Effort-Aware Just-in-Time Defect Identification in Practice: A Case Study at Alibaba

Meng Yan*
School of Big Data and Software Engineering
Chongqing University, China
mengy@cqu.edu.cn

Xin Xia†
Faculty of Information Technology
Monash University, Australia
xin.xia@monash.edu

Yuanrui Fan
College of Computer Science and Technology
Zhejiang University, China
yrfan@zju.edu.cn

David Lo
School of Information System
Singapore Management University, Singapore
davidlo@smu.edu.sg

Ahmed E. Hassan
Software Analysis and Intelligence Lab (SAIL)
Queen's University, Canada
ahmed@cs.queensu.ca

Xindong Zhang
Alibaba Group
Hangzhou, China
zxd139932@alibaba-inc.com

ABSTRACT

Effort-aware Just-in-Time (JIT) defect identification aims at identifying defect-introducing changes just-in-time with limited code inspection effort. Such identification has two benefits compared with traditional module-level defect identification, i.e., identifying defects in a more cost-effective and efficient manner. Recently, researchers have proposed various effort-aware JIT defect identification approaches, including supervised (e.g., CBS+, OneWay) and unsupervised approaches (e.g., LT and Code Churn). The comparison of the effectiveness between such supervised and unsupervised approaches has attracted a large amount of research interest. However, the effectiveness of the recently proposed approaches and the comparison among them have never been investigated in an industrial setting.

In this paper, we investigate the effectiveness of state-of-the-art effort-aware JIT defect identification approaches in an industrial setting. To that end, we conduct a case study on 14 Alibaba projects with 196,790 changes. In our case study, we investigate three aspects: (1) The effectiveness of state-of-the-art supervised (i.e., CBS+, OneWay, EALR) and unsupervised (i.e., LT and Code Churn) effort-aware JIT defect identification approaches on Alibaba projects, (2) the importance of the features used in the effort-aware JIT defect identification approach, and (3) the association between project-specific factors and the likelihood of a defective change. Moreover, we develop a tool based on the best performing approach and investigate the tool's effectiveness in a real-life setting at Alibaba.

*also with Pengcheng Laboratory, Shenzhen, China.
†Corresponding author.

ESEC/FSE '20, November 8–13, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-7043-1/20/11…$15.00
https://doi.org/10.1145/3368089.3417048

CCS CONCEPTS

• **Software and its engineering** → *Software maintenance tools*;

1 INTRODUCTION

Software defect identification is a fundamental and active research question in software engineering. Researchers have proposed various defect identification approaches. The main benefit of defect identification is discovering defective software artifacts (i.e., package, file or change) in advance which can lead to effective use of the limited resources for software quality assurance.

There are two main types of defect identification approaches in terms of the identification granularity, i.e., module-level and change-level. Module-level is targeted at discovering defective modules (e.g., packages, files, or methods) [11, 22, 26, 28, 41, 50, 60]. Change-level is targeted at identifying defect-introducing changes [17]. Change-level defect identification has attracted an increasing interest in recent years, as it enables developers to identify defective artifacts at a finer and more timely granularity level (i.e., a change to source code) [7, 8, 14, 16, 17, 24, 36, 45, 52, 53, 55].

Change-level defect identification is commonly referred to as Just-in-Time (JIT) defect identification as identifies defect-introducing change at check-in time. A defect-introducing change is a software change that introduces one or several defects [37].

Compared with module-level defect identification, JIT defect identification has two main benefits: (1) **Identifying defective changes is more cost-effective.** As JIT defect identification is performed at a finer granularity, it enables developers to inspect a small amount of LOC (lines of code) to find latent defects. This can save a large amount of inspection effort compared with module-level (i.e., coarser) defect identification [17, 55]. (2) **Identifying defective changes is more timely.** As JIT defect identification is invoked at check-in time, it enables developers to inspect the changes for identifying defects while developers can still remember the context of these changes. This fresh context can lead developers to find the defects faster [36, 55].

Different changes would require different amounts of effort to inspect. Based on this intuition, *effort-aware JIT defect identification* was proposed to take into account the required effort to inspect the modified code for a change [12, 25]. Researchers have proposed

different effort-aware JIT defect identification approaches [7, 12, 13, 17, 55]. Such approach focuses on optimizing the number of defects that can be found given a fixed inspection budget (e.g., inspecting 20% of the modified LOC by all changes). It is more practical for practitioners, as it enables them to find more latent defects per unit of code inspection effort [55].

Due to the perceived benefits of JIT defect identification, several studies have attempted to apply JIT defect identification approaches into practice. Shihab et al. [36] conducted an industrial study to understand and identify risky changes at a large commercial company. Kamei et al. [17] evaluated the EALR approach on five commercial projects. Nayrolles et al. [31] proposed a two-phase approach (change metrics and clone detection) for intercepting risky changes and applied their approach to 12 Ubisoft projects. Among these industrial studies, only Kamei et al. [17] investigated the effectiveness of effort-aware JIT defect identification approach (i.e., EALR) in an industrial setting. The effectiveness of the follow-up effort-aware JIT defect identification approaches (e.g., CBS+ [13], OneWay [7] and unsupervised approaches [23, 55]) in an industrial setting has never been investigated. Furthermore, the effectiveness of supervised vs. unsupervised approaches in an industrial setting has never been explored.

In this paper, we investigate the effectiveness of effort-aware JIT defect identification approaches in an industrial setting. To that end, we conduct a case study to investigate the effectiveness of five state-of-the-art effort-aware JIT defect identification approaches (including three supervised and two unsupervised approaches) at a large commercial company. Additionally, we develop and deploy an effort-aware JIT defect identification tool based on the best performing approach to investigate its effectiveness as part of the real-life development process.

We partnered with Alibaba[1] to conduct this study for three reasons. First, Alibaba is one of the top ten most valuable and largest companies in the world [46]. It has accumulated diverse and mature software development data. Second, Alibaba's Development Efficiency department is very interested in effort-aware JIT defect identification. Hence, the department is willing to assign the resources and enforce the usage of our tool. Third, given that the headquarter of Alibaba is in Hangzhou, we can conveniently interview developers in person when needed.

Figure 1 presents an overview framework of our case study. In summary, the main contributions of this paper are as follows:

- We conducted a case study of effort-aware JIT defect identification on 14 Alibaba projects with 196,790 changes. The case study investigated the effectiveness of recently-proposed effort-aware JIT defect identification approaches on Alibaba projects. This paper is the first study to investigate the effectiveness of recently-proposed effort-aware JIT defect identification approaches in an industrial setting.
- We investigated the effectiveness of state-of-the-art supervised (i.e, CBS+ [13], OneWay [7] and EALR [17]) vs. unsupervised (i.e., LT [55] and Code Churn [23]) effort-aware JIT defect identification approaches on Alibaba projects. To the best of our knowledge, this paper is the first study to investigate the effectiveness of supervised vs. unsupervised effort-aware JIT defect identification approaches in an industrial setting.
- We investigated the important change-level features for effort-aware JIT defect identification on Alibaba projects and their differences compared with open source projects. Additionally, we investigated the association between project-specific factors and the likelihood of a defective change using a mixed effect model.
- We developed a tool based on the best performing approach on Alibaba projects. Using this tool, we conducted a user study to investigate the effectiveness of our tool when it is applied to the real-life industrial setting at Alibaba. This paper is the first study to investigate the effectiveness of deploying effort-aware JIT defect identification in a real-life development process.

Paper organization. Section 2 presents the background and related work on JIT defect identification. Section 3 presents our case study setup, including the projects that we decided to study, our approach along with the studied features and used evaluation measures. Section 4 presents our case study results. Section 5 presents the effectiveness of our tool when applied in a real-life industrial setting through a user study. Section 6 presents the threats to the validity of our work. Section 7 concludes and discusses possible avenues for future work.

2 RELATED WORK

This paper is a case study of effort-aware JIT defect identification in an industrial setting. Therefore, we divide our related work into two aspects: JIT defect identification and JIT defect identification in an industrial setting.

JIT defect identification. Several prior studies proposed approaches for JIT defect identification and performed empirical studies of these approaches. Śliwerski et al. [37] proposed an approach to identify defect-introducing changes and studied defect-inducing changes in two open-source projects. They found that the changes that are committed on Friday had a higher probability to be defect-inducing changes. Kim et al. [18] proposed a model for classifying a change as clean or buggy using various change features, including change log, source code, file names, change metadata and complexity metrics. Yin et al. [56] studied the relationship between defect-fixing changes and defect-introducing changes on several operating systems. Yang et al. [53, 54] proposed the use of more advanced modeling techniques for JIT defect identification, such as ensemble learning and deep learning.

With respect to effort-aware approaches, Kamei et al. [17] proposed the first effort-aware approach (called EALR) for JIT defect identification and conducted a large-scale empirical study. They used the number of lines modified by a change as the measure of the effort that is required to inspect a change. Following their study, many studies compared supervised vs. unsupervised effort-aware approaches for JIT defect identification [7, 12, 55]. Based on the latest studies, the CBS+ approach which is proposed by Huang et al [13] outperforms EALR [17], one way [7] and the best performing unsupervised approach LT [55].

[1] Alibaba is a Chinese multinational e-commerce, retail, internet, AI and technology company which is named as one of the world's most admired companies by Fortune [46].

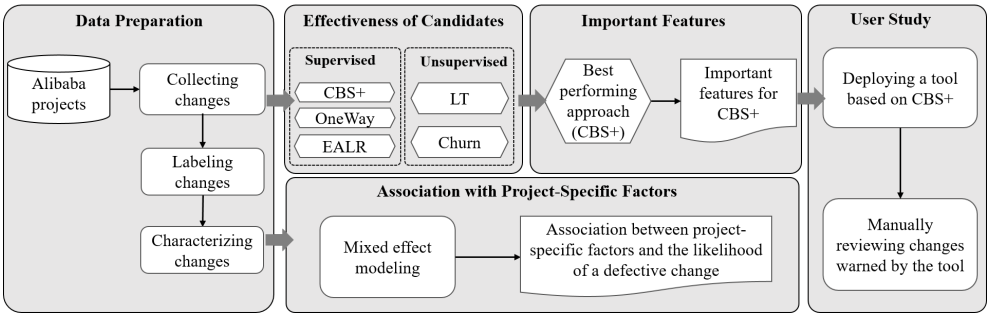

Figure 1: The overview of our case study

Inspired by the above-mentioned approaches, we choose five state-of-the-art approaches (i.e, CBS+ [13], OneWay [7], EALR [17], LT [55] and Code Churn [23]) as our candidate approaches for adoption at Alibaba. And we choose the effort-aware measures to evaluate our effectiveness since there are always limited code inspection resources in practice [12, 17, 55].

JIT defect identification in an industrial setting. Due to the importance of JIT defect identification, a few prior studies examined JIT defect identification approaches in an industrial setting. Mockus and Weiss [29] assess the risk of Initial Modification Requests (IMRs, ie., groups of code changes) of the 5ESS commercial project. Shihab et al. [36] conducted an industrial study to better understand risky changes. They developed a tool and asked the developers to label a change as risk or not at check-in time. Czerwonka et al. [5] present their experiences with CRANE, a tool used within Microsoft for change risk analysis. Kamei et al. [17] used both open source and commercial projects in their pioneering work on effort-aware JIT defect identification. Nayrolles and Hamou-Lhadj [31] proposed a two-phase approach (called CLEVER) for intercepting risky changes using code clone detection on 12 Ubisoft projects. Their first phase assesses the likelihood that an incoming change is risky or not. Their second phase uses clone detection to suggest fixes for risky changes that are identified in the first phase.

Our work is inspired by these above-mentioned studies but differs in several aspects. First, different from Mockus and Weiss [29], Shihab et al. [36], Czerwonka et al. [5] and Nayrolles and Hamou-Lhadj [31], our work focus on the effectiveness of **effort-aware** JIT defect identification approaches in an industrial setting. Different from Kamei et al. [17], we investigate the effectiveness of the **recently proposed** effort-aware JIT defect identification approaches (e.g., CBS+ [13], OneWay [7], LT [55] and Churn [23]) in an industrial setting. Furthermore, we are the first to investigate the effectiveness of **supervised vs. unsupervised** effort-aware approaches in an industrial setting.

3 CASE STUDY SETUP

3.1 Data preparation

Projects selection. We studied 14 Alibaba projects, which we refer to as P1 to P14 due to confidentiality reasons. A summary of these projects can be seen in Table 1. These projects are mainly written in Java. We select these projects as (1) they have released a few versions and are used by millions of users on a daily basis,(2) they differ in sizes and purposes, and (3) we are able to conveniently interview in person developers from these projects when needed.

Data labeling. Data labeling aims to label each historical change as defect-introducing or clean. To do this, we leveraged the SZZ algorithm proposed by Śliwerski et al. to identify defect-introducing changes from the historical changes of each project [37].

Studied features. We studied 14 change-level features from prior studies [7, 12, 13, 17, 23, 51, 55]. These features are grouped into five dimensions: diffusion (NS, ND, NF and Entropy), size (LA, LD and LT), purpose (FIX), history (NDEV, AGE and NUC) and experience (EXP, REXP and SEXP). We calculate these features by following the proposed approach by Kamei et al. [17].

3.2 Selected approaches

We choose five state-of-the-art effort-aware JIT defect identification approaches as our candidate approaches as they have already been shown to be effective for open source projects. In particular, we consider three supervised approaches (i.e., EALR [17], OneWay [7], and CBS+ [12, 13]) and two unsupervised approaches (i.e., LT [55] and Churn [23]). To make our paper self-contained, we briefly introduce these approaches:

Approach 1: EALR (Effort-Aware Linear Regression) is proposed by Kamei et al. [17]. First, EALR learns the relationships between the various change metrics of a change c (i.e., change features that are shown in Table **??**) and its defect-density. Second, for a new change in the testing dataset, EALR would predict its defect-density value using the learned model in phase 1. Finally, EALR sorts these changes in the testing dataset in descending order according to their predicted defect-density values.

Approach 2: OneWay is proposed by Fu and Menzies [7] which is a supervised approach built on the idea of a simple unsupervised approach that is proposed by Yang et al [55]. The basic idea of OneWay is to leverage the training data to automatically select the best metric for implementing the unsupervised model in Yang et al.'s work.

Approach 3: CBS+ (Classify Before Sorting) is proposed by Huang et al [13]. CBS+ is an extended version of CBS, which was proposed by Huang et al [12]. CBS+ leverages the advantages of both the supervised [17] and unsupervised approaches [55] by combining classification and sorting.

Approach 4: LT has been shown to be effective in Yang et al.'s work [55]. LT represents the lines of code in a file before the current

Table 1: Summary of the studied projects. "#Devs" represents the number of developers. "Med_size" represents the median change size. "Mean_size" represents the mean change size. In "Lifecycle" column, "Post-init" represents the post-initial version, "Pre-init" represents the pre-initial version.

Project	#Changes	#Defective	Def Ratio	#Branches	#Devs	#LOC	#File	Med_size	Mean_size	Users	Lifecycle	Time period
P1	5,218	871	16.7%	263	12	93,111	1,243	12	199	Internal	Post-init	2017.7-2018.7
P2	11,571	1,504	13.0%	821	36	189,742	2,380	9	84	Internal	Pre-init	2015.10-2018.7
P3	11,699	2,529	21.6%	180	41	56,691	847	12	229	Internal	Post-init	2014.7-2018.6
P4	30,777	5,642	18.3%	1,095	46	326,857	2,100	13	110	Internal	Post-init	2014.5-2018.6
P5	20,053	2,594	12.9%	875	67	285,887	3,986	9	87	External	Post-init	2015.3-2018.7
P6	3,707	285	7.7%	55	14	135,976	976	26	173	Internal	Pre-init	2016.11-2018.6
P7	3,640	294	8.1%	289	18	110,049	892	10	467	Internal	Pre-init	2016.9-2018.6
P8	18,053	1,810	10.0%	1,258	267	248,887	3,082	6	87	External	Post-init	2016.1-2018.7
P9	4,688	1,215	25.9%	605	30	33,896	330	3	61	External	Post-init	2016.12-2018.6
P10	18,888	1,616	8.6%	985	151	326,710	3,830	4	285	External	Pre-init	2016.1-2018.7
P11	20,760	1,998	9.6%	1,765	63	434,975	4,467	9	386	External	Post-init	2015.5-2018.7
P12	9,042	1,628	18.0%	633	10	130,017	1,288	7	116	External	Post-init	2015.12-2018.7
P13	18,424	1,869	10.1%	1,338	72	86,783	1,001	4	62	External	Post-init	2012.4-2018.7
P14	20,270	1,966	9.7%	1,279	27	422,540	4,323	11	135	External	Post-init	2016.7-2018.7
Total	196,790	25,821										

change. This unsupervised approach uses the feature LT to sort the changes in descending order according to the reciprocal of LT.

Approach 5: Churn[2] uses the churn metric that has been shown to be effective in a prior study by Liu et al. [23]. Churn represents the total lines of changed code (measured as $LA + LD$) by the current change. The unsupervised model Churn sorts the changes in descending order according to the reciprocal of $(LA + LD)$.

3.3 Data Preprocessing

We perform the following steps to preprocess the studied data when implementing EALR and CBS+ as described by Kamei et al. and Huang et al. [13, 17]. We do not perform the data preprocessing steps when implementing the other three selected approaches to ensure that we are following the same processes as prior studies [7, 23, 55].

1) Dealing with skewness. Most of the studied change features are highly skewed. To deal with data skewness, we apply a standard logarithmic transformation $ln(x + 1)$ to the values of each feature following Kamei et al. [17]. Notice that for the FIX feature, we do not apply the logarithmic transformation since this feature is a binary feature.

2) Dealing with correlated features. Kamei et al. and Huang et al. removed correlated features before building their models (i.e., EALR and CBS+) [13, 17]. Tantithamthavorn and Hassan note that correlated features may impact the interpretation of defect prediction models [40]. Hence, before learning models, we removed correlated features. Notice that the removal of correlated features is performed after the logarithmic transformation of the studied features. Hence, we detect correlation among the transformed features. Following the guidelines proposed by Harrel [9], we remove correlated features by following Kamei et al. [16, 17] and Li et al. [21].

3.4 Evaluation

Validation setting. To evaluate the effectiveness of the candidate approaches, we use a time-aware validation setting which ensures that the changes used for testing are always created later than the changes used for training, similar to prior studies [12, 55]. For each project, we first rank the changes in chronological order according to their creation time. Then, we divide all the changes into approximately 11 equal groups. Group 1 contains the changes that were created the earliest. For each group i (except for group 1, i.e., $i > 2$), we use changes in group 1 to group $i-1$ as a training dataset to build a model, then apply the model to identify the changes in group i. We use all the prior changes as a training set to ensure that the training set will have enough instances as suggested by a prior study [39]. Note that we remove the last group (i.e., group 11) from our evaluation experiment, since the changes in the most recent group may not be correctly labeled. Because the SZZ algorithm can only identify a defect-introducing change once the introduced defect has been fixed.

Performance measures. The key point of effort-aware JIT defect identification is to reduce the effort for code review. To that end, we mainly consider effort-aware measures by following prior studies [13, 55], namely *precision@20%*, *recall@20%*, *f1-score@20%*, *PCI@20%*, *IFA*, and *AUC*. The measures *precision@20%*, *recall@20%*, *f1-score@20%*, *PCI@20%* are calculated by considering a particular amount of inspection effort following Huang et al. [12, 13]. By default, we set the budget-effort as inspecting 20% of the total changed LOC in the testing set as done by prior state-of-the-art studies [12, 13, 55]. The measures *IFA*, and *AUC* are calculated without considering the amount of inspection effort. More detailed description for calculating these measures can be found in prior studies [13, 55].

Suppose in a testing dataset, we have M changes and N defective changes. After inspecting 20% of the total changed LOC in the testing dataset, we inspected m changes and found n defective changes. Then, these measures are calculated as follows:

Recall@20% is the proportion of inspected defective changes among all the actual defective changes[3], it is computed as n/N. *Precision@20%* is the proportion of inspected defective changes among all the inspected changes, it is computed as n/m. *F1-score@20%* is a summary measure that combines both *precision@20%* and *recall@20%*, it is computed as $\frac{2*Precision@20\%*Recall@20\%}{Precision@20\%+Recall@20\%}$. *PCI@20%* is the Proportion of Changes Inspected, it is computed as m/M. *IFA* is the number of Initial False Alarms encountered before we identify the first defective change, it is computed as the number of inspected non-defective changes before identifying the first defective change. *AUC* represents the area under the receiver operating characteristic (ROC) curve. In the ROC curve, the true positive rate (TPR) is plotted as a function of the false positive rate (FPR) across

[2]Other studies may also call this model as SBS (Sort by Size) [13] or ManualUp [59]

[3]Prior studies may also call the recall in an effort-aware model as ProfB@20% [14, 49] or ACC [17, 55].

all thresholds. AUC is threshold independent [3], robust towards the class distribution of the used dataset [19].

With respect to Recall@20%, Precision@20%, F1-score@20%, and AUC, a higher value is better. With respect to *PCI@20%* and *IFA*, a lower value is better. Because (1) a higher *PCI@20%* value indicates that developers need to inspect more changes under the same inspection effort. As a result, the additional effort might be required due to the context switches between changes and the additional communication overhead among developers [12, 27]. (2) A higher *IFA* value indicates that developers would inspect more false alarms before finding the first defect. As a result, developers would be frustrated and are not likely to continue inspecting other changes whenever the *IFA* is too high [12, 32].

3.5 Research questions

RQ1: How effective are state-of-the-art supervised and unsupervised effort-aware JIT defect identification approaches when applied on Alibaba projects?
RQ2: What are the important change-level features for effort-aware JIT defect identification at Alibaba? Do these important features differ between Alibaba and open source projects?
RQ3: Is there an association between project-specific factors and the likelihood of a defective change in Alibaba projects?

4 CASE STUDY RESULTS

4.1 RQ1: Effectiveness of studied approaches

Method. *First*, we extract the studied 14 change-level features. Using the SZZ algorithm, we prepare the needed dataset in our case study. *Second*, we implement the candidate supervised approaches namely CBS+, EALR and OneWay based on their description from the prior studies [7, 13, 17], and implement the candidate unsupervised approaches namely LT and Churn based on their description from the prior studies [13, 23, 55]. Note that we conduct the same preprocessing steps for these approaches as they are described [13, 17]. *Third*, we run these approaches using a time-aware validation setting. In this setting, we run each approach on each fold (i.e., including all changes within a two months period) and calculate the average performance across all the folds for each project. Note that we have a two-month gap between our training and testing dataset and we remove the last fold as described in Section 3. With respect to the unsupervised approaches, we only execute them on the testing set in order to keep a fair comparison with the results for the supervised approaches.

Additionally, to investigate whether the difference between two approaches is statistically significant, we employ the Wilcoxon signed-rank test [47] at a 95% significance level with a Bonferroni correction [1]. The Bonferroni correction is used to counteract the problem of multiple comparisons. We also employ Cliff's delta to measure the effect size. Cliff's delta is a non-parametric effect size measure that can evaluate the amount of the difference between two approaches. We use the following mapping for the values of the delta that are less than 0.147, between 0.147 and 0.33, between 0.33 and 0.474 and above 0.474 as *"Negligible (N)", "Small (S)", "Medium (M)", "Large (L)"* effect size, respectively [4].

Results. Tables 2 and 3 presents the performance of three supervised approaches considering our two groups of chosen performance measures respectively. Tables 4 and 5 presents the performance of two unsupervised approaches compared to the best performing supervised approach considering our two groups of chosen performance measures respectively.

In these tables, we list the statistical test results in the comparisons. The "Improvement" row represents the improvement ratio of each corresponding approach over the other approach. In terms of the performance measures where a higher value is better (i.e., *precision@20%*, *recall@20%*, *F1-score@20%* and AUC), the improvement of approach A over B is computed as $\frac{A-B}{B} * 100\%$. In terms of the performance measures where a lower value is better (i.e., *PCI20%* and *IFA*), the improvement of approach A over B is computed as $\frac{B-A}{B} * 100\%$, to indicate the ratio of decrease. The "p-value", "Cliff's delta" and "Effect size" rows represent the p-values, cliff's delta and effect sizes according to the Wilcoxon signed-rank test with a Bonferroni correction and Cliff's delta.

From the results shown in Tables 2, 3, 4 and 5 , we make the following observations:

(1) CBS+ statistically outperforms OneWay and EALR with a medium or large effect size in terms of *precision@20%*, *F1-score@20%*, *PCI20%*, and *AUC*. CBS+ improves OneWay and EALR by 131% and 95% in terms of *precision@20%*, by 74% and 60% in terms of *F1-score@20%*, by 62% and 50% in terms of *PCI20%*, and by 5% and 33% in terms of *AUC*. In terms of *IFA*, CBS+ statistically outperforms OneWay however there is no statistically significant difference with EALR (i.e., *p-value* > 0.05). In terms of *recall@20%*, there is no statistically significant difference between the three supervised approaches (i.e., *p-value* > 0.05).

(2) The average performance of CBS+ indicates that: if developers at Alibaba follow the CBS+ recommendation over a period of two months, they need to inspect 29% of the changes to identify 48% of the defective changes with a 37% precision. Additionally, they need to inspect nearly five changes before finding a truly defective change.

(3) In terms of the comparison between the best performing supervised approach (i.e., CBS+) and unsupervised approaches, CBS+ statistically outperforms LT and Churn with a large effect size in terms of *precision@20%*, *F1-score@20%*, *PCI@20%*, and *IFA*. CBS+ improves LT and Churn by 236% and 118% in terms of *precision@20%*, by 186% and 54% in terms of *F1-score@20%*, by 37% and 66% in terms of *PCI@20%*, and by 89% and 97% in terms of *IFA*. In terms of *AUC*, CBS+ significantly outperforms LT however there is no statistically significant difference with Churn (i.e., *p-value* > 0.05).

(4) In terms of *recall@20%*, although Churn statistically outperforms CBS+ with a large effect size, Churn sacrifices *precision@20%* to achieve a higher *recall@20%*. When considering *precision@20%* and *recall@20%* together (i.e., *F1-score@20%*), Churn no longer outperforms CBS+. In addition, Churn performs very poorly in terms of *IFA*, which may negatively impact developers' patience and confidence when reviewing changes.

(5) Churn achieves a high recall due to the skewed distribution of change sizes. We check the distribution of churn in each project. The results show that, for each project, the majority of changes modify a small number of LOC. Specifically, the churn of most

Table 2: The performance of the three supervised approaches in terms of our chosen performance measures which consider inspecting 20% of the changed LOC.

Project	Precision@20%			Recall@20%			F1-score@20%			PCI@20%		
	OneWay	EALR	CBS+	OneWay	EALR	CBS+	OneWay	EALR	CBS+	OneWay	EALR	CBS+
P1	0.23	0.24	**0.40**	**0.63**	0.53	0.53	0.33	0.31	**0.45**	0.83	0.69	**0.39**
P2	0.13	0.14	**0.27**	**0.56**	0.46	0.42	0.21	0.21	**0.32**	0.80	0.65	**0.30**
P3	0.20	0.21	**0.42**	**0.58**	0.50	0.46	0.30	0.29	**0.43**	0.77	0.64	**0.30**
P4	0.21	0.26	**0.49**	0.39	0.39	**0.49**	0.25	0.30	**0.49**	0.57	0.47	**0.31**
P5	0.13	0.15	**0.35**	0.42	**0.49**	0.47	0.19	0.22	**0.40**	0.61	0.64	**0.26**
P6	0.08	0.10	**0.22**	**0.46**	0.32	0.34	0.12	0.14	**0.23**	0.71	0.41	**0.22**
P7	0.11	0.13	**0.22**	**0.55**	0.42	0.40	0.18	0.19	**0.28**	0.73	0.49	**0.27**
P8	0.11	0.15	**0.30**	**0.65**	0.41	0.45	0.19	0.21	**0.35**	0.86	0.43	**0.21**
P9	0.38	0.44	**0.74**	0.56	0.47	**0.64**	0.44	0.44	**0.68**	0.69	0.53	**0.41**
P10	0.10	0.10	**0.26**	0.53	0.49	**0.54**	0.16	0.16	**0.35**	0.78	0.68	**0.28**
P11	0.12	0.15	**0.30**	**0.60**	0.55	0.49	0.19	0.23	**0.37**	0.83	0.62	**0.26**
P12	0.22	0.22	**0.48**	**0.63**	0.56	0.50	0.33	0.31	**0.48**	0.84	0.75	**0.32**
P13	0.11	0.20	**0.32**	0.51	0.48	**0.56**	0.17	0.27	**0.40**	0.80	0.40	**0.29**
P14	0.12	0.14	**0.37**	**0.61**	0.55	0.42	0.20	0.22	**0.39**	0.84	0.68	**0.20**
Average	*0.16*	*0.19*	***0.37***	***0.55***	*0.47*	*0.48*	*0.23*	*0.25*	***0.40***	*0.76*	*0.58*	***0.29***
Improvement	*131%*	*95%*		*-13%*	*2%*		*74%*	*60%*		*62%*	*50%*	
p-value	*<0.001*	*<0.001*		*>0.05*	*>0.05*		*<0.001*	*<0.001*		*<0.001*	*<0.001*	
cliff's delta	*0.87*	*0.83*		*0.50*	*0.00*		*0.81*	*0.81*		*1.00*	*0.98*	
Effect size	*L*	*L*		*L*	*N*		*L*	*L*		*L*	*L*	
Winner	**CBS+**			**OneWay**			**CBS+**			**CBS+**		

The best performing approach is highlighted in bold. "L", "M", "S" and "N" indicates "Large","Medium", "Small" and "Negligible" effect sizes respectively.

Table 3: The performance of the three supervised approaches in terms of our chosen performance measures while not considering the amount of inspection effort.

Project	IFA			AUC		
	OneWay	EALR	CBS+	OneWay	EALR	CBS+
P1	54.22	7.89	**5.00**	**0.82**	0.68	0.81
P2	109.89	9.11	**8.44**	**0.75**	0.62	0.74
P3	137.11	32.56	**3.78**	0.76	0.63	**0.79**
P4	130.22	**2.33**	3.67	0.72	0.61	**0.81**
P5	125.44	8.33	**4.67**	0.70	0.58	**0.81**
P6	45.11	7.89	**9.56**	0.73	0.58	**0.74**
P7	45.78	**5.56**	8.22	**0.71**	0.56	**0.71**
P8	281.78	**1.89**	4.33	0.77	0.54	**0.78**
P9	64.78	**0.33**	1.89	0.81	0.60	**0.91**
P10	359.11	12.89	**4.33**	0.78	0.61	**0.81**
P11	210.44	6.89	**4.78**	0.76	0.54	**0.81**
P12	130.89	**1.00**	4.33	**0.82**	0.65	**0.82**
P13	287.67	**1.67**	6.89	0.78	0.58	**0.83**
P14	191.89	12.56	**2.11**	0.77	0.58	**0.81**
Average	*155.31*	*7.92*	***5.14***	*0.76*	*0.60*	***0.80***
Improvement	*97%*	*35%*		*5%*	*33%*	
p-value	*<0.001*	*>0.05*		*<0.05*	*<0.001*	
cliff's delta	*1.00*	*0.14*		*0.45*	*1.00*	
Effect size	*L*	*N*		*M*	*L*	
Winner	**CBS+**			**CBS+**		

changes is less than 400 LOC. On the other hand, a small number of changes modify a very large number of LOC, thus, it is clear that the distribution of change size in each project is highly skewed.

(6) Most of our findings in terms of supervised vs. unsupervised approaches on Alibaba projects are consistent with the findings on open source projects by Huang et al. [13], i.e., CBS+ is the best performing approach. In comparison to Huang et al. [13], we further considered an additional unsupervised approach (Liu et al.'s Churn [23]) and an additional performance measure (AUC). Our novel finding on Alibaba projects is that although Churn achieves the highest recall@20% compared with other approaches and achieves a comparable AUC compared with CBS+, Churn underperforms in other performance measures (e.g., very poorly in terms of *IFA*) compared with CBS+.

The best performing supervised approach is CBS+. Although unsupervised approaches may achieve a higher recall@20%, they underperform in other performance measures compared with CBS+. Thus, we choose CBS+ as the approach to deploy at Alibaba.

4.2 RQ2: Important features

Method. For different projects, the most important features for effort-aware JIT defect identification may be different. Therefore, we calculate the feature importance for the CBS+ approach for each project. Notice that for the OSS projects, we also perform the two steps of data preprocessing as described in Section 3.3.

Different from RQ1, in this research question, we perform a 10-times 10-fold cross-validation following prior study [6]. In the time-aware validation (which in total includes ten folds), only less than 50% of the data is used for learning models in the first five folds. A small proportion of data may not reflect the overall data distribution. Hence, a model trained using such a small proportion of data may not lead to convincing feature importance results. On the other hand, in the 10-times 10-fold cross-validation, 90% of the data can be used to learn models in each fold. We consider that the feature importance results calculated using 10-times 10-fold cross-validation are more convincing than using the time-aware validation method.

In each fold, we build a CBS+ model using the training set. Then, we calculate the feature importance scores for the model. We apply the generic feature importance score that is proposed by Tantithamthavorn et al. [33, 43]. The generic feature importance score is calculated using the following two steps.

First, for each feature, we randomly permutate the values of the feature in the testing dataset. By doing so, we produce a testing dataset with the feature permutated and all other features as is. Second, for each feature, we calculate the difference between the F1-score@20% of the CBS+ model when applying it on the original testing dataset and the testing dataset with the randomly-permutated feature. A larger difference indicates that the feature is more important, and hence, we use the difference as the importance score of the feature.

Since CBS+ is an effort-aware JIT defect identification approach, the performance of CBS+ should be evaluated using effort-aware measures. F1-score@20% summarizes precision@20% and recall@20%. Thus, different from Tantithamthavorn et al. [43], we use F1-score@20% rather than misclassification rate to evaluate the performance of CBS+ in the above-mentioned second step for calculating feature importance score.

Table 4: The performance of the unsupervised approaches compared with CBS+ in terms of our chosen performance measures which consider inspecting 20% of the changed LOC.

Project	Precision@20%			Recall@20%			F1-score@20%			PCI@20%		
	LT	Churn	CBS+	LT	Churn	CBS+	LT	Churn	CBS+	LT	Churn	CBS+
P1	0.12	0.24	**0.40**	0.21	**0.67**	0.53	0.15	0.35	**0.45**	0.43	0.85	**0.39**
P2	0.12	0.14	**0.27**	0.29	**0.61**	0.42	0.17	0.22	**0.32**	0.46	0.83	**0.30**
P3	0.15	0.22	**0.42**	0.29	**0.69**	0.46	0.20	0.33	**0.43**	0.53	0.86	**0.30**
P4	0.18	0.25	**0.49**	0.27	**0.69**	0.49	0.21	0.36	**0.49**	0.45	0.85	**0.31**
P5	0.10	0.14	**0.35**	0.26	**0.63**	0.47	0.14	0.23	**0.40**	0.49	0.85	**0.26**
P6	0.05	0.08	**0.22**	0.18	**0.53**	0.34	0.08	0.13	**0.23**	0.34	0.81	**0.22**
P7	0.07	0.11	**0.22**	0.23	**0.66**	0.40	0.10	0.19	**0.28**	0.38	0.87	**0.27**
P8	0.09	0.12	**0.30**	0.32	**0.71**	0.45	0.13	0.20	**0.35**	0.54	0.89	**0.21**
P9	0.16	0.39	**0.74**	0.14	**0.71**	0.64	0.14	0.50	**0.68**	0.42	0.85	**0.41**
P10	0.07	0.11	**0.26**	0.27	**0.71**	0.54	0.11	0.19	**0.35**	0.52	0.89	**0.28**
P11	0.09	0.12	**0.30**	0.26	**0.64**	0.49	0.13	0.20	**0.37**	0.44	0.86	**0.26**
P12	0.15	0.24	**0.48**	0.28	**0.72**	0.50	0.19	0.36	**0.48**	0.49	0.88	**0.32**
P13	0.09	0.13	**0.32**	0.30	**0.66**	0.56	0.13	0.21	**0.40**	0.56	0.88	**0.29**
P14	0.09	0.13	**0.37**	0.24	**0.65**	0.42	0.13	0.21	**0.39**	0.45	0.86	**0.20**
Average	0.11	0.17	**0.37**	0.25	**0.66**	0.48	0.14	0.26	**0.40**	0.46	0.86	**0.29**
Improvement	236%	118%		92%	-27%		186%	54%		37%	66%	
p-value	<0.001	<0.001		<0.001	<0.001		<0.001	<0.001		<0.001	<0.001	
cliff's delta	1.00	0.85		1.00	0.94		1.00	0.69		0.96	1.00	
Effect size	*L*	*L*		*L*	*L*		*L*	*L*		*L*	*L*	
Winner	**CBS+**			**Churn**			**CBS+**			**CBS+**		

Table 5: The performance of the unsupervised approaches compared with CBS+ in terms of our chosen performance measures while not considering the amount of inspection effort.

Project	IFA			AUC		
	LT	Churn	CBS+	LT	Churn	CBS+
P1	42.22	51.89	**5.00**	0.78	**0.81**	**0.81**
P2	26.89	81.67	**8.44**	0.65	**0.74**	**0.74**
P3	38.00	138.89	**3.78**	0.73	0.77	**0.79**
P4	24.00	234.11	**3.67**	0.70	0.78	**0.81**
P5	28.00	213.22	**4.67**	0.71	0.78	**0.81**
P6	42.00	44.00	**9.56**	0.67	**0.76**	0.74
P7	25.44	45.33	**8.22**	0.64	**0.75**	0.71
P8	120.56	249.33	**4.33**	0.69	0.76	**0.78**
P9	16.89	94.67	**1.89**	0.81	**0.91**	**0.91**
P10	46.67	362.44	**4.33**	0.70	0.79	**0.81**
P11	25.89	207.11	**4.78**	0.67	0.76	**0.81**
P12	22.22	120.89	**4.33**	0.72	0.81	**0.82**
P13	152.67	283.89	**6.89**	0.74	0.79	**0.83**
P14	50.56	162.33	**2.11**	0.69	0.76	**0.81**
Average	47.29	163.56	**5.14**	0.71	0.78	**0.80**
Improvement	89%	97%		13%	3%	
p-value	<0.001	<0.001		<0.001	>0.05	
cliff's delta	1.00	1.00		0.81	0.33	
Effect size	*L*	*L*		*L*	*S*	
Winner	**CBS+**			**CBS+**		

Table 6: Summary of studied ten open source projects.

Project	#Changes	#Defective	Def Ratio
ActiveMQ	7,753	1,697	22%
Camel	17,374	2,957	17%
Derby	9,775	1,772	18%
Geronimo	16,152	2,309	14%
Hadoop Common	27,077	1,951	7%
HBase	14,581	3,323	23%
Mahout	2,762	539	20%
OpenJPA	6,368	810	13%
Pig	3,089	549	18%
Tuscany	21,595	2,157	10%
Total	126,526	18,064	14%

Then, we calculate the importance ranks of the features. In each run of 10-fold cross-validation, we have 10 importance scores for each feature. In total, we have 100 importance scores for each feature. Following prior studies [6, 42, 48], we apply the Scott-Knott ESD (SK-ESD) test on the feature importance scores. The SK-ESD test is an enhanced variant of the Scott-Knott test [34]. The SK-ESD enhances the Scott-Knott test in two folds [42]. First, the SK-ESD relaxes the assumption of normally distributed data (as required by the Scott-Knott test) by mitigating the skewness of input data. Second, the SK-ESD takes the effect size of the input data into consideration and merges any two statistically distinct groups that have a negligible effect size into one group.

Table 7: Ranks of the studied features in the Alibaba projects and OSS projects.

Projects	NS	ND	NF	Entropy	LA	LD	LT	FIX	NDEV	AGE	NUC	EXP	SEXP
P1	3	-	1	-	4	4	2	4	3	4	-	4	-
P2	5	-	1	-	4	4	4	2	4	5	3	3	4
P3	5	-	1	-	4	5	2	3	5	5	-	4	-
P4	8	-	1	-	6	8	2	6	7	8	3	4	5
P5	5	-	1	-	4	6	3	2	5	6	4	4	-
P6	4	-	1	-	4	3	4	2	4	4	4	4	-
P7	5	3	1	-	3	4	4	2	5	-	-	4	4
P8	8	6	1	-	5	7	9	2	3	8	4	7	-
P9	5	4	1	3	-	7	2	7	6	-	-	6	-
P10	4	4	1	5	3	5	6	2	5	-	-	5	5
P11	4	5	2	-	3	6	7	1	5	6	3	5	-
P12	7	6	1	-	3	7	2	4	5	7	-	7	7
P13	7	4	1	-	6	8	3	2	6	8	-	5	-
P14	6	-	1	-	4	6	2	1	5	5	3	3	-
ActiveMQ	3	-	1	-	8	8	4	6	7	9	-	5	2
Camel	6	-	1	7	6	6	2	5	3	4	-	4	-
Derby	7	-	1	8	4	8	5	6	7	2	7	3	-
Geronimo	6	-	1	-	8	7	3	5	5	9	-	4	2
HBase	5	-	1	6	7	5	5	2	4	5	3	6	7
Hadoop C.	7	-	1	6	9	8	3	4	2	9	-	5	9
Mahout	2	-	1	4	5	6	6	6	3	4	6	6	6
OpenJPA	5	-	1	-	5	4	2	5	4	4	3	1	2
Pig	1	-	1	4	4	5	5	6	2	3	-	3	-
Tuscany	5	-	1	-	6	3	4	5	7	6	6	2	2

Finally, we count the number of Alibaba projects and OSS projects where a feature is ranked as top-most and one of the top-3 important features. By doing so, we investigate whether the two groups of projects have different conclusions on the top-most and top-3 most important features for effort-aware JIT defect identification.

Results. Table 7 presents the ranks of the studied features in the Alibaba and OSS projects. The top-, second- and third-most important features are highlighted in an orange, green and blue background, respectively. The "-" symbol means that the feature is correlated with other features and it is removed. Note that for both Alibaba and OSS projects, we observe that REXP is correlated with other features (e.g., EXP). Thus, that feature is dropped. Table 8 presents the number of studied Alibaba and OSS projects where each of the studied features is ranked as a top-most or top-3 important feature.

From Table 8, we notice that considering the top-most important features, NF (i.e., number of files) is the dominant one for both Alibaba and open-source projects.

Considering the top-3 important features, we have different conclusions for the Alibaba and open-source projects. NF, FIX and LT are ranked as one of the top-3 important features for 14, 10 and 8 Alibaba projects (more than half of the studied Alibaba projects),

respectively. Hence, NF, FIX and LT are the most important features for the Alibaba projects. On the other hand, apart from NF, none of the other features is ranked as one of the top-3 important features for more than half of the studied OSS projects. From Table 7, we notice that apart from NF, EXP and SEXP are ranked higher than the other features. EXP is ranked as a top-most, second-most and third-most important feature in 1, 1 and 2 projects. SEXP is ranked as a second-most important feature in 4 projects.

The two groups of projects have the same top-most important feature (i.e., NF)—indicating that diffusion factors are the most important for effort-aware JIT defect identification for both groups of projects. But the two groups of projects show a difference in terms of the second- and third-most important features.

Table 8: Number of studied Alibaba and OSS projects where a feature is ranked as a top-most or top-3 important feature.

Features	Top-1		Top-3	
	Alibaba	OSS	Alibaba	OSS
NS	0	1	1	3
ND	0	0	1	0
NF	13	10	14	10
Entropy	0	0	1	0
LA	0	0	4	0
LD	0	0	4	1
LT	0	0	8	4
FIX	2	0	10	1
NDEV	0	0	2	4
AGE	0	0	0	2
NUC	0	0	4	2
EXP	0	1	2	4
SEXP	0	0	0	4

4.3 RQ3: Association analysis

Project-specific context factors. In total, we investigate 11 project-specific factors (as shown in Table 1) that are usually used in prior studies [57, 58, 60]. These factors are the number of changes (*Changes*), the number of defective changes (*Defective*), ratio of defective changes (*Def_ratio*), the number of branches (*Branches*), the number of developers (*Devs*), total lines of code (*LOC*), the number of files (*Files*), the median size of changes (*Median_size*), the mean size of changes (*Mean_size*), i.e., the size of a change represents the total changed number of lines (LA+LD) in the change, the user base (*Users*, i.e., "Internal" indicates that the project is used by internal users within the company, "External" indicates that the project is used by the external users), the lifecycle stage of the project (*Lifecycle*, i.e., ., pre-initial version or post-initial version), "Pre-initial" means the first version of the project has not been released. "Post-initial" means the first stable version of the project has been released, and the project has the opportunity to receive the users' reported bugs). For each numeric project-specific factor (i.e., all the project-specific factors except for *Users* and *Lifecycle*), we separated the values into four groups based on the first, second, and third quartiles (i.e., least, less, more, most), as suggested by prior work [16].

Mixed effect model. We use a mixed-effect logistic regression model [44] to investigate the association between project-specific factors and the likelihood of a defective change. A mixed-effect modeling approach is able to capture the variation of the interpretation of models among different projects [10]. Different from classic logistic regression models, a mixed-effect logistic regression model contains both fixed effects (independent factors at the change level, i.e., the explanatory factors) and random effects (independent factors at the project level, i.e., the context factors). Explanatory factors are used to explain the data at the change level, while context factors refer to the project-specific factors. Using explanatory variables and context factors, a mixed-effect model expresses the relationship between the outcome (i.e., the likelihood of a defective change) and the change-level features, while accounting for the different project-specific factors. The mixed-effect model is built as follows:

Step 1: Combining datasets of the studied projects. The mixed effect model analyzes the association between project-specific factors and the likelihood of a defective change in Alibaba projects. To that end, we combine all the changes from the studied projects. After that, we add the project-specific factors into the dataset of combined changes. Thus, the combined dataset contains change and project-specific factors for all the studied projects.

Step 2: Correlation and redundancy analysis. The highly correlated and redundant features would produce incorrect interpretations for a mixed effect model as noted by prior studies [10, 35]. In this step, we remove the highly correlated and redundant change factors in the combined dataset using the same approach as used in RQ2. As a result, Entropy, REXP, SEXP and NUC are removed from the combined dataset due to the high correlation. Furthermore, we remove redundant factors that can be predicted by combining other features. Our redundancy analysis found no redundant factors.

Step 3: Building a mixed effect model. There are two types of mixed-effect models: (1) random intercept models, and (2) random slope and intercept models [38]. A random intercept model has different intercepts for context factors and fixed slopes for explanatory factors. A random slope and intercept model has different intercepts for context factors and different slopes for explanatory factors. We choose to use the random slope and intercept model, because we assume that change-level features from different projects have different relationships with the likelihood of a defective change.

In our random slope and intercept model, we use project names as the random effect, and use NF as the random slope against the project in our model. In this manner, different projects have different baseline likelihood of a defective change, and NF has a different relationship with the likelihood of a defective change. We select NF as a random slope because NF is the most important change factor for our used approach (as we found in RQ2). Other change-level factors are used as fixed effects. We build the mixed-effect model using the `glmer` function of the `lme4`[4] R package.

Results. Table 9 presents the results of our mixed effect model. We discuss the results with respect to (1) the goodness-of-fit of our mixed effect model, (2) the association between project-specific factors and the likelihood of a defective change, (3) the association between change factors and the likelihood of a defective change.

Goodness-of-Fit. To measure how well our mixed effect model fits the combined dataset, we use the conditional coefficient of determination for generalized logistic regression and mixed-effect models (i.e., R^2 or R^2_{GLMM}) [15, 30]. We calculate such coefficient using the R^2_{GLMM} implementation of the `r.squaredGLMM` function of `MuMIn`[5] R package. This function reports two values: the

[4] https://cran.r-project.org/web/packages/lme4/lme4.pdf
[5] https://cran.r-project.org/web/packages/MuMIn/MuMIn.pdf

Table 9: Summary of the mixed effect model.

Type	Variable	Variance	Coef.	χ^2	Pr(> χ^2)
Random slope	NF	1.82	-	4572.88	***
Random intercept	Mean_size	0.01	-	57.20	***
	LOC	0.06	-	54.99	***
	Changes	<0.01	-	32.31	***
	Def_ratio	<0.01	-	16.62	***
	Median_size	0.07	-	3.10	o
	Users	<0.01	-	0.41	o
	Files	<0.01	-	0.08	o
	Lifecycle	0.08	-	<0.01	o
	Branches	0.13	-	<0.01	o
	Developers	<0.01	-	<0.01	o
	Defective	<0.01	-	<0.01	o
Fixed effect	FIX	-	0.86	2237.76	***
	LT	-	0.28	1496.30	***
	LA	-	0.43	1082.89	***
	AGE	-	-0.04	68.19	***
	NDEV	-	0.17	62.14	***
	LD	-	0.51	53.36	***
	NS	-	0.24	24.64	***
	EXP	-	-0.02	7.76	**
	ND	-	0.07	3.60	o

Statistical significance of χ^2:
o: p>= 0.05; *: p<0.05; **: p<0.01; ***: p<0.001

R^2 goodness of the model with just fixed effects (i.e., only using change factors), and the R^2 goodness of the full mixed effect model.

As a result, we found that the R^2 of our mixed effect model is 0.46, while the R^2 of the model with just fixed effects is 0.35. This finding indicates that our mixed effect model can explain the variability of the combined dataset by 46%. Additionally, our mixed effect model can explain the variability of the data 31% better than the model with just fixed effects.

Association between project-specific factors and the likelihood of a defective change. We use the χ^2 value of each project-specific factor (i.e., the context factors in our mixed effect model) to measure the association of a factor with the likelihood of a defective change as suggested by Bolker et al. [2]. Such χ^2 value is obtained from the Likelihood Ratio Test (LRT) that measures the association of a factor with the likelihood of a defective change. The larger the χ^2 value, the larger the association that a factor has with the likelihood of a defective change.

Table 9 presents a summary of the model statistics for our mixed effect model. The summary shows that the mean size of changes (*Mean_size*), total lines of Code (*LOC*), the number of changes (*Changes*), and the ratio of defective changes (*Def_ratio*) have a significant association with the likelihood of a defective change. These four project-specific factors have the largest χ^2 and a statistical significance.

This finding indicates that among our considered project-specific factors, the size factors (i.e., total LOC, total changes and mean size of changes) and the data distribution of defective changes of a project have the largest association with the likelihood of a defective change. For example, large projects tend to have a higher likelihood of a defective change. The findings in this subsection suggest that a mixed effect model considering project-specific factors can provide a deeper understanding of the characteristics of defect-introducing changes.

Association between change factors and the likelihood of a defective change. In terms of the change factors, Table 9 shows that the NF, FIX, LT, and LA have the highest association with the likelihood of a defective change. Note that in RQ2, we investigated the important change factors that impact the performance (i.e., *F1-score@20%*) of CBS+. In this RQ, we investigate the association between different factors (including project-specific factors and change factors) and the likelihood of a defective change on the combined projects of Alibaba. Our finding in this RQ suggests that the NF, FIX, LT, and LA factors are highly associated with the likelihood of defective changes. Such finding is consistent with the finding in RQ2 (i.e., the important change factors for CBS+).

Project-specific context factors have an association with the likelihood of a defective change in Alibaba projects. The size factors (i.e., total LOC, total changes and mean size of changes) and the data distribution of defective changes of a project have a significant association with the likelihood of a defective change for the Alibaba projects.

5 HUMAN STUDY

In this section, we wish to investigate the effectiveness of our effort-aware JIT defect identification tool when it is applied in a real-life industrial setting. To do so, we developed an effort-aware JIT defect identification tool based on CBS+. The extent of developers' approval of the warnings by our JIT defect identification tool is unknown. We believe that seeking developers' opinions would provide a practical view of the effectiveness of our effort-aware JIT defect identification approach. Thus, we used this tool to investigate its effectiveness in a real-life industrial setting. In summary, we focus on the following central question in our discussion:

How effective is our effort-aware JIT defect identification tool in a real-life industrial setting at Alibaba?

Method. Our user study was conducted on one project. We only choose one project because conducting a user study on several projects would cost a large amount of developers' effort and time. Developers in the user study were each asked to review a list of changes and we performed a personal interview whenever needed. We choose project P1 as described in Table 1, since the developers in this project are available and willing to do the user study. In this project, we have five participating developers who are active developers in this project. They vary in experience levels, ranging from one to seven years of work experience in the company.

Responses on each change. For each change in the user study, the participating developers were asked to mark each change as defective, clean, or unknown. Before performing the user study, the participating developers were told that *defective* means that a change will introduce risks (e.g., may lead to future bugs). Such risks would cause a negative impact on their products and processes. Developers believe that additional attention is needed in the form of careful code or design reviewing or more testing. *Clean* means that the developer considers the change to be of low risk. *Unknown* means that the developer has no strong opinion towards that change or that the developer does not have enough experience to make a choice. The detailed steps of the user study are:

Step 1: Taking the changes in the recent month (i.e., 205 changes from 20 June to 20 July, 2018) as the reviewing scope, we created a list of defective changes using our tool. And we assumed an inspection effort budget that is 20% of the changed LOC, following prior work (e.g., [12]). The total changed LOC of the warned changes cannot be larger than the inspection budget (i.e., 20% of total changed

LOC in the testing dataset). Note that we removed the truly defective changes that have already been marked as defective by SZZ before 20 July (i.e., 11 changes), since these defective changes have been fixed. As a result, our tool warned about 33 changes being defective.

Step 2: We sent a survey with the same list of these 33 changes to 5 participating developers on 21 July, 2018. We asked them to finish the survey separately. During the survey, developers were asked to inspect the diff of each change on the code review system. We also provided the code review url of each change on the code review system. On the url, developers can inspect detailed diff information of each change.

Step 3: For each change, each participating developer was asked to mark whether they consider a change as defective, clean, or unknown. After collecting the responses by the 5 participating developers, we synthesized their responses.

Table 10: Synthesized responses of user study 1. "Def" means defective, "Cle" means clean, "Dis" means disagree and "Unk" means unknown.

	#Def	#Cle	#Dis	#Unk
Synthesized responses	11	21	1	0

Synthesizing the responses of the 5 developers. Different developers may have different responses on each change due to their various understanding and experience. Thus, we have to synthesize their responses to obtain a more comprehensive result. Following the idea of prior study [20], we synthesize the responses by the 5 developers as follows:

(1) For a defective classification, at least 2 respondents indicated that the change is defective, and the number of defective responses must be larger than that of the clean responses (e.g., a change with 2 defective, 1 clean and 2 unknown respondents would be considered as a defective change); (2) For a clean classification, the same process as (1) is used but in reverse; (3) For an unknown classification, at least 3 responses indicated that the change is unknown.(4) Otherwise, if there are 2 defective responses, 2 clean responses and 1 unknown response, the change is marked as "disagreed".

Results. Table 10 presents the synthesized responses of the five participating developers. Among the 33 changes in our survey, 11 changes are determined as defective, 21 changes are determined as clean and 1 change is determined as disagreed. This indicates that developers concur with our tool's defective warning 33% of the time.

The results indicate that our tool helped developers focus on only 33 of the 205 changes. These 33 changes would cost 20% of the entire effort required to inspect all the changes. As a result, 11 defective changes from the 33 warned changes by our tool are correctly identified.

From the user study, we find that our tool helps developers focus on a small number of the warned changes. The tool's warned defective changes are correct in the 33% of the time.

6 THREATS TO VALIDITY

Threats to internal validity relate to potential errors in our implementation. One potential threat to validity is the potential errors in our approach implementation. To mitigate the threat, we implemented the selected five approaches following their original description [7, 13, 17, 23, 55]. We also double-checked the implementation and fully tested our code, still there could be errors that we did not notice.

Threats to external validity relate to generalizability of our results. We analyzed 14 projects and interviewed five developers at Alibaba. All our participating developers were volunteers and they have their own daily jobs at Alibaba. There is the possibility of project and developer selection bias in our user study. Additionally, since only one company is selected in our case study, the generalizability of our findings might be a threat to the validity of our findings. Actually, we would like to note that we do not seek to claim the generality of our findings in other companies. In contrast, the key message of our work is that we show our empirical findings on effort-aware JIT defect identification at Alibaba. Other companies can be aware of our empirical method and findings on effort-aware JIT defect identification in practice. They can borrow our method and findings for reference before they attempt to adopt JIT defect identification.

7 CONCLUSION

This paper explores the effectiveness of effort-aware JIT defect identification in an industrial setting. To that end, we conducted a case study on 14 Alibaba projects with 196,790 changes. In our case study, we investigated the effectiveness of five state-of-the-art effort-aware JIT defect identification approaches, including three supervised approaches (i.e., CBS+, OneWay, EALR) and two unsupervised approaches (i.e., LT and Code Churn). Then, we investigated the important change features for the best performing effort-aware approach. Additionally, we investigated the association between project-specific factors and the likelihood of a defective change. Finally, we developed a tool based on the best performing effort-aware approach (i.e., CBS+). By applying the developed tool at Alibaba, we conducted a user study to investigate the tool's effectiveness in the real-life industrial setting.

In summary, we make the following conclusions: (1) CBS+ outperforms all state-of-the-art supervised effort-aware JIT defect identification approaches on Alibaba projects. (2) The important change factors that impact the performance of effort-aware JIT defect identification approach differ between Alibaba projects and open source projects. (3) Project-specific factors have an association with the likelihood of a defective change in Alibaba projects. (4) The user study shows that our tool helps developers focus on a small number of the warned changes. The warned defective changes are correct in the 33% of the time.

Acknowledgment. This research is supported by the Australian Research Council's Discovery Early Career Researcher Award (DECRA)(DE200100021), Alibaba-Zhejiang University Joint Institute of Frontier Technologies, the Fundamental Research Funds for the Central Universities (No.2020CDJQY-A021) and the National Research Foundation, Singapore under its AI Singapore Programme (AISG Award No: AISG-100E-2018-004). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore and AI Singapore.

REFERENCES

[1] Hervé Abdi. 2007. Bonferroni and Šidák corrections for multiple comparisons. *Encyclopedia of measurement and statistics* 3 (2007), 103–107.

[2] Benjamin M Bolker, Mollie E Brooks, Connie J Clark, Shane W Geange, John R Poulsen, M Henry H Stevens, and Jada-Simone S White. 2009. Generalized linear mixed models: a practical guide for ecology and evolution. *Trends in ecology & evolution* 24, 3 (2009), 127–135.

[3] Andrew P Bradley. 1997. The use of the area under the ROC curve in the evaluation of machine learning algorithms. *Pattern recognition* 30, 7 (1997), 1145–1159.

[4] Norman Cliff. 2014. *Ordinal methods for behavioral data analysis*. Psychology Press.

[5] Jacek Czerwonka, Rajiv Das, Nachiappan Nagappan, Alex Tarvo, and Alex Teterev. 2011. Crane: Failure prediction, change analysis and test prioritization in practice–experiences from windows. In *Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on*. IEEE, 357–366.

[6] Yuanrui Fan, Xin Xia, David Lo, and Ahmed E Hassan. 2018. Chaff from the Wheat: Characterizing and Determining Valid Bug Reports. *IEEE Transactions on Software Engineering* (2018).

[7] Wei Fu and Tim Menzies. 2017. Revisiting unsupervised learning for defect prediction. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 72–83.

[8] Takafumi Fukushima, Yasutaka Kamei, Shane McIntosh, Kazuhiro Yamashita, and Naoyasu Ubayashi. 2014. An empirical study of just-in-time defect prediction using cross-project models. In *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM, 172–181.

[9] Frank E Harrell. 2001. *Regression modeling strategies: with applications to linear models, logistic regression, and survival analysis*. Springer.

[10] Safwat Hassan, Chakkrit Tantithamthavorn, Cor-Paul Bezemer, and Ahmed E Hassan. 2018. Studying the dialogue between users and developers of free apps in the google play store. *Empirical Software Engineering* 23, 3 (2018), 1275–1312.

[11] Hideaki Hata, Osamu Mizuno, and Tohru Kikuno. 2012. Bug prediction based on fine-grained module histories. In *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 200–210.

[12] Qiao Huang, Xin Xia, and David Lo. 2017. Supervised vs unsupervised models: A holistic look at effort-aware just-in-time defect prediction. In *Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on*. IEEE, 159–170.

[13] Qiao Huang, Xin Xia, and David Lo. 2018. Revisiting Supervised and Unsupervised Models for Effort-Aware Just-in-Time Defect Prediction. *Empirical Software Engineering* (2018), In press.

[14] Tian Jiang, Lin Tan, and Sunghun Kim. 2013. Personalized defect prediction. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 279–289.

[15] Paul CD Johnson. 2014. Extension of Nakagawa & Schielzeth's R2GLMM to random slopes models. *Methods in Ecology and Evolution* 5, 9 (2014), 944–946.

[16] Yasutaka Kamei, Takafumi Fukushima, Shane McIntosh, Kazuhiro Yamashita, Naoyasu Ubayashi, and Ahmed E Hassan. 2016. Studying just-in-time defect prediction using cross-project models. *Empirical Software Engineering* 21, 5 (2016), 2072–2106.

[17] Yasutaka Kamei, Emad Shihab, Bram Adams, Ahmed E Hassan, Audris Mockus, Aloka Sinha, and Naoyasu Ubayashi. 2013. A large-scale empirical study of just-in-time quality assurance. *Software Engineering, IEEE Transactions on* 39, 6 (2013), 757–773.

[18] Sunghun Kim, E James Whitehead Jr, and Yi Zhang. 2008. Classifying software changes: Clean or buggy? *IEEE Transactions on Software Engineering* 34, 2 (2008), 181–196.

[19] Stefan Lessmann, Bart Baesens, Christophe Mues, and Swantje Pietsch. 2008. Benchmarking classification models for software defect prediction: A proposed framework and novel findings. *IEEE Transactions on Software Engineering* 34, 4 (2008), 485–496.

[20] Chris Lewis, Zhongpeng Lin, Caitlin Sadowski, Xiaoyan Zhu, Rong Ou, and E James Whitehead Jr. 2013. Does bug prediction support human developers? findings from a google case study. In *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 372–381.

[21] Heng Li, Weiyi Shang, Ying Zou, and Ahmed E Hassan. 2017. Towards just-in-time suggestions for log changes. *Empirical Software Engineering* 22, 4 (2017), 1831–1865.

[22] Chao Liu, Dan Yang, Xin Xia, Meng Yan, and Xiaohong Zhang. 2019. A two-phase transfer learning model for cross-project defect prediction. *Information and Software Technology* 107 (2019), 125–136.

[23] Jinping Liu, Yuming Zhou, Yibiao Yang, Hongmin Lu, and Baowen Xu. 2017. Code churn: A neglected metric in effort-aware just-in-time defect prediction. In *Proceedings of the 11th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. IEEE Press, 11–19.

[24] Shane McIntosh and Yasutaka Kamei. 2017. Are fix-inducing changes a moving target? a longitudinal case study of just-in-time defect prediction. *IEEE Transactions on Software Engineering* (2017).

[25] Thilo Mende and Rainer Koschke. 2010. Effort-aware defect prediction models. In *Software Maintenance and Reengineering (CSMR), 2010 14th European Conference on*. IEEE, 107–116.

[26] Tim Menzies, Jeremy Greenwald, and Art Frank. 2007. Data mining static code attributes to learn defect predictors. *IEEE transactions on software engineering* 1 (2007), 2–13.

[27] André N Meyer, Thomas Fritz, Gail C Murphy, and Thomas Zimmermann. 2014. Software developers' perceptions of productivity. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 19–29.

[28] Osamu Mizuno and Tohru Kikuno. 2007. Training on errors experiment to detect fault-prone software modules by spam filter. 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, 405–414.

[29] Audris Mockus and David M Weiss. 2000. Predicting risk of software changes. *Bell Labs Technical Journal* 5, 2 (2000), 169–180.

[30] Shinichi Nakagawa and Holger Schielzeth. 2013. A general and simple method for obtaining R2 from generalized linear mixed-effects models. *Methods in Ecology and Evolution* 4, 2 (2013), 133–142.

[31] Mathieu Nayrolles and Abdelwahab Hamou-Lhadj. 2018. CLEVER: combining code metrics with clone detection for just-in-time fault prevention and resolution in large industrial projects. In *Proceedings of the 15th International Conference on Mining Software Repositories*. ACM, 153–164.

[32] Chris Parnin and Alessandro Orso. 2011. Are automated debugging techniques actually helping programmers?. In *Proceedings of the 2011 international symposium on software testing and analysis*. ACM, 199–209.

[33] Gopi Krishnan Rajbahadur, Shaowei Wang, Yasutaka Kamei, and Ahmed E Hassan. 2017. The impact of using regression models to build defect classifiers. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 135–145.

[34] Alastair J Scott and M Knott. 1974. A cluster analysis method for grouping means in the analysis of variance. *Biometrics* (1974), 507–512.

[35] Martin Shepperd, David Bowes, and Tracy Hall. 2014. Researcher bias: The use of machine learning in software defect prediction. *IEEE Transactions on Software Engineering* 40, 6 (2014), 603–616.

[36] Emad Shihab, Ahmed E Hassan, Bram Adams, and Zhen Ming Jiang. 2012. An industrial study on the risk of software changes. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, 62.

[37] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. 2005. When do changes induce fixes?. In *ACM sigsoft software engineering notes*, Vol. 30. ACM, 1–5.

[38] Tom AB Snijders. 2005. Fixed and random effects. *Encyclopedia of statistics in behavioral science* (2005).

[39] Ming Tan, Lin Tan, Sashank Dara, and Caleb Mayeux. 2015. Online defect prediction for imbalanced data. In *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, Vol. 2. IEEE, 99–108.

[40] Chakkrit Tantithamthavorn and Ahmed E Hassan. 2018. An experience report on defect modelling in practice: Pitfalls and challenges. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*. ACM, 286–295.

[41] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E Hassan, and Kenichi Matsumoto. 2016. Automated parameter optimization of classification techniques for defect prediction models. In *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*. IEEE, 321–332.

[42] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E Hassan, and Kenichi Matsumoto. 2017. An empirical comparison of model validation techniques for defect prediction models. *IEEE Transactions on Software Engineering* 43, 1 (2017), 1–18.

[43] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E Hassan, and Kenichi Matsumoto. 2018. The impact of automated parameter optimization on defect prediction models. *IEEE Transactions on Software Engineering* (2018).

[44] AB Tom, Tom AB Snijders Roel J Bosker, and Roel J Bosker. 1999. *Multilevel analysis: an introduction to basic and advanced multilevel modeling*. Sage.

[45] Zhiyuan Wan, Xin Xia, Ahmed E Hassan, David Lo, Jianwei Yin, and Xiaohu Yang. 2018. Perceptions, Expectations, and Challenges in Defect Prediction. *IEEE Transactions on Software Engineering* (2018).

[46] Wikipedia. [n. d.]. https://en.wikipedia.org/wiki/Alibaba_Group. ([n. d.]).

[47] Frank Wilcoxon. 1992. Individual comparisons by ranking methods. *Breakthroughs in Statistics* (1992), 196–202.

[48] Xin Xia, Lingfeng Bao, David Lo, Pavneet Singh Kochhar, Ahmed E Hassan, and Zhenchang Xing. 2017. What do developers search for on the web? *Empirical Software Engineering* 22, 6 (2017), 3149–3185.

[49] Xin Xia, LO David, Sinno Jialin Pan, Nachiappan Nagappan, and Xinyu Wang. 2016. Hydra: Massively compositional model for cross-project defect prediction. *IEEE Transactions on software Engineering* 42, 10 (2016), 977.

[50] Meng Yan, Yicheng Fang, David Lo, Xin Xia, and Xiaohong Zhang. 2017. File-level defect prediction: Unsupervised vs. supervised models. In *2017 ACM/IEEE*

International Symposium on Empirical Software Engineering and Measurement (ESEM). IEEE, 344–353.

[51] Meng Yan, Xin Xia, David Lo, Ahmed E Hassan, and Shanping Li. 2019. Characterizing and identifying reverted commits. *Empirical Software Engineering* 24, 4 (2019), 2171–2208.

[52] Meng Yan, Xin Xia, Emad Shihab, David Lo, Jianwei Yin, and Xiaohu Yang. 2018. Automating change-level self-admitted technical debt determination. *IEEE Transactions on Software Engineering* 45, 12 (2018), 1211–1229.

[53] Xinli Yang, David Lo, Xin Xia, and Jianling Sun. 2017. TLEL: A two-layer ensemble learning approach for just-in-time defect prediction. *Information and Software Technology* 87 (2017), 206–220.

[54] Xinli Yang, David Lo, Xin Xia, Yun Zhang, and Jianling Sun. 2015. Deep learning for just-in-time defect prediction. In *Software Quality, Reliability and Security (QRS), 2015 IEEE International Conference on*. IEEE, 17–26.

[55] Yibiao Yang, Yuming Zhou, Jinping Liu, Yangyang Zhao, Hongmin Lu, Lei Xu, Baowen Xu, and Hareton Leung. 2016. Effort-aware just-in-time defect prediction: simple unsupervised models could be better than supervised models. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 157–168.

[56] Zuoning Yin, Ding Yuan, Yuanyuan Zhou, Shankar Pasupathy, and Lakshmi Bairavasundaram. 2011. How do fixes become bugs?. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 26–36.

[57] Feng Zhang, Audris Mockus, Iman Keivanloo, and Ying Zou. 2014. Towards building a universal defect prediction model. In *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM, 182–191.

[58] Feng Zhang, Audris Mockus, Ying Zou, Foutse Khomh, and Ahmed E Hassan. 2013. How does context affect the distribution of software maintainability metrics?. In *2013 IEEE International Conference on Software Maintenance*. IEEE, 350–359.

[59] Yuming Zhou, Yibiao Yang, Hongmin Lu, Lin Chen, Yanhui Li, Yangyang Zhao, Junyan Qian, and Baowen Xu. 2018. How Far We Have Progressed in the Journey? An Examination of Cross-Project Defect Prediction. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 27, 1 (2018), 1.

[60] Thomas Zimmermann, Nachiappan Nagappan, Harald Gall, Emanuel Giger, and Brendan Murphy. 2009. Cross-project defect prediction: a large scale experiment on data vs. domain vs. process. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. ACM, 91–100.