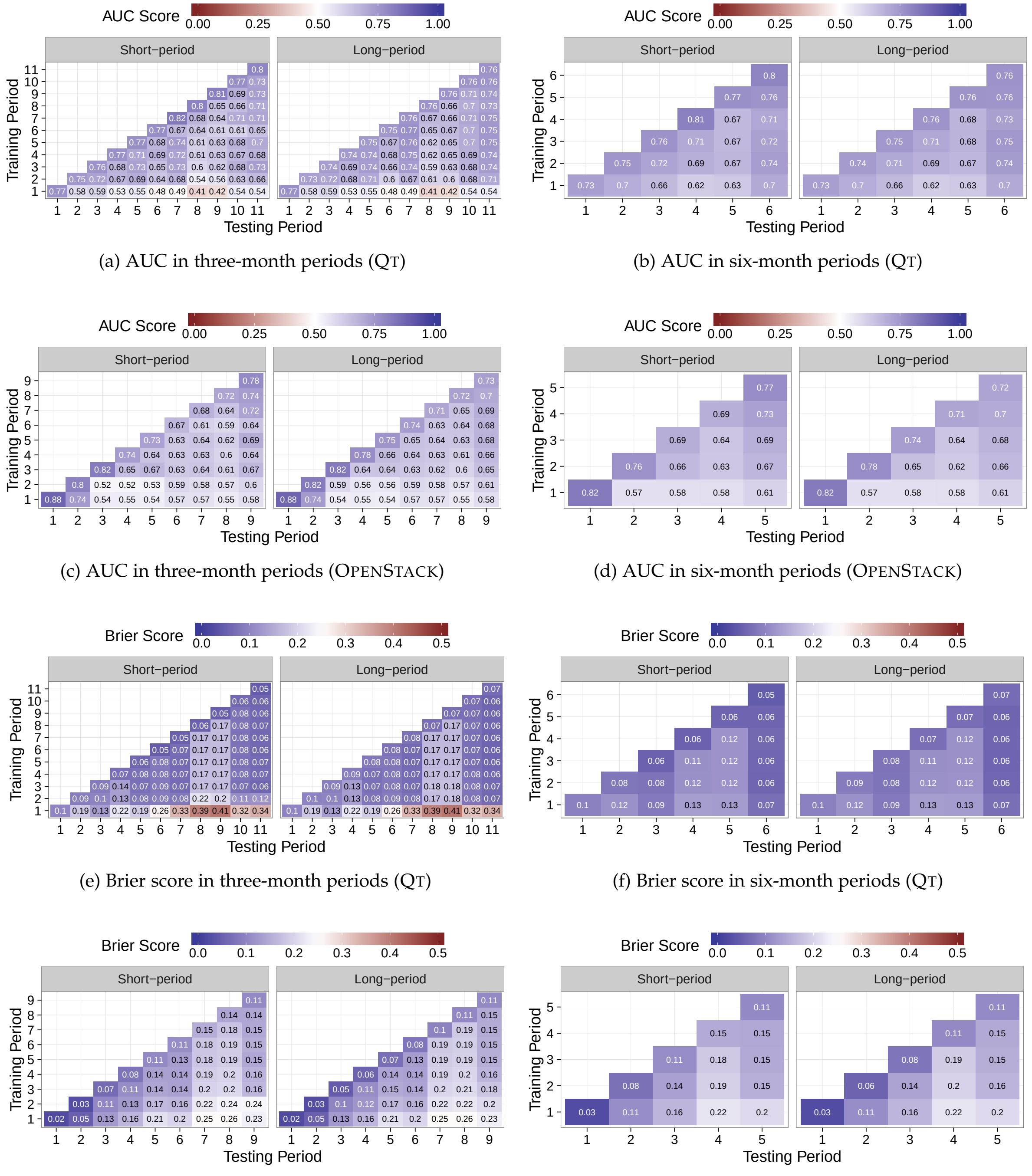
(RQ1) Do JIT models lose predictive power over time?

RQ1: Approach

To address RQ1, we study how quickly a JIT model loses its predictive power by training JIT models for each of the time periods (i.e., varying the training period), and measuring their performance on future periods. As illustrated in Figure 4, for each period, we train two types of models:

1) **Short-period models** are JIT models that are only trained using changes that occurred during one time period. We train short-period models because older changes may have characteristics that no longer apply to the latest changes.
2) **Long-period models** are JIT models that are trained using all of the changes that occurred during or prior to a particular period. We train long-period models because recent work suggests that larger amounts of training data tend to yield defect models that perform better, even when biases are introduced [35]. Hence, despite potential changes in the properties of fix-inducing changes, being exposed to additional data may improve the performance of our JIT models.

After training our JIT models, we test their performance when they are applied to the periods that occur after the last training period. As described in Section 3.4, we measure the



(a) AUC in three-month periods (QT)

(b) AUC in six-month periods (QT)

(c) AUC in three-month periods (OPENSTACK)

(d) AUC in six-month periods (OPENSTACK)

(e) Brier score in three-month periods (QT)

(f) Brier score in six-month periods (QT)

(g) Brier score in three-month periods (OPENSTACK)

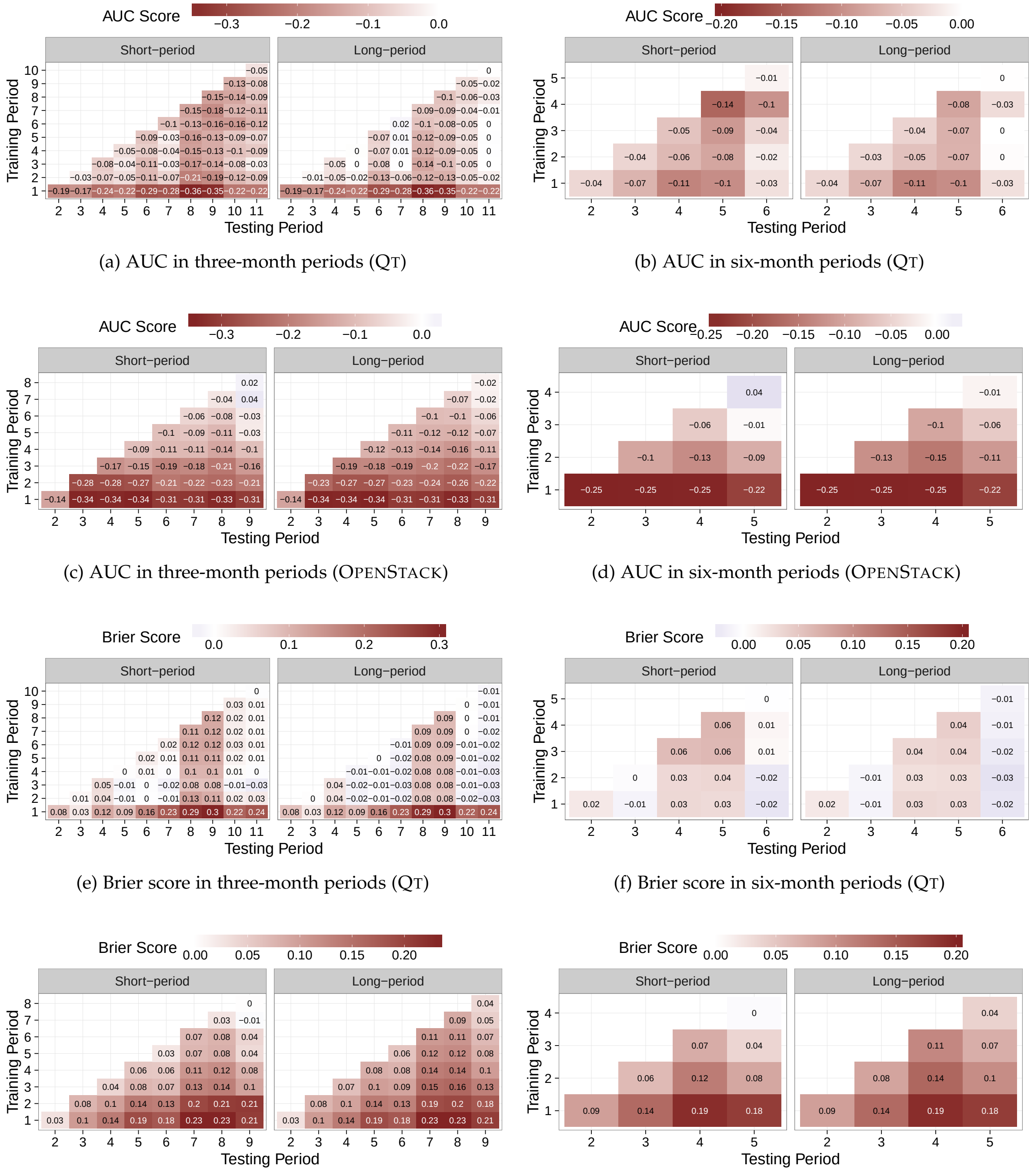
(h) Brier score in six-month periods (OPENSTACK)

Fig. 5. The predictive performance of JIT models as the studied systems age.

performance of our models using the AUC (discriminatory power) and the Brier score (calibration).

For example, Figure 4 illustrates that for a training period 3, the short-period model is trained using the changes that occurred during period 3, while the long-period model is trained using changes that occurred during periods 1, 2, and 3. These short-period and long-period models of period 3 are tested using periods 4 through to the last studied period. The AUC and Brier performance scores are computed for each testing period individually.

Finally, we plot the trends in AUC and Brier performance scores over time using heatmaps. In Figure 5, the shade of a box indicates the performance value, where blue shades indicate strong performance, red shades indicate weak performance, and the palest (white) shade indicates the performance that a random guessing model would achieve (on average). In Figure 6, the shade indicates the difference in AUC and Brier performance scores between the training and testing periods in question. Red, blue, and pale (white) shades indicate drops, improvements, and unchanged performance in the testing period, respectively.



(a) AUC in three-month periods (QT)

(b) AUC in six-month periods (QT)

(c) AUC in three-month periods (OPENSTACK)

(d) AUC in six-month periods (OPENSTACK)

(e) Brier score in three-month periods (QT)

(f) Brier score in six-month periods (QT)

(g) Brier score in three-month periods (OPENSTACK)

(h) Brier score in six-month periods (OPENSTACK)

Fig. 6. The delta in the estimated performance of JIT models as the studied systems age.

RQ1: Results

Models that are trained using periods that are closer to the testing period tend to outperform models that are trained using older periods. When we focus on the columns of Figure 5, the performance values tend to improve as the training period increases. In QT, the trend is especially prominent in testing periods 5 and later of the three-month period models, where at least one year has elapsed. For example, the long-period sections of Figures 5a and 5c show an AUC score improvement of 16–24 percentage points for QT and 6–12 percentage points for OPENSTACK by training using the most recent data (i.e., the period just prior to the testing period) instead of the data from period 1. Figures 5e and 5g also show a boost in Brier score of 12–28 and 6–7 percentage points for QT and OPENSTACK, respectively.

Although the magnitude is lower, similar improvement trends are observed in the six-month periods. Figures 5b and 5d show an AUC score improvement of 5–6 percentage points in QT and 9 percentage points in OPENSTACK. While Figure 5e shows an improvement in Brier score of only 1 percentage point for QT, Figure 5g shows a boost of 5 percentage points for OPENSTACK.

The improving trend tends to stabilize (at least) one period before the testing period. For example, Figures 5a and 5c show that the AUC improvement gained by adding the most recent three-month period to the long-period models of both studied systems is -1–2 percentage points. The -1 indicates a 1 percentage point loss in testing periods 6 and 8 of QT (0.68 and 0.67 in training periods 4 and 5) and OPENSTACK (0.64 and 0.63 in training periods 5 and 6), respectively. Figures 5b and 5d show similar trends of adding the most recent six-month period to the long-period models of both studied systems yields improvements of 0–3 percentage points. Figures 5e, 5f, 5g, and 5h show similar fluctuations of 0–2 percentage points in Brier score.

Our models lose a large amount of predictive power one year after being trained. Analysis of the rows of Figure 6 shows that after one year, there is often a large drop in the AUC and a sharp increase in the Brier score. Figures 6a and 6c show that our short-period models lose 3–22 and 3–34 AUC percentage points one year after being trained (i.e., *testing period* $=$ *training period* $+ 4$) in QT and OPENSTACK, respectively. The drops of only three AUC percentage points are observed in period 7 of QT and period 9 of OPENSTACK, which Figures 5a and 5c show have a tendency to yield strong performance scores (with performance scores noticeably higher than those of nearby rows). If those periods are omitted, the drops in AUC associated with our short-period models range from 11–22 percentage points in QT and 14–34 percentage points in OPENSTACK. Moreover, while training periods 1 and 2 of the long-period models in QT also suffer from large AUC drops of 22 and 13 percentage points, respectively, our long-period models that we train using periods 5 and later tend to retain their predictive power, only losing 1–9 AUC percentage points after one year. Similarly, after one year, the Brier score of our QT and OPENSTACK models drop by up to 11 and 19 percentage points, respectively (see Figures 6e and 6g).

Interestingly, Figures 6a, 6b, 6e, and 6h show that after losing a large amount of predictive power after one year, our QT and OPENSTACK models of periods 1 and 2 recover some predictive power in later periods. This suggests that the properties of fix-inducing changes in those later periods share some similarities with the fix-inducing changes of the earliest periods. We investigate this in RQ2.

Long-period JIT models do not always retain predictive power for longer than short-period JIT models. Note that the short-period and long-period models for period 1 are identical because there is no additional data added when training the long-period model. Hence, we only discuss the improvement in retention of predictive power for the JIT models that are trained using periods 2 and later.

We observe improvements in the predictive power of our QT models when they are tested in both the three-month and six-month settings. The rows of Figures 6a and 6b show that the long-period models of periods 2 and later retain more predictive power than their short-period counterparts in terms of AUC. For example, Figure 6a shows that the short-period model that was trained using period 3 drops 8 percentage points in AUC when it is tested on period 4, while the long-period model only drops 5 percentage points under the same circumstances. Moreover, Figure 6b shows that long-period models in the six-month period setting drop at most 3 percentage points of AUC when they are tested on the final period (period 6), while short-period models drop up to 10 percentage points. Figures 6e and 6f show that there is also an improvement in the retention of Brier score for QT.

Surprisingly, in OPENSTACK, Figures 6c and 6g show that the long-period models underperform with respect to the short-period models in terms of AUC and Brier score. This suggests that fix-inducing changes vary from period to period, with a sensitivity to more recent changes being more beneficial than accruing additional data in OPENSTACK.

A large proportion of the predictive power of JIT models is lost one year after being trained, suggesting that properties of fix-inducing changes may be in flux. JIT performance decay can be dampened by training JIT models using data that is recorded nearer to the testing period (i.e., more recent data).

(RQ2) Does the relationship between code change properties and the likelihood of inducing a fix evolve?

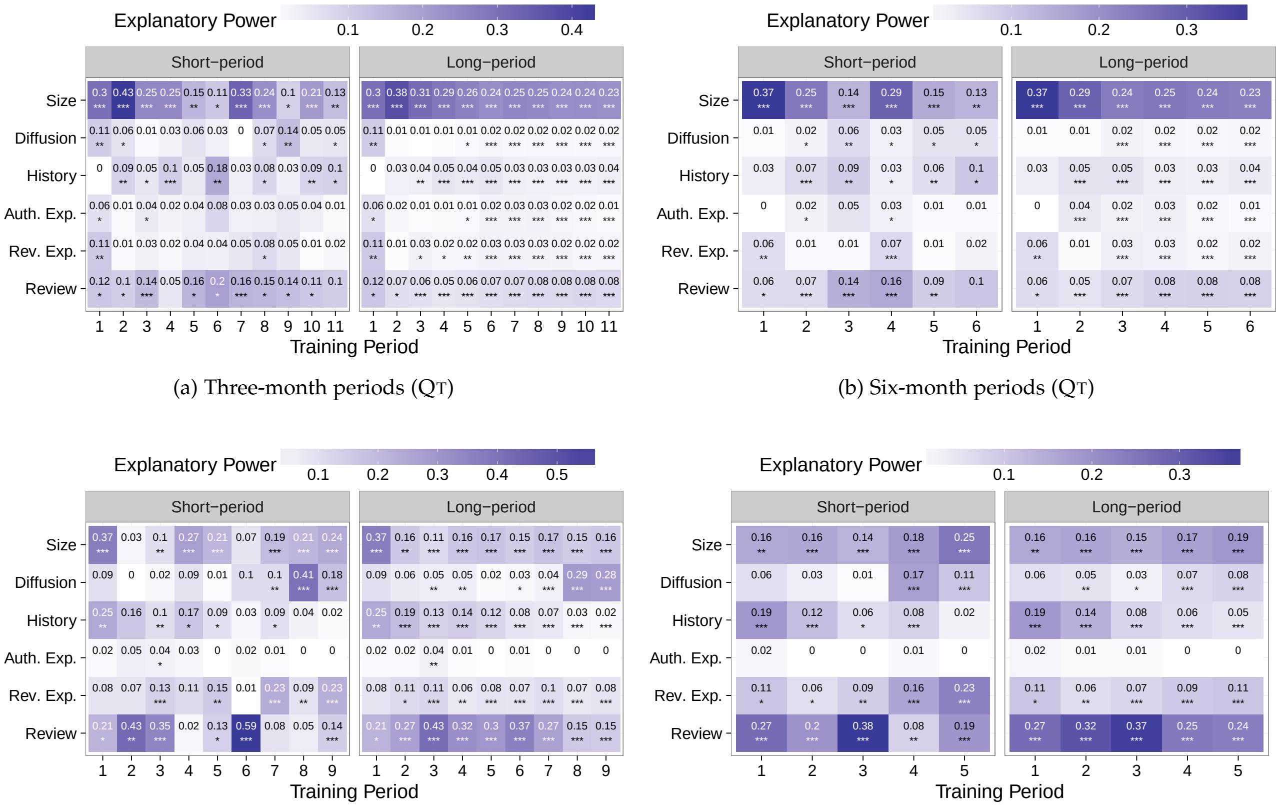
RQ2: Approach

In order to address RQ2, we compute the normalized Wald χ^2 importance scores (see Section 3.4) of each studied family of code change properties in each of our short-period and long-period JIT models. We study trends in the importance scores using heatmaps. The darker the shade of a given box, the more important that that family is to our model fit for that period. Figure 7 shows the results of our family importance analysis. In addition, we compute the p-values that are associated with the importance scores and denote the significance of each cell using asterisks (*).

RQ2: Results

The Size family is a consistent top-contributor to the fits of our models. Figure 7 shows that Size is often the most darkly shaded cell per period (columns). Figures 7a and 7b show that in QT, the Size family accounts for 23%–38% and 23%–37% of the explanatory power of our three- and six-month long-period JIT models, respectively. Moreover, Size accounts for 10%–43% and 13%–37% of the explanatory power of our three- and six-month short-period JIT models, respectively. Figures 7a and 7b also show that the Size family is the top contributor in 8 of the 11 three-month periods and 5 of the 6 six-month periods of our short-period JIT models, and all of the three- and six-month periods for our long-period JIT models. The contributed explanatory power of the Size family is statistically significant in all of the periods for our short- and long-period models in both three- and six-month settings ($p < 0.05$ and $p < 0.01$, respectively).

Similarly, Figures 7c and 7d show that in OPENSTACK, the Size family accounts for 11%–37% and 15%–19% of the explanatory power of our three- and six-month long-period JIT models, respectively. Size also contributes 3%–37% and 14%–25% of the explanatory power of our three- and six-month short-period JIT models, respectively. Moreover, Size is only a non-significant contributor to two of the nine three-month short-period models, providing a significant amount of explanatory power to all of the other three-month short- and long-period models, as well as all of the six-month short- and long-period models. In short, the magnitude and



(a) Three-month periods (QT)

(b) Six-month periods (QT)

(c) Three-month periods (OPENSTACK)

(d) Six-month periods (OPENSTACK)

Fig. 7. How the importance scores of the studied families of code change properties change over time. Shade indicates magnitude while asterisks indicate significance according to Wald χ^2 test, where: * $p < 0.05$; ** $p < 0.01$; *** $p < 0.001$.

consistency of the importance of Size suggests that advice about limiting change size is sound.

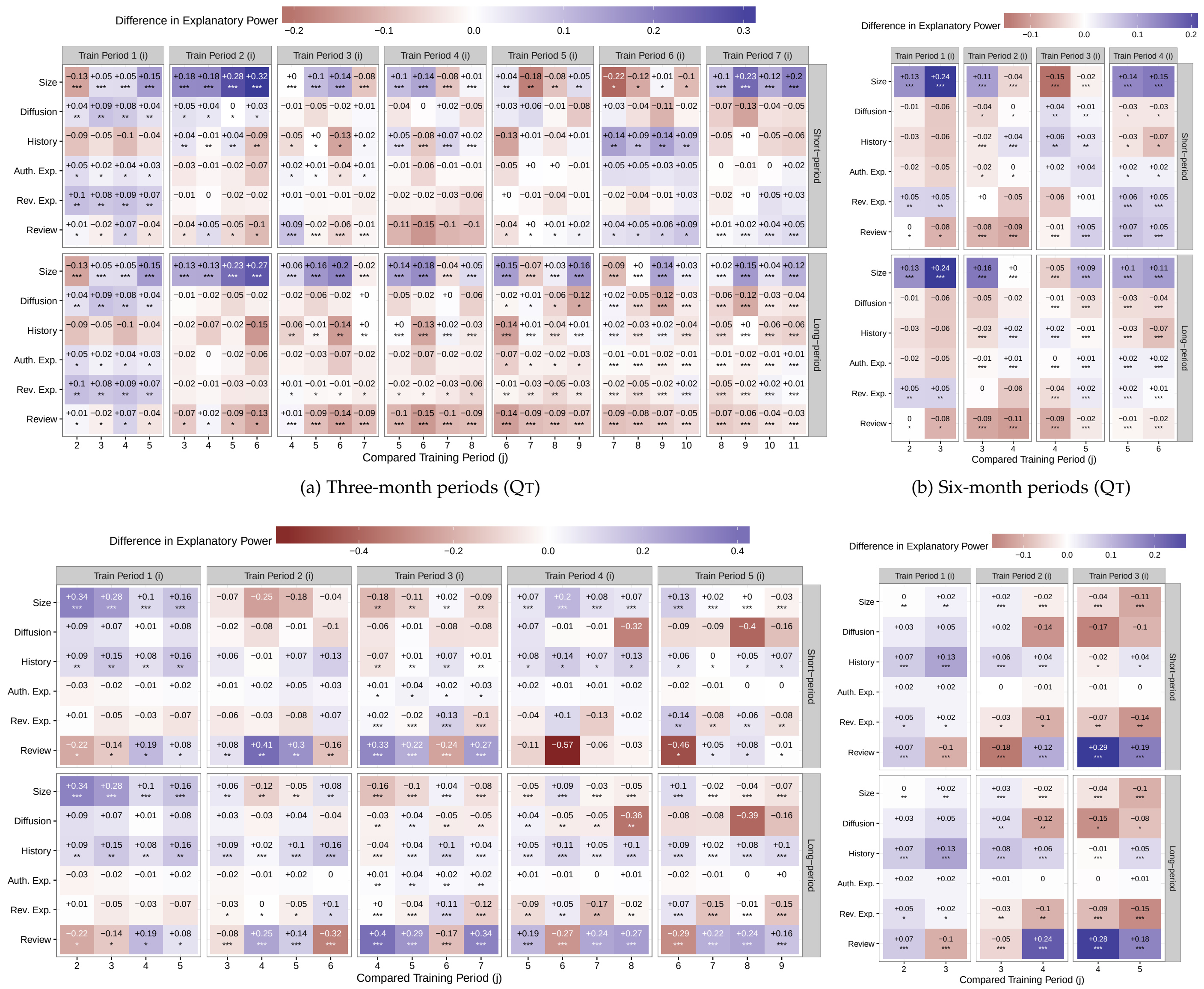
While the explanatory power of the Size family often dominates our models, there are periods where other families have larger importance scores. For example, Figures 7c and 7d also show that in OPENSTACK, the Review family achieves a larger importance score than the Size family in: (a) periods 2, 3, and 6 of the three-month short-term models, (b) periods 2–7 of the three-month long-period models, (c) periods 1–3 of the six-month short-term models, and (d) all periods of the six-month long-term models. Figures 7a shows that in QT, the Review family achieves a larger importance score than the Size family in periods 5, 6, and 9 of the three-month short-term models, while the History family achieves a larger importance score than the Size family in the short-period model of period 6.

Fluctuations in family importance are common in short-period models. Figure 7 shows that the shades of family importance (rows) vary more in short-period models than they do in long-period models. For example, in the three-month periods of QT (Figure 7a), the delta of the maximum and minimum explanatory power in the Size family is 0.33 ($0.43 - 0.10$) and 0.15 ($0.38 - 0.23$) in the short- and long-period settings, respectively. This is as one might expect, given that long-period models are eventually trained using much more data. Nonetheless, varying family importance in short-period models is another indication of fluctuation of the properties of fix-inducing changes.

Fluctuations in family importance also do occur in long-period models, albeit less often than short-period models. For example, Figure 7c shows that Diffusion, Reviewer Experience, and Review importance scores grow and shrink over the studied periods. Reviewer Experience increases and decreases in importance as OPENSTACK ages, while the importance of the Reviewer Experience and Diffusion fades in the early periods and grows again in later periods.

Our awareness measures do not contribute a significant amount of explanatory power. In this paper, we propose author and reviewer awareness—two new code change properties that compute how much of the prior change activity the author and reviewer has been involved with (see Table 2). Analysis of our model fits reveals that the awareness change properties did not improve our fits to a statistically significant degree. In fact, author awareness is often highly correlated with other Experience properties, suggesting that in the studied systems, author awareness does not provide any new information that existing Experience measures do not already capture. Despite the negative outcome of this empirical analysis, it may still be worthwhile to compute awareness values in future studies, since larger amounts of data may provide an opportunity for awareness values to differ from those of other Experience measures.

The importance of most families of code change properties fluctuate from period to period, suggesting that the properties of fix-inducing changes tend to evolve as projects age.



(a) Three-month periods (QT)

(b) Six-month periods (QT)

(c) Three-month periods (OPENSTACK)

(d) Six-month periods (OPENSTACK)

Fig. 8. The stability of the importance scores of the studied families of code change properties over time (*FISDiff*(f,i,j)).

(RQ3) How accurately do current importance scores of code change properties represent future ones?

RQ3: Approach

To address RQ3, we study the stability of the importance scores of the studied families of code change properties. To do so, we first select the short-period and long-period JIT models that we trained using each of the time periods. For each model, we compute the Family Importance Score (*FIS*) for each family f in: (a) the training period i ($FIS(f,i)$) and (b) the short-period JIT models in each period j in the year that follows after the training period (i.e., $FIS(f,j)$, where $j \in \{i+1, i+2, \cdots, i + \#\,\textit{periods per year}\}$). The $FIS(f,n)$ is the jointly tested model terms for all of the metrics belonging to a family f in the model of period n. Note that when computing $FIS(f,j)$ (i.e., those values that belong to "future" time periods), we use short-period JIT models instead of long-period models because short-period models should more accurately reflect the characteristics of the period in question. We then compute the differences between the importance scores in periods i and j using $FISDiff(f,i,j) = FIS(f,i) - FIS(f,j)$.

It is important to note that $FISDiff(f,i,j) > 0$ indicates that the importance of family f is larger in period i (the training period) than it is in period j, i.e., the JIT model that is trained using period i *overestimates* the future importance of family f. In these situations, quality improvement plans that are based on the importance of f at the end of period i would have a smaller impact than anticipated.

On the other hand, $FISDiff(f,i,j) < 0$ indicates that the importance of family f is smaller in period i than it is in period j, i.e., the JIT model that is trained using period i *underestimates* the importance of family f. In these situations, quality improvement plans that are based on the importance of f in period i may miss important families that would yield larger quality improvements than anticipated.

Similar to RQ2, we show the $\mathit{FISDiff}(f, i, j)$ values using heatmaps. We also compute the p-values that are associated with the importance score of the model of period i, since this is the model upon which quality improvement plans would be based. We again denote the statistical significance of these importance scores using asterisks (*). For presentation clarity, we only show $\mathit{FISDiff}(f, i, j)$ values for the year that follows the training of each model. As shown below, this one-year period is enough to demonstrate interesting trends.

RQ3: Results

Long-period models should be preferred for quality improvement planning. The short-period models of Figure 7 show several periods where trends in importance fluctuate sporadically. These spikes and troughs in importance scores can have a misleading impact on quality improvement plans when they are the source of data that is used to train JIT models. Figure 8 shows that long-period models tend to cope with these periods of sporadic fluctuation more gracefully than short-period models do.

For example, Figure 7a shows that the Size family has a large spike in importance in period 7 of the three-month setting of QT. Training periods 5 and 6 of Figure 8a show that the importance of the Size family is underestimated by 18 and 22 percentage points, respectively for testing period 7 in short-period models. Since the long-period models have a smoothing effect, these underestimates drop to 7 and 9 percentage points, respectively. When period 7 becomes the training period in Figure 8a, the impact of the Size family is overestimated by up to 23 percentage points in the short-period models. The maximum overestimate for the related long-period models is 15 percentage points.

Turning to OPENSTACK, Figure 7c shows that the Review family has a large spike in importance in period 6. Training periods 4 and 5 of Figure 8c show that the importance of the Review family is underestimated by 57 and 46 percentage points, respectively in the short-period models. Again, the smoothing effect of the long-period reduces the impact of this spike to 27 and 29 percentage points, respectively.

The six-month setting (Figures 8b and 8d) shows less severe over/underestimates, also suggesting that larger amounts of training data will smooth the impact of period-specific fluctuations on quality improvement plans.

The stability of many families of code change properties is project-sensitive. For example, the consistency of blue-shaded cells in Figure 8a indicates that the importance of the Size family is often overestimated in QT (median of 5%), especially in training period 2. Indeed, Figure 7a shows that importance of Size was largest at training period 2 of QT, trending downwards after that. Conversely, the consistency of red-shaded cells in Figure 8c indicates that the importance of the Size family is often underestimated in OPENSTACK (median of 2%), especially in training periods 2 and 3. Again, Figure 7c shows that importance of Size is growing as OPENSTACK ages from periods 2 and 3. Models that are trained using the early periods tend to underestimate the importance of the Size family in later periods.

Similarly, Figure 8a shows that the History family tends to be underestimated in QT (13 out of 24 periods for short-period and 20 out of 24 periods for long-period), while Figure 8c shows that the Review family tends to be overvalued in OPENSTACK (17 out of 20 periods for short-period and 19 out of 20 periods for long-period). Indeed, Figure 7a shows that the History family tends to grow more important as QT ages, while Figure 7c shows that the Review family tends to become less important as OPENSTACK ages.

When constructing quality improvement plans, one should favour large caches of data (e.g., long-period models, six-month periods). The importance of impactful families of code change properties like Size and Review are consistently under/overestimated in the studied systems.

5 PRACTICAL SUGGESTIONS

Based on our findings, we make the following suggestions for practitioners:

(1) **JIT models should be retrained to include data from at most three months prior to the testing period.** Our findings from RQ1 suggest that JIT models lose a large amount of predictive power one year after they are trained using the datasets that are collected from early periods. To avoid producing misleading predictions, JIT models should be retrained using more recent data often, at least more than once per year.

(2) **Long-term JIT models should be trained using a cache of plenty of changes.** Complementing recent work on module-level defect predictors [35], our findings from RQ1 indicate that larger amounts of data (i.e., our long-period models) can dampen performance decay in JIT models. Indeed, JIT models that are trained using more changes tend to retain their predictive power for a longer time than JIT models that are trained only using the changes that were recorded during most recent three-month period.

(3) **Quality improvement plans should be revisited periodically using feedback from recent data.** Our findings from RQ2 suggest that the importance of code change properties fluctuates over time. Moreover, our findings from RQ3 suggest that these fluctuations can lead to misalignments of quality improvement plans. Since RQ3 shows that long-period models achieve better stability, they should be preferred when making short-term quality plans. For long-term plans, we note that families for whom their importance scores were underestimated (overestimated) in the past tend to also be underestimated (overestimated) in the future. Hence, quality improvement plans should be periodically reformulated using the stability of the importance scores in the past to amplify or dampen raw importance scores.

6 THREATS TO VALIDITY

We now discuss the threats to the validity of our study.

6.1 Construct Validity

Threats to construct validity have to do the alignment of our choice of indicators with what we set out to measure. We identify fix-inducing changes in our datasets using the SZZ algorithm [39]. The SZZ algorithm is commonly used in defect prediction research [17, 18, 19, 23], yet has known limitations. For example, if a defect identifier is not recorded in the VCS commit message of a change, it will not be flagged as defect-fixing. To combat this potential bias, we select systems that use the Gerrit code reviewing tool, which tightly integrates with the VCS, allowing projects to automatically generate reliable commit messages based on fields of the approved code reviews. These commit messages can easily be processed. Hence, insofar as developers are carefully filling in code review records in Gerrit, our VCS data will be clean. Nonetheless, an approach to recover missing links that improves the accuracy of the SZZ algorithm [45] may further improve the accuracy of our results.

Similarly, we conduct a preliminary analysis of the rates of (a) fix-inducing changes and (b) reviewed changes in each time period to mitigate limitations of the SZZ algorithm and to prevent the turbulent initial adoption periods from impacting our reviewing measurements. Although the preliminary analysis is conducted to mitigate false positives in our datasets of fix-inducing changes, it may bias our results.

Copied and renamed files may truncate our history and experience code change properties. We rely on the copied and renamed file detection algorithms that are built into the `git` VCS. Since these algorithms are based on heuristics, they may introduce false positives or false negatives. To check how serious concerns of false positives are, we selected a sample of 50 files that were identified as copied/renamed from our dataset. In our opinion, all of these files were indeed copied/renamed, suggesting that false positives are not severely affecting our measurements. False negatives are more difficult to quantify. Nonetheless, a more accurate copy/rename detection technique may yield more accurate history and experience measurements.

More broadly speaking, our code change measurements are computed using various scripts that we have written. These scripts may themselves contain defects, which would affect our measurements and results. We combat this threat by testing our tools and scripts on subsamples of our datasets, and manually verifying the results.

Our results may be sensitive to the period length. We select period lengths of three and six months such that each period contains a sufficient amount of data to train stable defect models, while still yielding enough periods to study evolutionary trends. Moreover, this paper aims to study changes in the properties of fix-inducing changes over time, not to identify the "optimal" period length.

6.2 Internal Validity

Threats to internal validity have to do with whether other plausible hypotheses could explain our results. We assume that fluctuations in the performance and impactful properties of JIT models are linked with fluctuations in the nature of fix-inducing changes. However, other intangible confounding factors could be at play (e.g., changes to team culture, contributor turnover). On the other hand, we control for six families of code change properties that cover a broad range of change characteristics.

Our findings may be specific to nonlinear logistic regression models, and may not apply to other classification techniques. As suggested by our prior work [8], we are actively investigating other classification techniques like Random Forest. Preliminary results indicate that the findings of RQ1 can be reproduced in the Random Forest context. However, since the Wald χ^2 importance scores are not well-defined for such classifiers, a common importance score needs to be identified before RQ2 and RQ3 can be replicated.

6.3 External Validity

Threats to external validity have to do with the generalizability of our results to other systems. We focus our study on two open source systems. We chose strict eligibility criteria, which limits the systems that are available for analysis (see Section 3.1). Due to our small sample size, our results may not generalize to all software systems. However, the goal of this paper is not to build a grand theory that applies to all systems, but rather to show that there are some systems for which properties of fix-inducing changes are fluctuating. Our results suggest that these fluctuations can have a large impact on the performance of JIT models and the quality improvement plans that are derived from JIT models. Nonetheless, additional replication studies are needed to generalize our results.

7 Conclusions & Future Work

JIT models are trained to identify fix-inducing changes. However, like any method that is based on historical data, JIT models assume that future fix-inducing changes are similar to past fix-inducing changes. In this paper, we investigate whether or not fix-inducing changes are a moving target, addressing this following central question:

Do the important properties of fix-inducing changes remain consistent as systems evolve?

Through a longitudinal case study of the Qt and OpenStack systems, we find that the answer is no:

- JIT models lose a large proportion of their discriminatory power and calibration scores one year after being trained.
- The magnitude of the importance scores of the six studied families of code change properties fluctuate as systems evolve.
- These fluctuations can lead to consistent overestimates (and underestimates) of the future impact of the studied families of code change properties.

7.1 Future Work

Below, we outline several avenues for future work that we believe are ripe for exploration.

- **Measuring (and improving) the costs of retraining JIT models.** A continuous refitting solution, where JIT models are refit after each new change appears, may be the optimal choice from a performance standpoint. However, the costs of refitting JIT model must be quantified in order to check whether this continuous solution is truly the best option. These refitting costs are difficult to quantify, since they vary based on the model construction and analysis steps that need to be performed. For example, the approach that we adopt in this paper is semi-automatic. There are some manual steps in our correlation analysis (see Step MC-1 in Section 3.3) and model fitting process (see Step MC-2 in Section 3.3). These steps would be infeasible to repeat if one must refit models for every change. Future work may explore other modelling techniques where these steps could be automated (or omitted if collinearity and linearity assumptions are not of concern). In such a setting, continuous retraining may be a viable solution.

- **Analyzing other stratification approaches.** In this study, we stratify our data into time periods using three- and six-month period lengths. Other period lengths could be explored in future work. Furthermore, although time periods are intuitive for splitting data, there are other stratification approaches that could be used (e.g., a consistent number of changes, project releases).
- **Revisiting construct and internal validity concerns.** For example, the recovery of missing links between the individual repositories or a better technique for detecting copied or renamed entities may produce more accurate results.
- **Replication using systems that are developed in other contexts.** Historical data from other systems may provide other insights into the evolving nature of fix-inducing changes. For example, while we focused on two open source organizations in this paper, a natural avenue for future work would be to explore whether the same patterns emerge in proprietary software development organizations.

Replication

To facilitate future work, we have made the data that we collected and the scripts that we used to analyze them available online.[5]

Acknowledgments

This research was partially supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) and JSPS KAKENHI Grant Numbers 15H05306.

References

[1] G. Antoniol, K. Ayari, M. D. Penta, F. Khomh, and Y.-G. Guéhéneuc, "Is it a Bug or an Enhancement? A Text-based Approach to Classify Change Requests," in *Proc. of the IBM Centre for Advanced Studies Conference (CASCon)*, 2008, pp. 23:1–23:15.

[2] J. Aranda and G. Venolia, "The Secret Life of Bugs: Going Past the Errors and Omissions in Software Repositories," in *Proc. of the 31st Int'l Conf. on Software Engineering*, 2009, pp. 298–308.

[3] L. A. Belady and M. M. Lehman, "A model of large program development," *IBM Systems Journal*, vol. 15, no. 3, pp. 225–252, 1976.

[4] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu, "Fair and Balanced? Bias in Bug-Fix Datasets," in *Proc. of the 7th joint meeting of the European Software Engineering Conf. and the Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2009, pp. 121–130.

[5] D. A. da Costa, S. McIntosh, W. Shang, U. Kulesza, R. Coelho, and A. E. Hassan, "A Framework for Evaluating the Results of the SZZ Approach for Identifying Bug-Introducing Changes," *IEEE Transactions on Software Engineering*, vol. To appear, 2017.

[6] M. D'Ambros, M. Lanza, and R. Robbes, "An Extensive Comparison of Bug Prediction Approaches," in *Proc. of the 7th Working Conf. on Mining Software Repositories (MSR)*, 2010, pp. 31–41.

[7] J. Ekanayake, J. Tappolet, H. C. Gall, and A. Bernstein, "Tracking Concept Drift of Software Projects Using Defect Prediction Quality," in *Proc. of the 6th Working Conf. on Mining Software Repositories (MSR)*, 2009, pp. 51–60.

[8] B. Ghotra, S. McIntosh, and A. E. Hassan, "Revisiting the Impact of Classification Techniques on the Performance of Defect Prediction Models," in *Proc. of the 37th Int'l Conf. on Software Engineering (ICSE)*, 2015, pp. 789–800.

[9] E. Giger, M. D'Ambros, M. Pinzger, and H. C. Gall, "Method-Level Bug Prediction," in *Proc. of the 6th Int'l Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2012, pp. 171–180.

[10] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy, "Predicting Fault Incidence using Software Change History," *Transactions on Software Engineering (TSE)*, vol. 26, no. 7, pp. 653–661, 2000.

[11] F. E. Harrell Jr., *Regression Modeling Strategies*, 2nd ed. Springer, 2015.

[12] F. E. Harrell Jr., K. L. Lee, R. M. Califf, D. B. Pryor, and R. A. Rosati, "Regression modelling strategies for improved prognostic prediction," *Statistics in Medicine*, vol. 3, no. 2, pp. 143–152, 1984.

[13] F. E. Harrell Jr., K. L. Lee, D. B. Matchar, and T. A. Reichert, "Regression models for prognostic prediction: advantages, problems, and suggested solutions," *Cancer Treatment Reports*, vol. 69, no. 10, pp. 1071–1077, 1985.

[14] A. E. Hassan, "Predicting Faults Using the Complexity of Code Changes," in *Proc. of the 31st Int'l Conf. on Software Engineering (ICSE)*, 2009, pp. 78–88.

[15] H. Hata, O. Mizuno, and T. Kikuno, "Bug Prediction Based on Fine-Grained Module Histories," in *Proc. of the 34th Int'l Conf. on Software Engineering (ICSE)*, 2012, pp. 200–210.

[16] K. Herzig, S. Just, and A. Zeller, "It's Not a Bug, It's a Feature: How Misclassification Impacts Bug Prediction," in *Proc. of the 35th Int'l Conf. on Software Engineering (ICSE)*, 2013, pp. 392–401.

[17] Y. Kamei, T. Fukushima, S. McIntosh, K. Yamashita, N. Ubayashi, and A. E. Hassan, "Studying just-in-time defect prediction using cross-project models," *Empirical Software Engineering (EMSE)*, vol. 21, no. 5, pp. 2072–2106, 2016.

[18] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi, "A Large-Scale Empirical Study of Just-in-Time Quality Assurance," *Transactions on Software Engineering (TSE)*, vol. 39, no. 6, pp. 757–773, 2013.

[19] S. Kim, E. J. Whitehead, Jr., and Y. Zhang, "Classifying software changes: Clean or buggy?" *Transactions on Software Engineering (TSE)*, vol. 34, no. 2, pp. 181–196, 2008.

[20] S. Kim, H. Zhang, R. Wu, and L. Gong, "Dealing with Noise in Defect Prediction," in *Proc. of the 33rd Int'l Conf. on Software Engineering*, 2011, pp. 481–490.

[21] S. Kim, T. Zimmermann, K. Pan, and E. J. J. Whitehead, "Automatic Identification of Bug-Introducing Changes," in *Proc. of the 21st Int'l Conf. on Automated Software Engineering (ASE)*, 2006, pp. 81–90.

[22] S. Kim, T. Zimmermann, E. J. Whitehead Jr., and A. Zeller, "Predicting Faults from Cached History," in *Proc. of the 29th Int'l Conf. on Software Engineering (ICSE)*, 2007, pp. 489–498.

[23] O. Kononenko, O. Baysal, L. Guerrouj, Y. Cao, and M. W. Godfrey, "Investigating Code Review Quality: Do People and Participation Matter?" in *Proc. of the 31st Int'l Conf. on Software Maintenance and Evolution (ICSME)*, 2015, pp. 111–120.

[24] S. Matsumoto, Y. Kamei, A. Monden, K. ichi Matsumoto, and M. Nakamura, "An analysis of developer metrics for fault prediction," in *Proc. of the 6th Int'l Conf. on Predictive Models in Software Engineering (PROMISE)*, 2010, pp. 18:1–18:9.

[25] S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan, "The Impact of Code Review Coverage and Code Review Participation on Software Quality: A Case Study of the QT, VTK, and ITK Projects," in *Proc. of the 11th Working Conf. on Mining Software Repositories (MSR)*, 2014, pp. 192–201.

[26] ——, "An empirical study of the impact of modern code review practices on software quality," *Empirical Software Engineering*, vol. 21, no. 5, pp. 2146–2189, 2016.

[27] T. Menzies, A. Butcher, D. Cok, A. Marcus, L. Layman, F. Shull, B. Turhan, and T. Zimmermann, "Local versus Global Lessons for Defect Prediction and Effort Estimation," *Transactions on Software Engineering (TSE)*, vol. 39, no. 6, pp. 334–345, 2013.

[28] A. Mockus and D. M. Weiss, "Predicting Risk of Software

[5] https://github.com/software-rebels/JITMovingTarget

Changes," *Bell Labs Technical Journal*, vol. 5, no. 2, pp. 169–180, 2000.

[29] R. Morales, S. McIntosh, and F. Khomh, "Do Code Review Practices Impact Design Quality? A Case Study of the Qt, VTK, and ITK Projects," in *Proc. of the Int'l Conf. on Software Engineering (ICSE)*, 2015, pp. 171–180.

[30] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in *Proc. of the 27th Int'l Conf. on Software Engineering (ICSE)*, 2005, pp. 284–292.

[31] N. Nagappan, T. Ball, and A. Zeller, "Mining metrics to predict component failures," in *Proc. of the 28th Int'l Conf. on Software Engineering (ICSE)*, 2006, pp. 452–461.

[32] J. Nam, S. J. Pan, and S. Kim, "Transfer defect learning," in *Proc. of the Int'l Conf. on Software Engineering (ICSE)*, 2013, pp. 382–391.

[33] T. H. D. Nguyen, B. Adams, and A. E. Hassan, "A Case Study of Bias in Bug-Fix Datasets," in *Proc. of the 17th Working Conf. on Reverse Engineering (WCRE)*, 2010, pp. 259–268.

[34] A. Porter, H. Siy, A. Mockus, and L. Votta, "Understanding the Sources of Variation in Software Inspections," *Transactions On Software Engineering and Methodology (TOSEM)*, vol. 7, no. 1, pp. 41–79, 1998.

[35] F. Rahman, D. Posnett, I. Herraiz, and P. Devanbu, "Sample Size vs. Bias in Defect Prediction," in *Proc. of the 9th joint meeting of the European Software Engineering Conf. and the Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2013, pp. 147–157.

[36] E. S. Raymond, *The Cathedral and the Bazaar*. O'Reilly Media, 1999.

[37] W. S. Sarle, "The VARCLUS Procedure," in *SAS/STAT User's Guide*, 4th ed. SAS Institute, Inc., 1990.

[38] E. Shihab, A. E. Hassan, B. Adams, and Z. M. Jiang, "An Industrial Study on the Risk of Software Changes," in *Proc. of the 20th Int'l Symposium on the Foundations of Software Engineering (FSE)*, 2012, pp. 62:1–62:11.

[39] J. Śliwerski, T. Zimmermann, and A. Zeller, "When Do Changes Induce Fixes?" in *Proc. of the 2nd Int'l Workshop on Mining Software Repositories (MSR)*, 2005, pp. 1–5.

[40] M. Tan, L. Tan, S. Dara, and C. Mayeux, "Online Defect Prediction for Imbalanced Data," in *Proc. of the 37th Int'l Conf. on Software Engineering*, vol. 2, 2015, pp. 99–108.

[41] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, A. Ihara, and K. Matsumoto, "The Impact of Mislabelling on the Performance and Interpretation of Defect Prediction Models," in *Proc. of the 37th Int'l Conf. on Software Engineering (ICSE)*, 2015, pp. 812–823.

[42] P. Thongtanunam, S. McIntosh, A. E. Hassan, and H. Iida, "Investigating Code Review Practices in Defective Files: An Empirical Study of the Qt System," in *Proc. of the 12th Working Conference on Mining Software Repositories (MSR)*, 2015, pp. 168–179.

[43] ——, "Review Participation in Modern Code Review: An Empirical Study of the Android, Qt, and OpenStack Projects," *Empirical Software Engineering*, p. To appear, 2016.

[44] B. Turhan, "On the dataset shift problem in software engineering prediction models," *Empirical Software Engineering (EMSE)*, vol. 17, no. 1, pp. 62–74, 2012.

[45] R. Wu, H. Zhang, S. Kim, and S.-C. Cheung, "ReLink: Recovering Links between Bugs and Changes," in *Proc. of the 8th joint meeting of the European Software Engineering Conf. and the Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2011, pp. 15–25.

[46] M. Zhou and A. Mockus, "Does the Initial Environment Impact the Future of Developers?" in *Proc. of the 33rd Int'l Conf. on Software Engineering (ICSE)*, 2011, pp. 271–280.

[47] T. Zimmermann, N. Nagappan, and A. Zeller, "Predicting bugs from history," in *Software Evolution*. Springer, 2008, ch. 4, pp. 69–88.

[48] T. Zimmermann, R. Premraj, and A. Zeller, "Predicting defects for eclipse," in *Proc. of the 3rd Int'l Workshop on Predictor Models in Software Engineering (PROMISE)*, 2007, p. 9.

Shane McIntosh is an assistant professor in the Department of Electrical and Computer Engineering at McGill University, where he leads the Software Repository Excavation and Build Engineering Labs (Software REBELs). He received his Bachelor's degree in Applied Computing from the University of Guelph and his Master's and PhD in Computer Science from Queen's University, for which he was awarded the Governor General of Canada's Academic Gold Medal. In his research, Shane uses empirical software engineering techniques to study software build systems, release engineering, and software quality. More about Shane and his work is available online at http://rebels.ece.mcgill.ca/.

Yasutaka Kamei is an associate professor at Kyushu University in Japan. He has been a research fellow of the JSPS (PD) from July 2009 to March 2010. From April 2010 to March 2011, he was a postdoctoral fellow at Queens University in Canada. He received his B.E. degree in Informatics from Kansai University, and the M.E. degree and Ph.D. degree in Information Science from Nara Institute of Science and Technology. His research interests include empirical software engineering, open source software engineering and Mining Software Repositories (MSR). His work has been published at premier venues like ICSE, FSE, ESEM, MSR and ICSM, as well as in major journals like TSE, EMSE, and IST. He will be a program-committee co-chair of the 15th International Conference on Mining Software Repositories (MSR 2018). More information about him is available online at http://posl.ait.kyushu-u.ac.jp/~kamei/.