A Procedure to Continuously Evaluate Predictive Performance of Just-In-Time Software Defect Prediction Models During Software Development

Liyan Song, Leandro L. Minku, *Senior Member, IEEE*

Abstract—Just-In-Time Software Defect Prediction (JIT-SDP) uses machine learning to predict whether software changes are defect-inducing or clean. When adopting JIT-SDP, changes in the underlying defect generating process may significantly affect the predictive performance of JIT-SDP models over time. Therefore, being able to continuously track the predictive performance of JIT-SDP models during the software development process is of utmost importance for software companies to decide whether or not to trust the predictions provided by such models over time. However, there has been little discussion on how to continuously evaluate predictive performance in practice, and such evaluation is not straightforward. In particular, labeled software changes that can be used for evaluation arrive over time with a delay, which in part corresponds to the time we have to wait to label software changes as 'clean' (waiting time). A clean label assigned based on a given waiting time may not correspond to the true label of the software changes. This can potentially hinder the validity of any continuous predictive performance evaluation procedure for JIT-SDP models. This paper provides the first discussion of how to continuously evaluate predictive performance of JIT-SDP models over time during the software development process, and the first investigation of whether and to what extent waiting time affects the validity of such continuous performance evaluation procedure in JIT-SDP. Based on 13 GitHub projects, we found that waiting time had a significant impact on the validity. Though typically small, the differences in estimated predicted performance were sometimes large, and thus inappropriate choices of waiting time can lead to misleading estimations of predictive performance over time. Such impact did not normally change the ranking between JIT-SDP models, and thus conclusions in terms of which JIT-SDP model performs better are likely reliable independent of the choice of waiting time, especially when considered across projects.

Index Terms—Just-in-time software defect prediction, performance evaluation procedure, concept drift, data stream learning, online learning, verification latency, and label noise.

1 Introduction

Just-In-Time Software Defect Prediction (JIT-SDP) is a type of SDP that makes predictions at the software change level, aiming to label software changes as *defect-inducing* or *clean* upon commit time (i.e., just-in-time) [1]. It has been attracting increased interest from both industry and academia [2], [3], [4], [5], [6]. In practice, JIT-SDP operates in an *online learning* scenario, where additional software changes are produced and labeled over time, becoming available for training and evaluating JIT-SDP models as part of a data stream. Ignoring the chronology nature in research studies means that JIT-SDP models are trained with future data which would not have been available in practice, leading to over-optimistic estimations of predictive performance [4].

Studies that take such chronological nature into account found that the predictive performance of JIT-SDP models suffers significant variations over time [5], [6], [7]. Such variations are likely a result of variations in the defect-generating process, which can cause JIT-SDP models to become less suitable over time. This phenomenon is referred to as *concept drift* [8]. Given the variability of JIT-SDP predictive performance over time, it is of utmost importance for software companies to be able to continuously track the predictive performance of JIT-SDP models over time during the software development, so that practitioners are aware of when a JIT-SDP model becomes unreliable. However, there has been little discussion on how to continuously evaluate predictive performance in practical scenarios, and such evaluation is not straightforward.

In particular, any procedure for continuously evaluating predictive performance would rely on labeled data being received over time. However, the labels attributed to software changes over time may not be immediately reliable, because it takes time for the true label of a software change to be revealed in real scenarios [6], [8]. So, examples may need to be created based on *observed* rather than *true* labels.

A software change is labeled to produce a defect-inducing example when a defect is found to be induced by it. A software change is labeled as clean when no defect has

- *We would like to thank Dr. Zongliang Hu, a statistician in the college of mathematics and statistics at Shenzhen University, for valuable advice on the statistical analyses. This work was supported by EPSRC (Grant No. EP/R006660/2), National Natural Science Foundation of China (Grant No. 62002148), Guangdong Provincial Key Laboratory (Grant No. 2020B121201001), the Program for Guangdong Introducing Innovative and Enterpreneurial Teams (Grant No. 2017ZT07X386), Shenzhen Science and Technology Program (Grant No. KQTD2016112514355531), and Research Institute of Trustworthy Autonomous Systems, Southern University of Science and Technology, China.*
- *L. Song is with Research Institute of Trustworthy Autonomous Systems, Southern University of Science and Technology, China and Guangdong Provincial Key Laboratory of Brain-inspired Intelligent Computation, Department of Computer Science and Engineering, Southern University of Science and Technology, China. E-mail: songly@sustech.edu.cn.*
- *L.L. Minku is with the School of Computer Science, University of Birmingham, UK. Email: L.L.Minku@bham.ac.uk (Corresponding Author).*

yet been found to be induced by it *and* enough time has passed for one to be confident that this software change is clean. Such length of time is referred to as *waiting time* [6]. As the time it takes to find defects associated to a software change is a priori unknown and may vary from software change to software change, the need for adopting a waiting time is inherently associated to label noise. In particular, the clean observed label resulting from the waiting time may or may not be the same as the true label of the software change. Whenever it is not the same, a noisy example is created. Such noise caused by the waiting time may affect not only the training of JIT-SDP models, but also the validity of any procedure to continuously evaluate the predictive performance of JIT-SDP models over time. However, the impact of such noise has not yet been investigated.

Waiting time is a controllable parameter and different values may induce different observed labels for the same software change. A large waiting time may alleviate label noise as we wait for longer time for defects to be found. However, a large waiting time increases the time it takes for labeled examples to be produced. So, by the time an example is produced, it may be obsolete and not represent the current defect-generating process well anymore, due to concept drift [5]. Therefore, the chosen waiting time has ramifications on the evaluation procedures of JIT-SDP models. Any evaluation based on high levels of label noise or on obsolete / unrepresentative examples as a result of an inadequate waiting time can potentially be invalid.

However, such ramifications have not yet been analyzed in the literature. In particular, whether and the extent to which waiting time has an impact on the procedure to continuously evaluate JIT-SDP models is unknown. An evaluation procedure that ignores the fact that observed labels produced based on a waiting time are being used rather than the true labels, or that adopts inappropriate waiting times could possibly lead to invalid conclusions. Understanding such potential impact is thus very important. Otherwise, practitioners would not know whether and to what extent they can trust the performance estimated by continuous performance evaluation procedures for JIT-SDP models being adopted during software development.

Therefore, this paper formulates the procedure for continuously evaluating predictive performance during software development in JIT-SDP. It then provides the first investigation of whether and the extent to which waiting time (and its resulting label noise and delay) has an impact on the validity of continuous performance evaluation in JIT-SDP. We answer four Research Questions (RQs) as follows.

[RQ1] How large is the amount of label noise caused by different waiting times in JIT-SDP? Intuitively, larger waiting time would induce smaller amount of label noise, but one cannot wait forever as larger waiting time would also cause examples to become obsolete. So far, no one knows how much label noise different waiting times would cause. We will quantitatively analyze the impact of different waiting times on label noise in JIT-SDP.

[RQ2] To what extent different levels of noise resulting from different choices of waiting time impact the validity of the continuous performance evaluation procedure? So far, no one knows the extent of the impact of such label noise on the validity of the procedure for continuously evaluating predictive performance and how much the estimated predictive performance varies when different waiting times are used. This knowledge is important for practitioners to decide whether or not to adopt a JIT-SDP model, as they will use an estimated performance calculated based on observed labels to make such decisions. This investigation will enable us to check how reliable the estimated performance tracked continuously over time in JIT-SDP is, given the label noise caused by different waiting times.

[RQ3] To what extent is the validity of the continuous performance evaluation procedure impacted by waiting time? Part of this can be answered by combining the conclusions of RQ1 and RQ2: if waiting time has significant impact on label noise (RQ1) and label noise has significant impact on the validity of the continuous performance evaluation procedure (RQ2), waiting time may have significant impact on the validity of the continuous performance evaluation procedure through the label noise it generates. However, another perspective still needs to be explored: whether waiting time has additional impact on the validity of continuous performance evaluation that cannot be captured by label noise. Such additional impact could happen as a result of the obsolescence of the examples created based on a given waiting time. This is our main research question enabling us to check how reliable the continuous estimation of predictive performance retrieved by a continuous performance evaluation procedure is given the waiting time. Such knowledge is important because the estimated predictive performance can be used to decide whether JIT-SDP performs well enough to be adopted in practice.

[RQ4] If waiting time has significant impact on the validity of the continuous performance evaluation procedure, can inappropriate waiting times change the conclusions in terms of the ranking of different JIT-SDP models? Even if waiting time has significant impact on the validity of the continuous performance evaluation procedure, if all JIT-SDP models are influenced in exactly the same way, this will not change the relative rankings of JIT-SDP models. Therefore, the conclusions in terms of which models perform better than others would still be reliable. This investigation is important because practitioners may wish to decide which JIT-SDP models to adopt over time in their company, based on which of them is better ranked in terms of its predictive performance over time on their projects.

To answer these RQs, we conduct experimental studies based on 13 GitHub software projects and statistically analyze them. We find that waiting time has significant impact on the validity of the continuous performance evaluation procedure for JIT-SDP. Though typically in small magnitude, the differences in estimated performance caused by different waiting times were sometimes large. Therefore, inappropriate choices of waiting time can lead to misleading estimations of predictive performance over time. Nevertheless, the results also show that such impact does not normally change the estimated ranking of JIT-SDP models. Therefore, the conclusions in terms of which JIT-SDP models perform better are likely reliable independent of the choice of waiting time.

The main contributions of this paper are listed below:

- The first study on how to continuously evaluate predictive performance of JIT-SDP during software develop-

ment.
- A mathematical formulation of the continuous prediction, training and evaluation procedures for JIT-SDP in the online learning scenario, enabling a non-ambiguous understanding of such procedures.
- The first investigation of whether and to what extent the conclusions of the JIT-SDP continuous performance evaluation procedure is potentially invalid when adopting different waiting time values.
- A detailed analysis on why waiting time impacts the validity of the continuous performance evaluation procedure.
- Recommendations on how to choose appropriate waiting times to prevent an invalid performance evaluation procedure.

It is worth noting that this paper investigates the impact of waiting time on the validity of the performance evaluation procedure. This is different from investigating the impact of waiting time on the predictive performance of JIT-SDP models. The latter is about training JIT-SDP models with examples produced based on different waiting times, and checking which waiting time leads to the best performing models. It measures the impact of waiting time on the capability of JIT-SDP models to correctly predict the labels of new software changes. The former is about how correct the *procedure to estimate such capability* is. This involves analyzing the difference between (1) the true predictive performance computed with ground-truth evaluation examples, which are unavailable for continuous performance evaluation in practice, and (2) the estimated predictive performance computed based on examples associated to a waiting time, which can be collected through a continuous predictive performance evaluation procedure in practice.

The remainder of this paper is organized as follows. Section 2 presents a motivating scenario to illustrate the need for a continuous performance evaluation procedure that can be adopted in practice. Section 3 discusses background and related work. Section 4 explains our notation system and formulates the continuous prediction, training and evaluation procedures of JIT-SDP in the online learning scenario. Section 5 formalizes our four research questions. Real-world software projects used in the paper are described in Section 6, followed by the experimental design in Section 7. Experimental results are discussed in Section 8, answering our RQs. Section 9 discusses threats to validity. The paper is concluded in Section 10.

2 Motivating Scenario

Consider a software development manager who read that researchers proposed a new JIT-SDP method. This method achieved competitive results against other methods on several open source and proprietary projects. This manager is thus keen to adopt this method in their company. However, this manager also read that the predictive performance of such methods may vary over time. They do not wish to rely on this JIT-SDP method if/when its predictive performance becomes poor. Therefore, this software manager will only agree to adopt such JIT-SDP method in their company if they can continuously monitor the predictive performance of this method during software development, using a reliable performance evaluation method.

Alas, even the procedures used to evaluate predictive performance in online JIT-SDP studies [6], [7] are unsuitable to continuously evaluate predictive performance in practice, during software development. This is because these studies assume that (1) the evaluation of the JIT-SDP model is based on software changes whose label is immediately known at commit time, and (2) this label is the true label. In other words, such evaluation procedures assume that there is no delay or label noise resulting from verification latency for evaluation purposes. And, indeed, for the evaluation purposes of those papers, all data was already available beforehand. However, if a software manager wishes to monitor the predictive performance of JIT-SDP models over time during software development, the labels of the software changes will arrive with a delay and may be noisy due to verification latency, as explained in Section 1.

Therefore, the software manager wishes to know (1) what procedure could be used to continuously monitor the predictive performance of JIT-SDP models over time during software development, (2) how reliable this continuous evaluation procedure itself is, and (3) how to use this procedure. This paper addresses these issues. Point (1) is addressed by formulating a continuous evaluation procedure that can be used during software development. Point (2) is addressed through RQ1-RQ4 presented in Section 1. Point (3) is addressed by recommendations on how to choose appropriate waiting times to use with such procedure.

3 Background and Related Work

3.1 JIT-SDP

In 2008, Kim et al. published the first study on JIT-SDP, where they described a set of input features for JIT-SDP models [9]. Several other studies investigated potentially beneficial input features for JIT-SDP, such as the day of the week [10] or the time of the day [11] a software change was produced, and input features to enable the identification of software changes that require a lot of effort to fix [12]. Shihab et al. [3] showed that the number of lines of code added, the ratio of bug fixing to total changes that touched a file, the number of bug reports linked to a commit, and the developer experience are good indicators of defect-inducing software changes. Kamei et al. conducted a large-scale empirical study [1] investigating a variety of factors extracted from commits and bug reports as input features for JIT-SDP models. They considered 14 features grouped into five dimensions of diffusion, size, purpose, history and experience, and showed such features to be good indicators of defect-inducing software changes for yielding high predictive performance on both open source and commercial projects. Many subsequent studies have adopted these features [13], [12], [5], [6].

From a machine learning perspective, most existing work considers JIT-SDP as a binary classification problem, where a model needs to be built based on examples of software changes that have been labeled as defect-inducing or clean. This model can be used for predicting whether or not new software changes are defect-inducing. Existing work has investigated the use of support vector machines

[9], random forests [13], logistic regression [1], [5] and deep learning [14], [15]. Among them, tree-based and logistic regression-based models are among the most popular used learning models that have shown potential in providing good performance for JIT-SDP. Techniques such as under-sampling [16] and SMOTE [17] have also been adopted to deal with the fact that JIT-SDP suffers from class imbalance [1], [4], [13], where the defect-inducing class is typically a minority in comparison to the clean class.

All studies above considered JIT-SDP as an offline learning problem, where all examples for building JIT-SDP models are available beforehand and no further adjustment or evaluation of the constructed model with new examples is performed over time.

3.2 Chronology in JIT-SDP

Most existing work on JIT-SDP overlooks the chronology of the software changes, for which software changes arrive sequentially in order over time. Based on one proprietary project from Cisco and six open source projects, Tan et al. [4] showed that overlooking chronology leads to JIT-SDP with deceptively higher predictive performance than could be achieved in practice, when chronology must be respected. Possibly, such over-optimistic predictive performance happens as the characteristics of the software changes vary over time, which can cause JIT-SDP models trained on old data to become obsolete (with deteriorated predictive performance) [5]. If a JIT-SDP model is trained on data that would only have been available in the future, it may deceptively provide more accurate predictions on future software changes than if it had only been trained on past data.

The variations in the characteristics of software changes discussed above correspond to a phenomenon called *concept drift* [8], which in the context of JIT-SDP are changes in the defect generating process. Examples of non-stationary characteristics of software development projects that may result in concept drift include changes in the stage of the software development process, changes in the type of software feature currently being implemented, changes in the development team, changes in management team, etc. Through a longitudinal case study of the QT and OPENSTACK systems on 37,524 software changes, McIntosh and Kamei [5] showed that the characteristics of defect-inducing software changes fluctuate over time, and such fluctuations can negatively impact predictive performance of JIT-SDP models trained on old examples. Cabral et al. [6] showed that the proportion of examples of the defect-inducing and the clean classes also fluctuates over time in JIT-SDP. Based on an analysis of ten GitHub projects, they showed that models specifically designed for coping with such fluctuations can improve predictive performance.

The challenges posed by concept drift in JIT-SDP are exacerbated by verification latency. *Verification latency* is the length of time between the production of a software change and the labeling of this software change [8]. As the *bug-fix* software change or bug issue report required to identify a *bug-introducing* software change [10] comes after this bug-introducing change, it takes time for the label of a software change to be revealed. The labels of defect-inducing software changes can usually be determined only months or even years after their commit time [6]. As a result, one also needs to wait for enough time to be confident that a given software change can be labeled as clean. Therefore, in practice, the label of a software change only becomes available long after this change has been committed. This can delay adaptation of JIT-SDP models to concept drift.

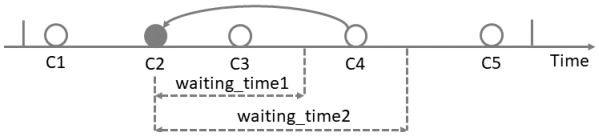


Fig. 1. Label noise caused by an inadequate waiting time in JIT-SDP. Software changes are denoted by 'C' followed by a number indicating their order of commit time. Gray (white) circles denote defect-inducing (clean) software changes. An arrow links the defect-inducing software change C2 and the one (C4) that fixes the defect. Using waiting_time1 will cause a wrong clean label assigned to C2. Using waiting_time2 will lead to the correct defect-inducing label being assigned.

In file-based SDP, Chen et al. [18] defined *dormant bugs* as defects introduced in a version of the software system but found only several versions later. Based on 20 open-source Apache foundation software systems, they found that typically 33% of the defects introduced in a version were reported in future versions as dormant bugs. They concluded that performance evaluation that ignores dormant bugs could be misleading. Even though this study was in the context of module-based SDP, it has ramifications on JIT-SDP, as it means that defects may take a lot of time to be discovered, resulting in verification latency.

Cabral et al. [6] further showed in a JIT-SDP study involving 10 GitHub projects that the time it takes for defects induced by software changes to be found varied from 1 to 11.5 years after commit time, with a median of 90 days. To take such verification latency into account for training purposes, they explicitly considered a *waiting time* strategy, where examples can only be used for training as clean examples after a period of time (waiting time) has passed from their commit time, or as defect-inducing examples when a defect is found to be induced by this software change, whichever is shorter.

The studies above highlight the importance of studying JIT-SDP in online learning scenarios, i.e., scenarios that fully respect the chronological order of arrival of software changes and their labels. They show that JIT-SDP models can suffer drops in predictive performance over time and that recovering from such drops may be challenging due to verification latency. However, there has been no investigation of the impact of verification latency on the validity of the performance evaluation procedure, and no discussion on how to continuously evaluate predictive performance in practical scenarios in view of the waiting time.

3.3 Label Noise Caused By Waiting Time in JIT-SDP

Waiting time is a parameter that is inherently necessary in JIT-SDP for producing clean / defect-inducing labels [6] and that ought to be carefully chosen. As briefly mentioned in Section 1, different waiting times may lead to different observed labels for the same software change, potentially leading to label noise. Figure 1 illustrates the scenario where label noise occurs due to an inadequate waiting time. As

shown in this figure, software change C2 is defect-inducing. If waiting_time1 had been adopted to decide C2's observed label, as C4 is nonexistent, C2 will be assigned a 'clean' label erroneously, meaning that a training example with label noise is created. In contrast, if waiting_time2 had been used, the correct label (defect-inducing) would have been assigned to C2. Indeed, we can get the correct label of C2 at any time after C4. However, large waiting times could mean that the labeled examples are already obsolete by the time they are produced, due to concept drift.

McIntosh et al. [5] suggested to train JIT-SDP models on software changes that are only up to 90 days old to reduce the drops in predictive performance potentially caused by concept drift. This implies that a waiting time much lower than 90 days would be necessary to avoid hindering predictive performance of JIT-SDP models or to provide adequate estimates of predictive performance in view of concept drift. However, the median time to find defects was around 90 days in Cabral et al.'s study [6], suggesting that the waiting time should typically be set to 90 days or more in view of the label noise that can result from the defect discovery delay. Thus, choosing an appropriate waiting time to obtain a good trade-off between the obsolescence of the resulting labeled examples and label noise may not be easy.

However, these studies did not investigate the amount of label noise resulting from different choices of waiting time and did not investigate the impact of such label noise on the validity of the performance evaluation procedure. Moreover, they focused on how to achieve good predictive performance in JIT-SDP. No study investigated what aspects should be the driving force for choosing the waiting time and what waiting times are typically most adequate for the purpose of continuous performance evaluation. Our work is the first to provide such analyses. It is worth noting that any continuous performance evaluation procedure would require a waiting time to be used, as the generation of labeled examples depends on it. Therefore, any continuous performance evaluation procedure risks being affected by the noise caused by the waiting time choice, leading to being potentially invalid estimates of predictive performance.

3.4 Label Noise Not Caused By Waiting Time

Existing studies in the module-based SDP literature usually focus on the mislabeling that occurs when extracting and collecting data examples from software repositories [19], [20], [21], [22]. These studies found that issue reports can often be mislabeled (e.g., reports that describe defects are mislabeled as "enhancements"), influencing the issue tracking system and version control system records based on which source code modules are labeled as defective or clean. This can potentially result in labeling defective modules as clean erroneously or vice-versa. Some studies reported that such mislabeling can lead to a negative impact on SDP's predictive performance [23], while others concluded that this rarely leads to a severe impediment to SDP [24].

In the JIT-SDP literature, the popular SZZ algorithm [10] used to collect input features and labels for software changes and its variants [25], [26], [27] may also result in label noise, as they may erroneously label a clean software change as defect-inducing or vise versa. Fan et al. [28] investigated the impact of mislabeled software changes based on four popular SZZ algorithms on predictive performance of JIT-SDP. Considering the model that is trained on examples labeled by RA-SZZ (Refactoring Aware SZZ, the most recent SZZ version) [27] as the baseline, they concluded that the SZZ-related label noise caused by AG-SZZ (Annotation Graph SZZ) [25] can cause a significant performance reduction, whereas label noise caused by B-SZZ (the original and most popular SZZ algorithm) [10] and MA-SZZ (Meta-change Aware SZZ) [26] were unlikely to cause a considerable performance reduction. Label noise resulting from waiting time has not been investigated in these studies.

Our study is concerned with label noise resulting from inadequate waiting time for JIT-SDP, which has not been discussed in existing literature. Therefore, hereafter, whenever we refer to label noise, we mean the one associated to waiting time, unless otherwise specified.

4 Online JIT-SDP Problem Formulation

As explained in Sections 1 to 3, any procedure to evaluate predictive performance of JIT-SDP models needs to be based on observed labels, potentially suffering from validity issues. This paper aims to (1) formulate the procedure for continuously evaluating predictive performance of JIT-SDP models during software development and to (2) analyze the extent to which the results of JIT-SDP continuous performance evaluation can be affected by such validity issues. To achieve this objective, this section will explain our notation system and formulate the continuous labeling, prediction, training, and evaluation processes of JIT-SDP in the online learning scenario. We will formulate the true performance based on true labels of software changes and the estimated performance based on observed labels.

4.1 Three Types of Time Step and Data Stream

Time step is a sequential natural number, representing the order of the usage (i.e., training, prediction and evaluation) of software changes in the project development process. The actual time interval between time steps t and $t+1$ may be different from that between $t+1$ and $t+2$.

In online JIT-SDP, there are three types of time step: (1) the *training time step* when a software change is labeled to train a JIT-SDP model, (2) the *commit time step* when a new software change is produced and needs to be predicted as defect-inducing or clean, and (3) the *evaluation time step* when the model performance is evaluated. In line with them, there are three types of data streams: (1) the *training data stream* composed of labeled examples ordered according to their training time step, (2) the *commit data stream* composed of unlabeled software changes ordered according to their commit time step, and (3) the *evaluation data stream* composed of labeled examples ordered according to their evaluation time step.

Note that any software change appearing at a given moment in the commit data stream can only appear in the training and the evaluation data streams at a later moment, due to verification latency. Any labeled software change that appears both in the evaluation and training data streams must not be used for training before it is used for

evaluation, even if it appears at both streams at the same moment. Otherwise, this would invalidate the performance evaluation procedure. Though software changes at commit and evaluation time steps have to be predicted by JIT-SDP models, the purpose of such prediction is different. At a commit time step, a software change is unlabeled and needs to be predicted to guide decision-making in the real-world software development process. At an evaluation time step, a software change is predicted to determine whether the JIT-SDP model is giving a correct or incorrect prediction, so that the performance of JIT-SDP models can be evaluated.

In our mathematical formulation, we will frequently need to convert between a time step and an actual Unix timestamp in a data stream of software changes. We will use an uppercase letter such as T and T_s to indicate a Unix timestamp, and a lowercase letter such as t and t_s to denote its corresponding time step in a data stream. Lower and uppercase will be used interchangeably whenever we need to emphasize the time step or timestamp in our formulation.

4.2 Labeling Procedure

As true labels of software changes are usually unknown in practice due to verification latency, observed labels are actually used for training and evaluating JIT-SDP models.

We use $y^*_{u,t}$ to denote the label observed at time step t for the software change at commit time step u, where $U < T$ and "$*$" indicates that this is an observed label. We can use $y^*_{u,T}$ to denote the same label. We will denote an observed label in the form of $y^*_{u,t}$ or $y^*_{u,T}$ when we need to emphasize the time step or timestamp of the labeling time. We use the value 0 to indicate clean and 1 to indicate defect-inducing.

Given a software change X_u at commit time step u, its observed label is obtained based on the cases below [6]:

1) If the software change is not found to be defect-inducing until Unix timestamp $T = U + W$, where W is the waiting time parameter, $y^*_{u,T}$ is labeled as clean, producing the example $(X_u, 0)$ at time step t. This case is illustrated by software change X_2 in Figure 2 – no defect was found until the end of the waiting time.
2) If it is found to be defect-inducing at a Unix timestamp $T' < T$, then $y^*_{u,T'}$ is labeled defect-inducing, producing the example $(X_u, 1)$ at time step t'. This case is illustrated by software change X_5 in Figure 2 – a defect was found to be associated to it based on the fix by software change X_7, before the end of the waiting time.
3) It may also happen that the software change is first labeled as clean at timestamp T, but later is found to have induced a defect at timestamp $T'' > T$. This happens when the waiting time is not enough to correctly label a training example, resulting in this example being mistakenly labeled as clean and found to be defect-inducing afterwards. In such case, a training example $(X_u, 0)$ is first produced based on case 1) at time step t, and then a new training example $(X_u, 1)$ is produced at t''. This case is illustrated by software change X_1 in Figure 2 – this software change is not associated to any defect by the end of the waiting time, but a defect is found to be induced by it through the fix change X_4.

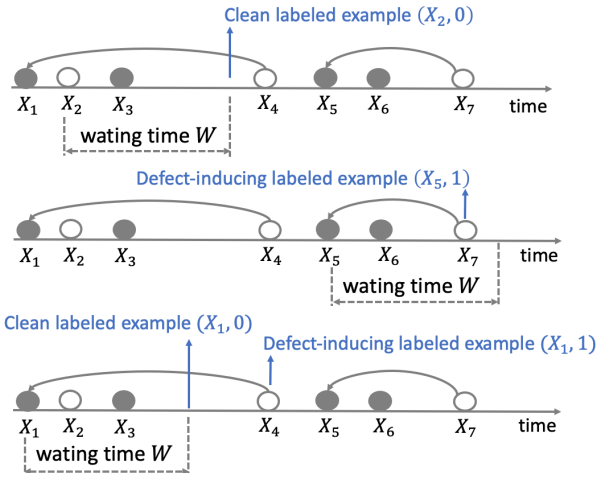


Fig. 2. Illustration of the three different example labeling cases. Software changes (input features) are denoted by 'X' followed by a number indicating their order of commit time. Solid (hollow) circles denote software changes that are truly defect-inducing (clean). Note that such knowledge does not exist at commit time in practice. A gray arrow links the defect-inducing change and the one that fixes the defect. The moments when software changes X_2, X_5 and X_1 are labeled are indicated by blue arrows. The moments when other software changes are labeled is not shown, so that the figure does not get too crowded.

4.3 Prediction, Evaluation and Training Procedures

Whenever a software change is produced, it is inserted into the commit data stream and the most up-to-date JIT-SDP model available is used to predict it. Whenever a labeled example is created based on the labeling procedure presented in Section 4.2, it is immediately inserted into the evaluation data stream and used to estimate the predictive performance of the JIT-SDP model. *After* being used for evaluation purposes, the labeled example will be inserted into the training data stream and used for training purposes, i.e., for updating the JIT-SDP model. If desired, online machine learning algorithms [6] that are able to learn example-by-example without requiring access to past examples could be used for training purposes.

These prediction, evaluation and training procedures ensure that chronology is always respected. No data from the future is ever used to train a model for the present, and no example is ever used for training before being used for evaluation in any of our experiments and analyses. A detailed mathematical formulation of the training and prediction procedures can be found in Sections II and III of the supplementary material. The mathematical formulation of the evaluation process is presented in Section 4.4.

We will adopt online JIT-SDP models in this study, i.e., JIT-SDP models able to learn over time. However, it is worth noting that the continuous predictive performance evaluation procedure discussed in this paper could also be applied to offline JIT-SDP models that do not learn over time. The predictive performance of offline models may also vary over time depending on the incoming software changes being predicted. Therefore, not only online JIT-SDP models, but also offline JIT-SDP models would benefit from a continuous performance evaluation procedure.

4.4 Formulating the Evaluation Procedure

The predictive performance computed based on observed labels through an evaluation procedure is referred to as the *estimated performance*. This is in contrast to the *true performance* that is calculated based on true labels without label noise. This section will formulate the procedure to compute the true and the estimated performance.

As discussed in Section 1, the predictive performance of JIT-SDP models should ideally be continuously monitored in practice, so that its fluctuations can be traced to judge how (un)reliably a model is becoming over time. For that, a *continuous performance* evaluation procedure is necessary. In the data stream learning literature, prequential calculation of the predictive performance based on a forgetting factor has been recommended to continuously track predictive performance over time [29], [30], [31], [32]. Prequential here means that, whenever a new labeled example is received, it is first used to test the model and incrementally update the value of its predictive performance. Only after that it can be used for training this model, if desired. The *forgetting factor* is a predefined value used to emphasize the predictive status of most recent evaluation steps and weaken the effect of evaluation examples from older time steps. This enables tracing fluctuations in predictive performance over time.

This section explains how the prequential performance based on a forgetting factor can be used to formulate the true and the estimated predictive performance using a continuous performance evaluation procedure in JIT-SDP.

4.4.1 True Performance of JIT-SDP at Unix Timestamp T

Given a timestamp T, one would ideally evaluate the predictive performance of JIT-SDP based on the *predicted* and *true* (not observed) labels of evaluation examples. In the ideal scenario where the true labels are available, the evaluation data stream available at T can be formulated as

$$\mathbb{E} = \{(X_u, y_u)\}_{u=1}^{t},$$

where the mathematical bold $\mathbb{E}$ is used to represent that this data stream is used for evaluation, X_u denotes the software change being used for evaluation at evaluation time step u, y_u is the true label of X_u, and t is the evaluation time step corresponding to T. As t represents the last evaluation time step, we can also use it to denote the number of time steps in the evaluation process. In the illustrative example of Figure 3, $\mathbb{E}$ would be composed of all software changes from X_1 to X_t with their true labels from y_1 to y_t.

The *true continuous performance* of a JIT-SDP model at timestamp $U \leq T$ is calculated based on software changes that are predicted up to U based on their true labels recursively as

$$E_{cn}(u) = \theta \cdot E_{cn}(u-1) + (1-\theta) \cdot ||\widehat{y}_u - y_u||_G, \qquad (1)$$

where the subscript cn is used to indicate a performance computed through a continuous evaluation procedure, u denotes the time step of U in evaluation data stream $\mathbb{E}$, $\widehat{y}_u$ (y_u) represents the predicted (true) label of X_u, $||\widehat{y}_u - y_u||_G$ measures the correctness of the prediction given by the JIT-SDP model on the example received at evaluation time step u, the subscript G can denote any performance metric such as accuracy, G-mean [33] or F1-score, and $\theta \in (0, 1)$ is the *forgetting factor* that controls how much emphasis should be placed on past evaluation examples compared to the new one. Larger/smaller values for θ place more emphasis on the past/present. The predicted and true labels used for computing $E_{cn}(u)$ in the scenario where $u = t$ are illustrated in the green box of Figure 3.

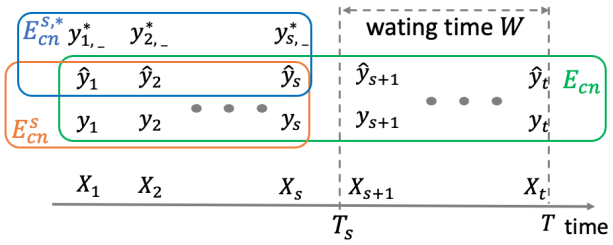


Fig. 3. Illustration of evaluation data streams that are used to compute the true performance E_{cn}, the estimated performance based on surrogate time steps E_{cn}^s, and the estimated performance based on surrogate time steps and observed labels $E_{cn}^{s,*}$ at evaluation timestamp T given waiting time W. The underscore in $y_{s,\_}^*$ is used to denote any given timestamp $T' < T_s$ when this label was observed. In particular, the observed labels $y_{s,\_}^*$ can be observed at any timestamp before T_s.

We can compute the continuous performance measurement at each evaluation time step $u \in \{1, \cdots, t\}$ in $\mathbb{E}$, enabling us to track the fluctuation of true continuous performance [31]. We can then use the average of those true performance metrics to quantify the total true continuous performance for the whole evaluation data stream $\mathbb{E}$ as

$$E_{cn} = \frac{1}{t}\sum_{u=1}^{t} E_{cn}(u), \qquad (2)$$

where $E_{cn}(u)$ is defined in Eq. (1). Such averaging process would not be used in practice when tracking predictive performance of JIT-SDP models, but it will be useful to evaluate the impact of waiting time on the validity of the continuous performance evaluation procedure, as will be explained in Section 5.

Computing the true performance nevertheless requires the true labels y_u of evaluation examples in the evaluation procedure, being infeasible in practice due to verification latency. Therefore, practitioners need to evaluate their models through an evaluation procedure that uses estimations of such true performance. In Sections 4.4.2 and 4.4.3, we will discuss two types of estimations of the true performance based on surrogate time steps and true labels, and surrogate time steps and observed labels, respectively.

4.4.2 Estimated Performance of JIT-SDP at Timestamp T Based on Surrogate Time Steps

If we use, for example, training examples corresponding to changes produced 90 days ago or older to estimate the current true performance, the time step of 90 days ago becomes a *surrogate* time step of now, and the true performance at the surrogate time step becomes an estimate of the true performance at current time step. In this sense, the true performance at a surrogate time step could be used as an estimate of the current true performance.

Given the waiting time W, the evaluation data stream at Unix timestamp T based on surrogate time steps that is used to estimate the true performance can be formulated as

$$\mathbb{E}^s = \{(X_u, y_u)\}_{u=1}^{t_s},$$

where the mathematical bold $\mathbb{E}$ is used to represent this data stream is used for evaluation, the superscript s is used to indicate that this is using surrogate time steps, and time step t_s corresponds to the surrogate timestamp $T_s = T - W$ of T in the evaluation data stream. As t_s represents the last evaluation time step, we can also use it to denote the number of evaluation time steps used in the estimation of the true performance. In the illustrative example of Figure 3, $\mathbb{E}^s$ would be composed of software changes from X_1 to X_s with their true labels from y_1 to y_s.

The *estimated continuous performance* of a JIT-SDP model at timestamp $U \leq T$ based on the surrogate time steps given the waiting time W can be formulated as

$$E^s_{cn}(u) = \theta \cdot E^s_{cn}(u_s - 1) + (1 - \theta) \cdot ||\widehat{y}_{u_s} - y_{u_s}||_G \quad (3)$$

where the subscript cn is used to indicate a predictive performance computed through a continuous evaluation procedure, the superscript s is used to indicate that this procedure uses surrogate time steps, time step u_s corresponds to the surrogate timestamp $U_s = U - W$ of U in $\mathbb{E}^s$, and $\theta \in (0, 1)$ is the forgetting factor. The predicted and true labels used for computing $E^s_{cn}(u)$ in the scenario where $u = t$ are illustrated in the orange box of Figure 3. This means that, even though the current timestamp is T, we can only use software changes committed up to T_s to estimate the true predictive performance at timestamp T.

We can compute the estimated continuous performance measurements at each time step $u \in \{1, \cdots, t\}$ in $\mathbb{E}^s$ using the continuous evaluation procedure explained above, enabling us to track the fluctuation of the estimated continuous performance based on surrogate time steps. We can then use the average of those estimated performance metrics to quantify the total estimated continuous performance for the whole evaluation data stream $\mathbb{E}^s$ as

$$E^s_{cn} = \frac{1}{t_s} \sum_{u=1}^{t_s} E^s_{cn}(u), \quad (4)$$

where time step t_s corresponds to the surrogate timestamp $T_s = T - W$ of T in $\mathbb{E}^s$ and $E^s_{cn}(u)$ is formulated in Eq. (3). As in Section 4.4.1, the averaging process from Eq. (4) would not be used in practice when tracking the predictive performance of JIT-SDP models, but it will be useful to evaluate the impact of waiting time on the validity of the continuous performance evaluation procedure in our study, as will be explained in Section 5.

4.4.3 Estimated Performance of JIT-SDP at Timestamp T Based on Surrogate Time Steps and Observed Labels

The estimated performance explained in Section 4.4.2 still assumes that the true predictive performance at the surrogate time step is available. However, the estimated performance at timestamp T based on surrogate time steps and *observed labels* is typically what would be available in practice. As T represents a given current time, no knowledge that comes after T would be available to determine the observed labels. This section formulates the continuous evaluation procedure introduced in Section 4.3, which can be adopted during software development in practice.

Given a waiting time W, we can set up the evaluation data stream at an arbitrary timestamp T when a JIT-SDP model is evaluated by using (1) software changes that have at least W waiting time for their observed labels and (2) those that are found defect-inducing after the surrogate evaluation timestamp $T_s = T - W$ and before T. The first part can be formulated as

$$\mathbb{E}^{s,*}_1 = \{(X_u, y^*_{u,T})\}_{u=1}^{t_s},$$

where $\mathbb{E}$ represents this data stream is used for evaluation, and the superscript $s, *$ indicates that this is using observed labels of evaluation examples at surrogate time steps. The second part can be formulated as

$$\mathbb{E}^{s,*}_2 = \{(X_u, 1)\}_{u=t_s+1}^{t}.$$

Altogether, the evaluation data stream at T based on surrogate time steps and observed labels is formulated as

$$\mathbb{E}^{s,*} = \mathbb{E}^{s,*}_1 \cup \mathbb{E}^{s,*}_2.$$

In this way, the whole evaluation data stream is produced and updated with time.

When all defect-inducing software changes are found before surrogate time step t_s ($\mathbb{E}^{s,*}_2 = \emptyset$), it is actually the last evaluation time step in $\mathbb{E}^{s,*}$. When there are defect-inducing software changes found after t_s and before T ($\mathbb{E}^{s,*}_2 \neq \emptyset$), the largest time step where software changes are found to be defect-inducing is the last evaluation time step in $\mathbb{E}^{s,*}$. Without loss of generalization and for the sake of simplicity, we will assume that t_s is the last evaluation time step in $\mathbb{E}^{s,*}$ in the remaining of this section. In this way, we can also use t_s to denote the number of evaluation time steps, which makes it explicit that surrogate time steps are being used. In the illustrative example of Figure 3, $\mathbb{E}^{s,*}$ would be composed of software changes from X_1 to X_s with the observed labels from $y^*_{1,\_}$ to $y^*_{s,\_}$.

The same evaluation example may appear more than once with different observed labels in the evaluation data stream, first as clean and then as defect-inducing. If a software change that was previously labeled as clean is afterwards found to be defect-inducing, the corresponding evaluation example will be updated with the label "defect-inducing" and presented again in the evaluation data stream for evaluation. Such change in the label of a given software change occurs when the chosen (evaluation) waiting time is smaller than the time it takes to find a defect associated to this software change. It results in an initially noisy clean evaluation example, whose label is later on changed to defect-inducing when the defect is found and fixed.

The *estimated continuous performance* of a JIT-SDP model at timestamp $U \leq T$ based on surrogate time steps and observed labels given waiting time W can be formulated as

$$E^{s,*}_{cn}(u) = \theta \cdot E^{s,*}_{cn}(u_s - 1) + (1 - \theta) \cdot ||\widehat{y}_{u_s} - y^*_{u_s,u}||_G \quad (5)$$

where the subscript cn indicates that this is a predictive performance computed through a continuous evaluation procedure, the superscript $s, *$ indicates that this estimated performance is based on observed labels at surrogate time

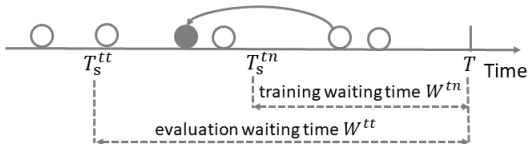


Fig. 4. Evaluation waiting time W^{tt} should be no larger than training waiting time W^{tn} in JIT-SDP. This figure shows the case $W^{tt} > W^{tn}$. The JIT-SDP model will be trained on examples at surrogate training timestamp T_s^{tn} before using them for evaluation at surrogate evaluation timestamp T_s^{tt}, violating the principle of the online evaluation scheme.

steps, time step u_s corresponds to the surrogate timestamp $U_s = U - W$ of U in $\mathbb{E}^{s,*}$, and $\theta \in (0, 1)$ is the forgetting factor. The predicted and observed labels used for computing $E_{cn}^{s,*}(u)$ in the scenario where $u = t$ are illustrated in the blue box of Figure 3. This means that we can only use software changes committed up to T_s to estimate the predictive performance at T, and their labels are the observed labels rather than the true labels.

We can compute the estimated continuous performance measurements at each time step $u \in \{1, \cdots, t\}$ in $\mathbb{E}^{s,*}$ using the continuous evaluation procedure explained above, enabling us to track the fluctuation of estimated continuous performance based on surrogate time steps and observed labels. We can then use the average of those estimated performance metrics to quantify the total estimated continuous performance for the whole evaluation data stream $\mathbb{E}^{s,*}$ as

$$E_{cn}^{s,*} = \frac{1}{t_s} \sum_{u=1}^{t_s} E_{cn}^{s,*}(u), \tag{6}$$

where time step t_s corresponds to the surrogate timestamp $T_s = T - W$ of the last timestamp T in the evaluation data stream $\mathbb{E}^{s,*}$ and $E_{cn}^{s,*}(u)$ is formulated in Eq. (5). As in Section 4.4.2, such averaging process will be useful to evaluate the impact of waiting time on the validity of the continuous performance evaluation procedure in our work.

4.4.4 Evaluation Waiting Time in the Evaluation Procedure

The continuous evaluation procedures explained in Sections 4.4.2 and 4.4.3 employ the waiting time W to cater for verification latency in the evaluation process, so the true performance of a JIT-SDP model at evaluation timestamp T (Section 4.4.1) can be estimated based on surrogate time steps or both surrogate time steps and observed labels. This is similar to the strategy to cater for verification latency in the training process [6], as a waiting time is also used to label the examples used for training. Nevertheless, the waiting time used for the training procedure and that used for the evaluation procedure do not need to be set the same. In particular, one may wish to choose a training waiting time that leads to a JIT-SDP model with better predictive performance, which might not be the same as the optimal evaluation waiting time that leads to the best trade-off between label noise and obsolescence of evaluation examples.

Whenever the two cases of waiting time need to be distinguished from each other, from this point onward, we will refer to the waiting time in the evaluation procedure as the *evaluation waiting time* (W^{tt}) and to the waiting time in the training procedure as the *training waiting time* (W^{tn}). In line with them, we will refer to the amount of label noise associated to the evaluation waiting time as the *evaluation label noise* (η^{tt}) and to the amount of label noise associated to the training waiting time as the *training label noise* (η^{tn}), when the two cases of label noise need to be distinguished.

In practice, evaluation waiting time should be no larger than training waiting time ($W^{tt} \leq W^{tn}$). Figure 4 illustrates the reason for such requirement. At an evaluation timestamp T, when $W^{tt} > W^{tn}$ as shown in this figure, the surrogate training timestamp $T_s^{tn} = T - W^{tn}$ would come later than the surrogate evaluation timestamp $T_s^{tt} = T - W^{tt}$. As a result, the JIT-SDP model would be trained on examples before using them for evaluation. This violates the principle of the online evaluation scheme that we should only use examples for evaluation *before* using them for training.

5 Formulation of Research Questions

This section aims to formalize our four research questions based on the notations and the online learning formulation of JIT-SDP in Section 4. In particular, we will define the validity of the performance evaluation procedure based on the difference between the true and the estimated performance.

5.1 RQ1: Impact of Waiting Time on Label Noise

RQ1 investigates the impact of waiting time on label noise for JIT-SDP. For that, we will adopt a metric corresponding to the proportion of defect-inducing software changes that are labeled as clean. This is because only defect-inducing examples can be mislabeled as clean as a result of waiting time (Section 4.2). Clean software changes cannot be mislabeled as defect-inducing. As our investigation considers a continuous performance evaluation procedure, we need to compute such proportion continuously over time. This section defines the metric for that.

It is worth noting that the investigation conducted for RQ1 is independent of whether the waiting time corresponds to the training or the evaluation procedure, because no actual training or evaluation needs to be performed in order to analyze the impact of waiting time on the amount of label noise in the data streams.

Denote a data stream $\mathbb{D} = \{(X_u, y_u)\}_{u=1}^{t}$, where the mathematical bold $\mathbb{D}$ represents this is an arbitrary data stream and t denotes the last time step (a.k.a. the length of the data stream). We will formulate the impact of waiting time on label noise from a continuous evaluation perspective. Given a waiting time W, the amount of label noise associated to it calculated continuously at each time step u of the data stream $\mathbb{D}$ can be formulated as

$$\eta_{cn}(u) = \frac{\sum_{s=1}^{u_s} \theta^{(u_s - s)} \cdot |y_{s,u}^* - y_s|}{\sum_{s=1}^{u_s} \theta^{(u_s - s)} \cdot y_s}, \tag{7}$$

where the subscript cn indicates that this refers to a continuous evaluation procedure, time step u_s corresponds to the surrogate timestamp $U_s = U - W$ of time step u in $\mathbb{D}$ and $\theta \in (0, 1)$ is the *fading factor* that controls the emphasis on software changes closer to the investigated time step u, enabling to track fluctuations in the amount of label noise over time.

This equation represents the proportion of noisy examples over the number of defect-inducing software changes at time step u, to which we will refer as *continuous label noise*.

As the type of label noise investigated in this paper can only happen to defect-inducing software changes, the numerator is not influenced by the status of clean data. A large $\eta_{cn}(u)$ indicates severe label noise induced by waiting time at time step u in a continuous evaluation scenario.

In this way, we can compute the impact of waiting time on the amount of label noise continuously at each time step $u \in \{1, \cdots, t\}$ of the data stream $\mathbb{D}$, tracking the fluctuations of continuous label noise. We can use the average of those metrics to quantify the total impact of waiting time on label noise calculated continuously for the whole data stream $\mathbb{D}$, which can be formulated as

$$\eta_{cn} = \frac{1}{t}\sum_{u=1}^{t}\eta_{cn}(u). \tag{8}$$

5.2 RQ2: Impact of Label Noise on the Validity of the Performance Evaluation Procedure

RQ2 investigates how reliable the predictive performance estimated by a continuous evaluation procedure based on observed labels is, given the amount of label noise incurred by waiting time. For that, we need to define a validity metric corresponding to how accurate the estimated predictive performance based on noisy data is with respect to the true predictive performance. This section defines such metric.

The estimated performance based on surrogate time steps approximates the true performance of a JIT-SDP model by considering concept drift in the evaluation procedure, whereas the estimated performance based on surrogate time steps and observed labels approximates the true performance by considering both concept drift and label noise in the evaluation procedure. Therefore, the difference between the two types of estimated performance can measure the impact of label noise (as the impact of concept drift cancels off) on approximating the true performance, which will be used to formulate the validity of the performance evaluation procedure for RQ2.

Given an amount of label noise η, the validity of the continuous performance evaluation procedure for a JIT-SDP model at Unix timestamp T can be measured based on the discrepancy between the estimated continuous performance based on surrogate time steps and that based on both surrogate time steps and observed labels, which is formulated as

$$\Delta_{cn}(\eta) = 1 - |E^{s}_{cn} - E^{s,*}_{cn}|, \tag{9}$$

where the subscript cn denotes that this refers to a continuous evaluation procedure and E^{s}_{cn} and $E^{s,*}_{cn}$ are the two types of estimated performance defined in Eqs. (4) and (6), respectively. It quantifies the accuracy of the continuous performance evaluation procedure in view of label noise η. A larger value for $\Delta_{cn}(\eta)$ indicates a better validity of the continuous performance evaluation procedure. The continuous label noise associated to a waiting time used in this context is computed according to Eq. (8).

5.3 RQ3: Impact of Waiting Time on the Validity of the Performance Evaluation Procedure

In JIT-SDP, a small waiting time may cause large label noise in the evaluation data stream, whereas a large waiting time may lead to obsolete data being used for performance evaluation. RQ2 investigates only the impact of label noise on the validity of the continuous performance evaluation procedure. RQ3 will investigate the combined impact of label noise and data obsolescence on such validity. For that, we need to define a validity metric corresponding to how accurate the estimated predictive performance based on noisy data and surrogate time steps is with respect to the true predictive performance. In other words, this metric reflects the similarity between the estimated continuous evaluation that can be done in practice and the true continuous evaluation that is inaccessible in practice.

As explained in Section 4.4.3, the estimated performance $E^{s,*}_{cn}$ is computed based on the evaluation time steps that are W period of time earlier than the current timestamp T. Therefore, the discrepancy between $E^{s,*}_{cn}$ and the true performance E_{cn} can measure the impact of waiting time on approximating the true performance, which will be used to formulate the validity of the performance evaluation procedure for RQ3.

Given a waiting time W, the validity of the continuous performance evaluation procedure for a JIT-SDP model at Unix timestamp T can be measured by the difference between the estimated continuous performance based on surrogate time steps and observed labels and the true continuous performance, which is formulated as

$$\Delta_{cn}(W) = 1 - |E_{cn} - E^{s,*}_{cn}|, \tag{10}$$

where the subscript cn indicates that this refers to a continuous evaluation procedure and E_{cn} and $E^{s,*}_{cn}$ are the true and the estimated performance defined in Eqs. (2) and (6), respectively. It quantifies the accuracy of the continuous performance evaluation procedure in view of the waiting time W. A larger value for $\Delta_{cn}(W)$ indicates a better validity of the continuous performance evaluation procedure.

5.4 RQ4: Impact of Waiting Time on the Validity of Model Ranking

As waiting time has significant impact on the validity of the performance evaluation procedure (RQ3), we want to know whether such impact will change the relative ranking of JIT-SDP models (RQ4). Then, we can judge how much the conclusions in terms of which JIT-SDP models perform better are potentially affected by validity issues. For that, we need a metric to quantify the similarity between (1) the estimated ranking produced based on the estimated predictive performance using observed labels and surrogate time steps and (2) the true ranking produced based on the true performance. This section proposes to use an existing ranking correlation metric for this purpose. This metric represents how concordant or discordant these rankings are.

Kendall Tau is a popular measure of ranking correlation that counts the numbers of concordant and discordant pairs to measure the similarity of two rankings [34]. Denote m_i ($\widehat{m}_i$) as the true (estimated) ranking of the i^{th} out of n models. If $m_i > m_j$ and $\widehat{m}_i > \widehat{m}_j$, the pair of the i^{th} and the j^{th} model is *concordant*; if $m_i > m_j$ and $\widehat{m}_i < \widehat{m}_j$, the pair is *discordant*; if $m_i = m_j$ or $\widehat{m}_i = \widehat{m}_j$, the pair is neither concordant nor discordant.

The similarity between the estimated and the true rankings of n JIT-SDP models can be formulated as

$$\tau(R, \widehat{R}) = \frac{\#(\text{concordant pairs}) - \#(\text{discordant pairs})}{C_n^2}, \quad (11)$$

where $\tau(\cdot,\cdot)$ denotes the Kendal similarity, R ($\widehat{R}$) denotes the true (estimated) ranking of JIT-SDP models in a certain performance evaluation scenario, $C_n^2 = \frac{n(n-1)}{2}$ is the binomial coefficient for the number of ways to choose two from n models and the operator $\#(\cdot)$ denotes the number of items. Kendall Tau is larger when the two rankings are more similar and is smaller if they are less similar.

Given a waiting time W, the validity of continuous model ranking in JIT-SDP at Unix timestamp T is

$$\Theta_{cn}(W) = \tau(R_{cn}, \widehat{R}_{cn}), \quad (12)$$

where the subscript cn indicates that this is related to a continuous evaluation procedure, R_{cn} denotes the true ranking of JIT-SDP models compared according to the true continuous performance (E_{cn} in Eq. (2)), and $\widehat{R}_{cn}$ denotes their estimated ranking compared according to the estimated continuous performance ($E_{cn}^{s,*}$ in Eq. (6)). A larger value for Θ_{cn} indicates better validity of model ranking in the practical evaluation scenario.

6 Datasets

We use 13 GitHub open source projects shown in Table 1 to investigate our research questions. They were chosen among projects with more than 4 years of duration, rich history (>10k commits) and a wide range of defect-inducing change ratio (from 2% to 45%). The first six projects were made available by Cabral et al. [6] and were chosen in this study for having relatively more software changes.

We use `Commit Guru` [35] to collected datasets from these software projects, which is a tool to extract the input features and labels of software changes. It implements the SZZ algorithm [10] when an issue tracking system is available and its approximate [1] otherwise. Eight of our projects used an issue tracking system (Brackets, Broadleaf, Fabric8, Rails, Rust, Tensorflow, VSCode and wp-Calypso), and five did not (Camel, Nova, Django, JGroups and Corefx).

The input features include 14 change metrics [1]: NS (number of modified subsystems), ND (number of modified directories), NF (number of modified files), Entropy (distribution of modified code across each file), LA (lines of code added), LD (lines of code deleted), LT (lines of code in a file before the change), FIX (whether or not the change is a defect fix), NDEV (number of developers that changed the modified files), AGE (average time interval between the last and the current change), NUC (number of unique changes to the modified files), EXP (developer experience), REXP (recent developer experience) and SEXP (developer experience on a subsystem). These metrics have been shown to be adequate indicators for JIT-SDP for both open source and commercial projects [1], [13], [5]. The second column of Table 1 shows the total number of extracted software changes for each project.

Our study requires collecting true and observed labels of software changes. The observed label can be determined based on the waiting time [6], as explained in Section 4.2. However, collecting true labels of software changes is less straightforward, as there is always a non-zero chance of a software change labeled as clean to be in fact defect-inducing. Consequently, we may never get the true label of a software change that is labeled as clean. The longer we wait to label a clean software change, the more confident we are that this change is really clean. For this consideration, all data streams used in this study contain software changes covering a period of at least four years, with most of them covering a period of more than eight years.

To collect true labels of software changes, we eliminate software changes from the latter periods of the data stream, as they are not 'old' enough for us to be confident on their true labels. We can then be confident about the true labels of the remaining software changes. Considering a data stream with a 10-year duration, the procedure to determine what software changes to eliminate is explained as follow. First, find the 99%-quantile of the verification latency of defect-inducing software changes (amount of time to find defects induced by a software change) in the whole data stream. Let's consider that the 99%-quantile value is 1.5 years. Then, eliminate the software changes that were committed during the latter 1.5 years of the data stream, which are referred to as the *tail* of this data stream. The remaining software changes cover a period of 8.5 years and will have had at least 1.5 years to find whether they have induced any defects. As the 99%-quantile of the verification latency is 1.5 years, we are very confident (with at least 99% confidence level) that the changes labeled clean are really clean. Defect-inducing labels are always considered to be true labels, as they cannot involve label noise caused by inappropriate waiting time. As explained in Section 3.4, label noise that does not result from waiting time is out of the scope of this study.

The 4th column of Table 1 lists the number of retained software changes for which we have at least 99% confidence that the labeling is correct. All projects have at least 5,000 software changes we are confident of their labeling. Therefore, we will retain the first 5,000 time steps in each software project for answering our research questions, so that all investigated projects have the same maximum data stream length. In this way, the impact of the length of the data stream will also be investigated in our analyses. As most software projects have considerably more than 5,000 time steps, the confidence level in the labels of their clean changes is higher than 99%, as more software changes at the tail of the valid data stream have been removed.

7 Experimental Setup

We adopt G-mean [33] to investigate our research questions. Different from other metrics such as F-measure and precision, G-mean is known to be robust to class imbalance, which is particularly important for studies suffering from class imbalance evolution such as JIT-SDP [6], [32], [36]. Consider that the positive (negative) class in JIT-SDP corresponds to defect-inducing (clean). G-mean is

$$\text{G-mean} = \sqrt{\frac{tp}{tp+fn} \cdot \frac{tn}{tn+fp}}, \quad (13)$$

where tp denotes true positives (the number of defect-inducing software changes that are predicted correctly), fn

TABLE 1
An overview of the software projects investigated in this work. '%Defect-inducing' denotes the percentage of defect-inducing software changes over the total number of extracted software changes for each project.

Project	Total Changes	%Defect-inducing Changes	Retained Changes at 99%-quantile	Time Period	Main Language
Brackets	11,601	34.02	5,997	12/2011 - 12/2017	JavaScript
Broadleaf	12,336	20.28	5,190	11/2008 - 12/2017	Java
Camel	30,229	20.67	9,850	03/2007 - 12/2017	Java
Fabric8	12,495	20.65	9,310	12/2011 - 12/2017	Java
jGroup	18,003	17.48	13,028	09/2003 - 12/2017	Java
Nova	26,313	44.34	14,900	08/2010 - 01/2018	Python
Django	26,360	42.64	14,236	07/2005 - 09/2019	Python
Rails	57,949	25.64	28,421	11/2004 - 09/2019	JavaScript
Corefx	26,627	6.91	7,611	11/2014 - 10/2019	python
Rust	73,876	2.02	35,766	06/2010 - 10/2019	python
Tensorflow	65,034	24.85	21,466	11/2015 - 11/2019	python
VScode	51,846	2.28	19,413	11/2015 - 10/2019	JavaScript
wp-Calypso	31,206	22.75	8,708	11/2015 - 10/2019	JavaScript

denotes false negatives (the number of defect-inducing software changes that are erroneously predicted as clean), tn denotes true negatives (the number of clean software changes that are predicted correctly) and fp denotes false positives (the number of clean software changes that are erroneously predicted as defect-inducing). As the false positive rate is defined as $1 - tn/(tn + fp)$, G-mean takes into account the trade-off between true positives and false positives. Larger G-means represent better predictive performance.

We use G-mean to implement the metrics $||\cdot||_G$ from Eqs. (1), (3) and (5) as the basic element to evaluate the predictive performance in JIT-SDP. The forgetting factor $\theta = 0.99$ is used in this paper following previous studies [6], [7]. Our preliminary experiment on studying various values of θ also showed the effectiveness of setting $\theta = 0.99$ for tracking the trend in performance fluctuation without emphasizing single examples too much.

Oversampling Online Bagging (OOB) with Hoeffding trees was adopted whenever JIT-SDP models needed to be created. This approach was chosen for being the state-of-the-art in dealing with online class imbalance learning [31] besides having been applied for JIT-SDP [6], [7].

A grid search based on the first 500 (out of the total 5,000) software changes in the data stream of a software project was conducted to choose the parameter settings based on G-mean. The parameters of the investigated JIT-SDP model (OOB ensemble of Hoeffding trees) include the decay factor $\in \{0.9, 0.99\}$ and the ensemble size $\in \{5, 10, 20\}$. Given a software project, the parameter combination leading to the best average G-mean across 30 runs at the first 500 time steps was adopted. The predictive performance of the JIT-SDP model with the best parameter setting was then evaluated based on the whole data stream of a software project. Hoeffding trees adopted the default parameters in the python package *scikit-multiflow* [37], following previous studies in JIT-SDP [6], [7]. All analyses and statistical tests are based on the mean performance across 30 runs based on the chosen parameter settings. Section V of the supplementary material lists the chosen parameter settings for each project. The code used for our experiments is available at https://github.com/sunnysong14/ContinualPerformanceValidityTSE2022.

8 Experimental Results and Discussion

8.1 RQ1 – Impact of Waiting Time on Label Noise

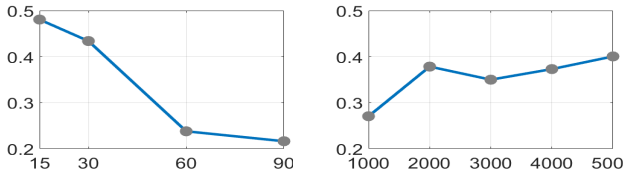
This section answers RQ1 based on the formulation in Section 5.1. This investigation is suitable to analyze both the impact of evaluation waiting time (for model evaluation, the focus of this paper) and training waiting time (for model training) on label noise in JIT-SDP.

8.1.1 Methodology

We perform Analysis of Variance (ANOVA) [38] to analyse the impact of waiting time on continuous label noise in the data stream $\mathbb{D}$. The length of the data stream will also be investigated to check whether it has an impact on the analysis. If we had assigned $\theta = 1$ in Eq. (7), it would be reasonable to hypothesize that larger data streams are less affected by the amount of label noise, as there would be more time to reveal the true labels of a large portion of older software changes. In practice, when computing the current amount of label noise sequentially with a fading factor as we did in the continuous evaluation scenario, the fading factor always places more emphasis on the most recent evaluation examples. So, the length of the data stream may not have a significant impact on the amount of label noise anymore.

The within-subject factors for ANOVA are the waiting time W and the length t of the data stream. The factor W varied among four levels (15, 30, 60 and 90 days). The waiting time of 90 days was investigated because it has been shown to be appropriate for training JIT-SDP models [6]. Waiting time values smaller than 90 days were investigated following McIntosh and Kamei's suggestion [5]. Waiting time older than 90 days may be too obsolete in view of concept drift [5]. The factor t varied among five levels (1000, 2000, 3000, 4000 and 5000 time steps). We set up the upper bound of 5000 time steps to get high confidence that the true labels in use are genuine as explained in Section 6. The response variables is the amount of continuous label noise (defined in Eq. (8)). ANOVA will be performed across the 13 datasets shown in Table 1, with level of significance of 0.05.

Sphericity is an important assumption made by the repeated measures ANOVA design, which is used in our study. We have adopted Greenhouse-Geisser corrections whenever Mauchly's test of sphericity detected violations to this assumption.



(a) Impact of waiting time. (b) Impact of data stream length.

Fig. 5. Plots of median continuous label noise across datasets (y-axis), for different waiting times and lengths of data streams (x-axis).

8.1.2 Impact of Waiting Time on Continuous Label Noise

Mauchly's tests show that the sphericity assumptions on W and t are violated (p-values 3.44-9 and 1.00E-6, respectively). Thus, Greenhouse-Geisser corrections have been adopted, based on which ANOVA reports that both W and t have significant impact on continuous label noise (p-values 0.0085 and 0.040636, respectively), whereas the interaction $W * t$ does not (p-value 0.258199). Effect size in terms of partial eta-square is 0.595 for W, 0.368 for t and 0.157 for $W * t$. Pairwise comparisons based on the post-hoc Bonferroni test between waiting times find significant difference between 15 vs 30 days (p-value 0.008), between 15 vs 90 (p-value 0.037) and between 60 vs 90 days (p-value 0.009). Pairwise comparisons between the data stream lengths do not find significant difference between any pair, possibly because the post-hoc comparisons are weaker than ANOVA.

Figure 5(a) shows the plots of median continuous label noise across projects for different waiting times. We can see that larger waiting times are associated to smaller amounts of continuous label noise. This is reasonable as a larger waiting time allows for higher possibility to find defect-inducing software changes, contributing to smaller continuous label noise. We can also see that the magnitude of the differences in continuous label noise when varying waiting time from 15 to 90 days was high, causing a drop of 26.43% in the proportion of noisy defect-inducing examples. Therefore, the differences are typically considerable, even though they can be smaller for some projects (minimum of about 10% in jGroup) and very large for others (maximum of 71.32% in wp-Calypso). Section 8.2.2 will investigate whether such amounts of label noise lead to a significant impact on the validity of the continuous performance evaluation procedure.

Figure 5(b) shows the plots of median continuous label noise across projects for different lengths of data streams. We can see that larger data streams are generally associated to larger amounts of continuous label noise. A smaller data stream length of 1,000 caused a drop of 12.99% in the proportion of noisy defect-inducing examples compared to a larger length of 5,000. Therefore, the differences are typically considerable, even though they can be smaller for some projects (minimum of around 5% in Fabric8) and larger for others (maximum of 19.62% in VScode). The increasing amount of continuous label noise as the length of the data stream increases is rather unexpected. We will discuss the reasons behind this result in Section 8.1.3.

8.1.3 Why Are Larger Data Streams Associated to More Continuous Label Noise?

We will first present our original conjecture on the impact of waiting time on continuous label noise and then explain why it does not hold true. Then, we will discuss the reason why larger data streams have larger (rather than smaller) amounts of continuous label noise.

Original conjecture. We conjectured that the length of the data stream would not have any particular (neither significant positive nor significant negative) impact on *continuous* label noise for the following reason. By using a fading factor, continuous label noise $\eta_{cn}(u)$ at time step u (Eq. (7)) is mainly determined by a few neighboring software changes that arrived shortly before time step u. Examples from much earlier time steps than u contribute little to $\eta_{cn}(u)$ as their effect is exponentially decayed with time. As $\eta_{cn}(u)$ is calculated based on the observed labels received up to time step u (rather than the knowledge obtained until the end t of the data stream $\mathbb{D}$), the snapshot of the data stream that is effectively used to compute $\eta_{cn}(u)$ would be given the same amount of time to obtain the observed labels for different time steps u, and any time step u should thus typically hold the same amount of label noise. So, the total amount of continuous label noise η_{cn} (Eq. (8)) would remain similar for different lengths t of the data stream.

Why our conjecture does not hold true? Figure 6 shows the continuous label noise throughout time steps with waiting time 15 days on the projects. Other waiting times have also been studied, showing similar patterns. Many projects, such as Bracket, Broadleaf, Tensorflow, Nova, Django and Rails, demonstrate a positive correlation between the amount of label noise and the length of the data stream, meaning that there is typically more label noise in the latter portions of the data stream. So, despite the snapshot effectively used to compute $\eta_{cn}(u)$ being given *the same amount of time* to collect the observed labels is independent of the value of u, the amount of label noise in different portions of the data stream still differs considerably from each other. And, as different time steps are associated to considerably different amounts of continuous label noise, the length t of the data stream can have a significant impact on the total amount of label noise η_{cn}. Though there are a few projects, such as jGroups and Calypso, for which continuous label noise and the length of the data stream have slightly negative correlation, they are rather weak negative correlations.

Why there is typically more continuous label noise at latter portions of the data streams? As all time steps u were given the same amount of time to observe labels, the larger amounts of continuous label noise at the latter portions of the data streams probably arise from increasing levels of verification latency for defect-inducing software changes, which is an intrinsic property of the data stream regardless of waiting time. The *continuous level of verification latency* of software changes in a data stream $\mathbb{D}$ can be formulated as

$$\tau_{cn}(u) = \frac{\sum_{s=1}^{u} \theta^{(u-s)} \cdot \tau_s}{\sum_{s=1}^{u} \theta^{(u-s)} \cdot y_s}, \tag{14}$$

where the subscript cn indicates that this refers to a continuous evaluation process, u denotes a time step between $1 \sim t$ in $\mathbb{D}$, $\theta \in (0, 1)$ denotes the fading factor that controls the emphasis on older examples and those closer to u, τ_s denotes the amount of time taken to find the defect-inducing label associated to the software change at time step s or zero if this change is clean and y_s denotes the software change's true label. In this way, we can compute verification latency

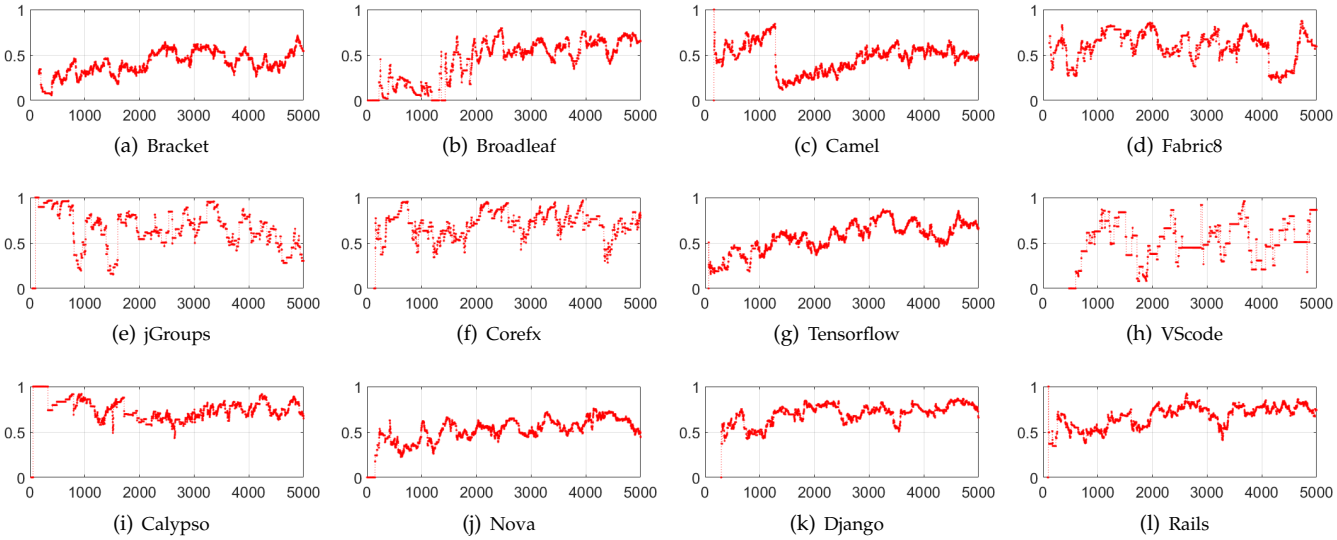


Fig. 6. Continuous label noise throughout time steps on the projects investigated in the paper with the waiting time 15 days. The x-axis represents the length of data stream and the y-axis represents the continuous label noise. The fading factor is 0.99. We randomly omit the dataset Rust for a better and simpler graph allocation (the length of data stream is firstly positively and then negatively correlated with the continuous label noise).

continuously at each time step $u \leq t$ of the data stream $\mathbb{D}$, tracking the fluctuations of the delay to find defect-inducing software changes. We can use the average of these values to quantify the total *continuous verification latency* for the whole data stream $\mathbb{D}$, which can be formulated as

$$\tau_{cn} = \frac{\sum_{u=1}^{t} \tau_{cn}(u)}{t}. \tag{15}$$

To better understand the reason for larger data streams to be associated to larger continuous label noise in JIT-SDP, we will first investigate the impact of the length t of the data stream on the continuous verification latency τ_{cn}, and then investigate the relation between τ_{cn} and the continuous label noise η_{cn} (Eq. (8)). We report our results as follows.

Step 1. Positive correlation between the length t of data stream and continuous verification latency τ_{cn}. To check whether t has a positive impact on τ_{cn}, we will calculate the Spearman's rank correlation between the length of data stream and the median value of continuous verification latency across the 13 projects. Spearman's rank correlation $r_s \in [-1, +1]$ is a non-parametric statistic that assesses how well the relationship between two variables can be described by a monotonic function [39]. The value +1/-1 means a perfectly increasing/decreasing monotone of one variable over the other. Conventionally, the correlation strength $|r_s|$ is interpreted according to Fieller and Pearson's [39] as 0.00 – 0.19 "very weak", 0.20 – 0.39 "weak", 0.40 – 0.59 "moderate", 0.60 – 0.79 "strong", and 0.80 – 1.00 "very strong".

Spearman correlation reports a strong positive correlation (0.7000) between the length of data stream t and the median of continuous verification latency τ_{cn}, confirming the positive impact of the length of data stream on continuous verification latency.

Considering the fact that verification latency is the length of the delay to detect a defect induced by a software change, the positive correlation between t and τ_{cn} means that, the later in the software development process a change is created, the longer it takes to find and/or fix defects induced by it. Possibly, during earlier stages, the code is put more under scrutiny, so that defects would be found more quickly. It may also be that, as the software code becomes larger (and probably more complex and tangled) over time, it typically becomes increasingly more time consuming to fix a defect. And, the longer it takes to fix a defect, the longer time it takes to find out the corresponding defect-inducing software change, meaning that more continuous label noise would likely be generated. Therefore, we investigate whether larger verification latency is associated to larger amounts of continuous label noise in our next step.

Step 2. Positive impact of continuous verification latency τ_{cn} on continuous label noise η_{cn}. We will perform bi-variate linear regression across 13 projects to investigate the relation between τ_{cn} and η_{cn}. Specifically, τ_{cn} and waiting time W are the independent variables, and η_{cn} is the dependent variable. W is included because it has been shown to have significant impact on η_{cn}. ANOVA is not applicable as the independent variable is continuous rather than ordinal. We have checked the assumptions of linear regression to investigate the impact of both τ_{cn} and W on η_{cn}. The analysis shows that the assumptions are well satisfied and is presented in Section IV-A of the supplementary material, due to space restrictions.

The linear regression statistics show that the linear relationship between the independent variables and the dependent variable is significant (p-value 1.5087E-12), indicating the suitability of using the linear regression statistic for this analysis. The model coefficients show that both τ_{cn} and W have significant impact on η_{cn} (p-values 7.144E-9 and 2.6753E-9, respectively). The standardized coefficient is 0.572 for τ_{cn} and -0.595 for W, showing that larger continuous verification latency (waiting time) is associated to significantly larger (smaller) continuous label noise.

Summary of the two-step analysis. Latter points in time in a data stream tend to have larger continuous verification latency, whereas larger continuous verification latency is associated to larger amounts of continuous label noise. Therefore, it is reasonable for the length of data stream to have positive impact on continuous label noise.

Answer to RQ1: Smaller waiting times were found to be associated to significantly larger label noise. The amount of continuous label noise increased by up to 71.32% as a result of smaller waiting time. The length of data stream had significant impact on continuous label noise, with larger data streams leading to more continuous label noise due to the larger verification latency of defect-inducing software changes in the latter portions of the data streams.

8.2 RQ2 – Impact of Label Noise on the Validity of the Performance Evaluation Procedure

This section investigates RQ2, formulated in Section 5.2.

8.2.1 Methodology

We will use linear regression to investigate the impact of evaluation label noise (i.e., the label noise associated to waiting time in the evaluation data stream) on the validity of the continuous performance evaluation procedure. Both the continuous evaluation and training label noise will be included as independent variables, for a more thorough analysis. Including training label noise enables the analysis to consider the extent to which different trained JIT-SDP models could impact the conclusions of the study. They are both computed based on Eq. (8), but the training label noise is computed based on the training waiting time, and the evaluation label noise is computed based on the evaluation waiting time. The dependent variable is the validity of the continuous performance evaluation procedure (Eq. (9)). ANOVA, which was used to answer RQ1, is not viable for RQ2 because the independent variables are continuous rather than ordinal, and thus we cannot set up the levels of within-subject factors [38]. A discussion of linear regression's assumptions for this analysis is provided in Section IV-B of the supplementary material.

8.2.2 Impact of Label Noise on Continuous Validity

Linear regression shows that the linear relationship between the two independent variables and the dependent variable is significant (p-value 3.2885E-4). Thus, we can continue to investigate the model coefficients.

Continuous evaluation label noise has significant impact on the validity of the continuous evaluation procedure (p-value 0.0485). The standardized coefficient is -0.01935, showing that the continuous evaluation label noise has significant negative impact on such validity. Even though the impacts of continuous evaluation label noise on the validity in individual projects were different, most of them presented decreasing trends (except for Camel, Fabric8, jGroup, Corefx and wp-Calypso). Therefore, larger continuous evaluation label noise typically means significantly worse validity of the continuous performance evaluation procedure.

Nevertheless, the drops in the magnitude of the validity suffered by larger evaluation continuous label noise were small. The median drop across projects was only 0.3614%. The maximum drops were observed in Django and VScore (up to about 3%). Therefore, a larger continuous evaluation label noise means a worse validity of the evaluation procedure, but the magnitude of the drop in the validity is small.

Moreover, the validity of the performance evaluation procedure with respect to label noise was typically high (median value of around 97% across projects and waiting times). This suggests that the use of observed labels is acceptable for the continuous performance evaluation procedure in online JIT-SDP.

No significant impact of continuous training label noise has been found on the validity of the performance evaluation procedure (p-value 0.0611). It is worth noting that this does not mean that training label noise has no impact on the quality of the resulting model. Rather, it has no impact on the *validity of the evaluation* of the resulting model.

Answer to RQ2: Continuous evaluation label noise had significant negative impact on the validity of the continuous performance evaluation procedure. However, different amounts of label noise led to differences of up to around 3% in the validity of the evaluation procedure, being of small magnitude. The validity of the evaluation procedure was typically high (median of around 97%) despite the label noise resulting from waiting time.

8.3 RQ3 – Impact of Waiting Time on the Validity of the Performance Evaluation Procedure

This section answers RQ3, formulated in Section 5.3.

8.3.1 Methodology

We will use linear regression to analyse the impact of waiting time on the validity of the performance evaluation procedure. Both the evaluation and the training waiting time will be included as independent variables to enable a more thorough analysis of the validity. The validity of the performance evaluation procedure (defined in Eq. (10)) is the dependent variable. ANOVA used in RQ1 is not viable for RQ3. This is because there is a constraint between the two independent variables: the evaluation waiting time should be equal to or smaller than the training waiting time, as discussed in Section 4.4.4, and thus the number of levels to be investigated for the evaluation waiting time varies depending on the training waiting time. Section IV-C of the supplementary material provides a discussion of the assumptions of the linear regression analysis.

8.3.2 Impact of Waiting Time on Continuous Validity

Linear regression analysis shows that the linear relationship between the two independent variables and the dependent variable is significant (p-value 0.008720). Thus, we can continue to investigate the model coefficient.

Evaluation waiting time has significant impact on the validity of the performance evaluation procedure (p-value 0.004574). The standardized coefficient is -0.284094, showing that the evaluation waiting time has significant negative impact on the validity of the continuous performance evaluation procedure. Figure 7 shows the relationship between evaluation waiting time (in x-axis) and validity of the evaluation procedure (in y-axis). We can see that even though larger evaluation waiting times were sometimes associated to better validity (Fabric8, Tensorflow, Nova and Django), large evaluation waiting times typically negatively impacted the validity, as reflected by the negative standardized coefficient found by linear regression.

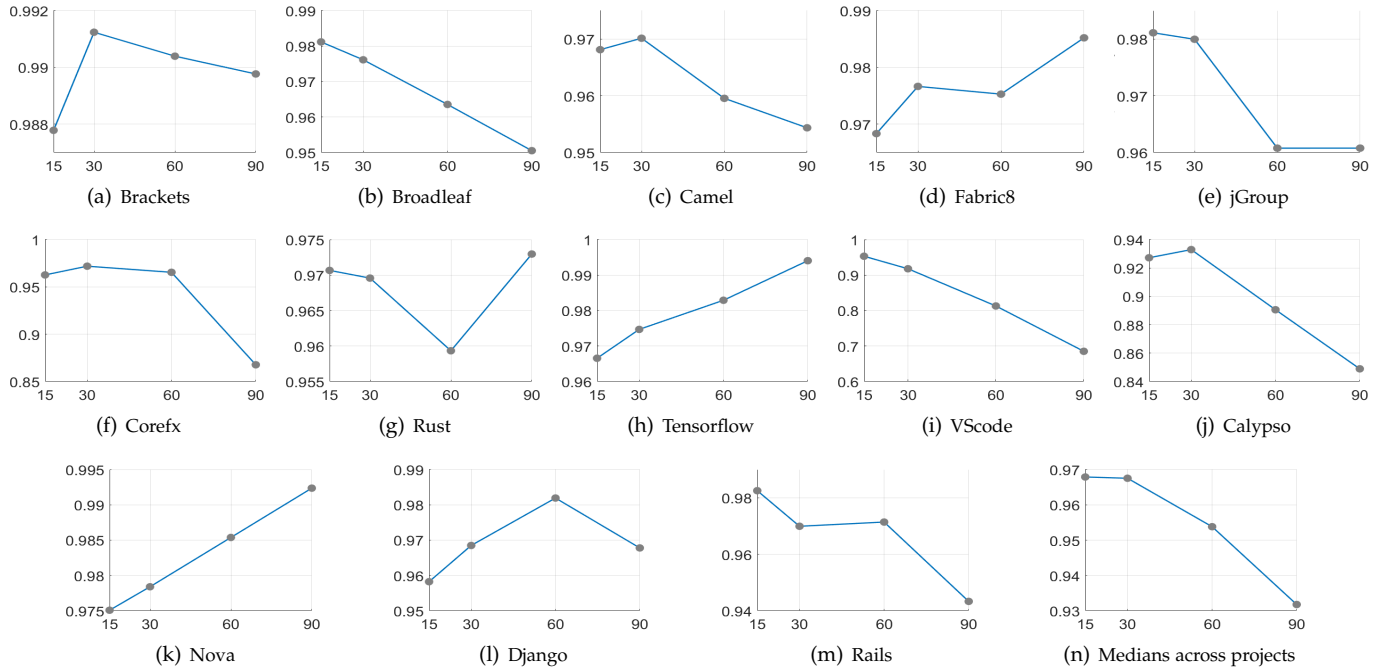


Fig. 7. Impact of the evaluation waiting time (in x-axis) on the validity of the continuous performance evaluation procedure (in y-axis). Note that the range of y-axis may not be the same in all the plots in order to facilitate visualization.

Figure (7(n)) shows the plot of median impact of evaluation waiting time on the validity across projects. A larger evaluation waiting time of 90 days causes a median drop of 3.73% in the validity compared with that of 15 days. Figure 7 also shows that in some projects, despite the significant impact of evaluation waiting time on the validity, the drops in the magnitude suffered by worse choices of evaluation waiting time were not large, with the validity differing by less than 1% (e.g., Brackets and Django). However, for some projects, the differences were larger by up to around 25% (VScode). This means that poor choices of evaluation waiting time can potentially lead to considerable differences in estimated predictive performance. Therefore, we should carefully choose the evaluation waiting time to obtain a good estimate of predictive performance over time. Such choice will be further discussed in Section 8.5.

No significant impact of the training waiting time on the validity of the continuous performance evaluation procedure was found (p-value 0.718232). It is worth noting that it is encouraging to see that the validity of the evaluation procedure obtained with the best values for evaluation waiting time was typically high (median value of a bit less than 97% across projects). This suggests that the use of observed labels and surrogate time steps is acceptable for the continuous performance evaluation procedure in online JIT-SDP when appropriate evaluation waiting times are used.

8.3.3 Why Evaluation Waiting Time Had a Negative Impact on the Validity of the Performance Evaluation Procedure?

The results in Section 8.3.2 can sound surprising given the results from Sections 8.1 and 8.2. Section 8.1 found that larger waiting time significantly reduced the amount of label noise, and Section 8.2 found that smaller amounts of label noise led to significant better validity of the performance evaluation procedure. Therefore, larger evaluation waiting time should in principle correspond to better validity of the performance evaluation procedure in view of evaluation label noise. This is illustrated in the top part of Figure 8.

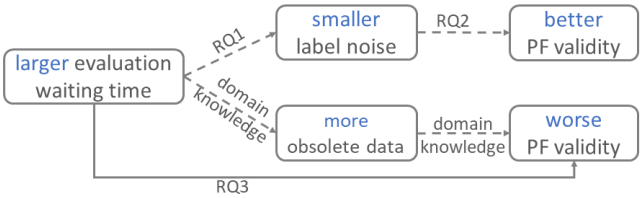


Fig. 8. Two-way evaluation waiting times can affect the validity of the continuous performance evaluation procedure in JIT-SDP. 'PF' is short for 'performance'. RQ3 shows that evaluation waiting time has a significant negative impact on the validity, shown in the solid line.

However, Section 8.3.2 shows that the evaluation waiting time generally has a significant negative impact on the validity of the performance evaluation procedure. It means that, even though larger evaluation waiting time is related to better validity in view of the smaller label noise that it produces, there must be something other than label noise that has an even larger effect on the validity.

It is reasonable that a larger evaluation waiting time makes the validity of the evaluation procedure better by reducing the amount of label noise, but one cannot ignore the fact that JIT-SDP is also likely to suffer from concept drift [5], [6], [7]. It is difficult to quantify concept drift in real world problems, because that would require access to the true underlying probability distribution of the problem, which is unavailable in practice. Therefore, it is not possible to perform direct analyses of the impact of waiting time on the validity of the evaluation procedure in view of concept drift as done in view of label noise in RQ1 and RQ2.

Nevertheless, by definition larger evaluation waiting

times cause obsolete data to be used to estimate the current status of the predictive performance via surrogate time steps if there is concept drift. This is illustrated in the lower part of Figure 8. In this sense, a larger evaluation waiting time will correspond to a worse validity of the evaluation procedure in view of concept drift, or more specifically an obsolete performance evaluation.

To experimentally justify this conjecture, we can isolate the effect of the obsolescence of evaluation data to conduct an analyses as specified by the difference between true performance at a given evaluation timestamp T (Section 4.4.1) and the estimated performance based on surrogate time steps (Section 4.4.3). The estimated performance based on surrogate time steps excludes the impact of label noise on the estimation of the evaluation procedure. Therefore, the difference between it and the true performance reflects how valid the evaluation procedure is in view of the obsolescence of the data being used for evaluation purposes.

The impact of concept drift on the validity of the continuous performance evaluation procedure can be analyzed based on the following validity metric:

$$\Delta_{cn}(CD) = 1 - |E_{cn} - E^s_{cn}| \tag{16}$$

where the subscript cn indicates that this is a continuous evaluation and E_{cn} and E^s_{cn} are the true and the estimated performance defined in Eqs. (2) and (4), respectively. A larger value for $\Delta_{cn}(CD)$ indicates a better validity of the evaluation procedure in view of data obsolescence.

We use linear regression to analyse the impact of data obsolescence on the validity of the performance evaluation procedure. Both the evaluation and the training waiting times are included as independent variables, where the evaluation waiting time corresponds to different levels of data obsolescence and the training waiting time corresponds to JIT-SDP models with different prediction capability. The validity of the evaluation procedure defined in Eq. (16) is the dependent variable. Section IV-C of the supplementary material provides a discussion of the assumptions of the linear regression analysis.

Linear regression analysis shows that the linear relationship between two independent variables and the dependent variable is significant (p-value 0.000022). Thus, we can continue to investigate the model coefficients. No significant impact of the training waiting time on the validity of the evaluation procedure in view of concept drift was found (p-value 0.737560). Evaluation waiting time in view of concept drift had significant impact on the validity of the evaluation procedure (p-value 0.000096). The standardized coefficient is -0.378022, showing that the larger the evaluation waiting time, the larger the difference between the continuous performance evaluation based on the current data and the surrogate data. This means that larger evaluation waiting time leads to an obsolete performance evaluation, potentially due to concept drift. This is consistent with our conjecture illustrated in the lower part of Figure 8.

We have now shown (as illustrated in Figure 8) that, on the one hand, larger evaluation waiting time is related to better validity of the evaluation procedure in view of the smaller label noise it produces (RQ2); on the other hand, it can also cause worse validity of the evaluation procedure in view of producing more obsolete data (RQ3). Given the results of RQ3, the effect of concept drift were larger than the effect of label noise, being a more important consideration to take when choosing evaluation waiting time.

This implies that the concept drifts suffered by JIT-SDP cause drops in predictive performance that are frequently larger than the errors in the estimate of the predictive performance caused by label noise. Indeed, it is known that JIT-SDP can suffer drops in predictive performance as large as around 40% G-mean over time [7]. If recent data is not used to continuously evaluate the predictive performance of obsolete models, such large drops in predictive performance will not be noticed until a much later date in practical scenarios, having a worse effect than label noise.

Answer to RQ3: Evaluation waiting time has significant impact on the validity of the evaluation procedure, with larger waiting times being typically associated to worse validity. This implies that concept drift is a more important issue to take into consideration than label noise when evaluating JIT-SDP models. The validity of the evaluation procedure had up to around 25% difference when adopting poor evaluation waiting time choices.

8.4 RQ4 – Impact of Waiting Time on the Validity of the Model Ranking Procedure

This section answers RQ4, formulated in Section 5.4.

8.4.1 Methodology

In JIT-SDP, different training waiting times lead to different JIT-SDP models, which in turn may perform differently. Therefore, this section will make use of different training waiting times to produce different models (OOB with Hoeffding trees), and investigate whether the ranking of these models changes depending on the evaluation waiting time being adopted. Given the training waiting time of 15, 30, 60 and 90 days, we will have four JIT-SDP models, denoted by $\mathcal{M}_{15}$, $\mathcal{M}_{30}$, $\mathcal{M}_{60}$ and $\mathcal{M}_{90}$ (defined in Section 4.3). For simplicity, the notation of time step 't' is omitted.

As we are analyzing a single factor (evaluation waiting time), we will perform the non-parametric Friedman test instead of ANOVA to investigate its impact on the validity of the model ranking procedure in JIT-SDP, which, as explained in Section 5.4, is defined based on Kendall Tao ranking correlation between the true and estimated rankings. Given an evaluation waiting time W, we will compute the true ranking R_W and the estimated ranking $\widehat{R}_W$ between the JIT-SDP models $\mathcal{M}_{15}$, $\mathcal{M}_{30}$, $\mathcal{M}_{60}$ and $\mathcal{M}_{90}$ in each evaluation data stream. Then, we will compute the Kendall Tao ranking correlation $\tau(R, \widehat{R})$ (Eq. (11)) to measure the validity of the model ranking procedure. As the evaluation waiting time should be no larger than the training waiting time (Section 4.4.4), we will use three evaluation waiting times (15, 30 and 60 days), leading to three ranking similarities for each evaluation data stream. Each ranking similarity measures how close the estimated ranking is from the true ranking. The evaluation waiting time of 90 days is not investigated because it only corresponds to a single model $\mathcal{M}_{90}$. We will conduct Friedman test across projects to investigate whether the evaluation waiting time has a significant impact on the validity of the ranking procedure.

TABLE 2
The similarity in terms of Kendall Tau correlation between the estimated and true rankings of JIT-SDP models on each project. Each column corresponds to an evaluation waiting time based on which the estimated performance is computed.

Project	15 days	30 days	60 days
Brackets	1.0000	1.0000	1.0000
Broadleaf	0.6667	0.3333	1.0000
Camel	0.6667	0.3333	1.0000
Fabric8	1.0000	1.0000	1.0000
jGroup	1.0000	1.0000	1.0000
Nova	1.0000	1.0000	1.0000
Django	0.6667	-0.3333	1.0000
rail	1.0000	1.0000	1.0000
Corefx	1.0000	1.0000	1.0000
Rust	0.6667	0.3333	1.0000
Tensorflow	1.0000	1.0000	1.0000
VScode	0.6667	0.3333	1.0000
wp-Calypso	0.6667	1.0000	1.0000

8.4.2 Impact of Evaluation Waiting Time on Model Ranking

Table 2 shows the Kendall Tao correlation between the true and estimated rankings for each pair of projects and evaluation waiting time. In most projects, the estimated ranking can perfectly approximate the true ranking of JIT-SDP models, with many Kendal correlation being equal to 1.0000. This means that the choice of evaluation waiting time has little impact on the validity of the JIT-SDP model ranking procedure, despite having significant impact on the validity of the performance evaluation procedure (the answer to RQ3). The Friedman tests confirm that the evaluation waiting time has no significant impact on the validity of the model ranking procedure across projects (p-value of 0.1376). Therefore, when comparisons among models are made based on their rankings across projects (to determine which model is better than others across projects), the choice of evaluation waiting time is unlikely to significantly impact the validity of such comparisons when continuously evaluating JIT-SDP performance. This implies that the estimation of the true predictive performance of different models is affected in similar ways when all of them are evaluated using the same waiting time, resulting in no significant change in their rankings.

Answer to RQ4: Conclusions on which model performs better are likely reliable independent of the choice of waiting time, especially when considered across projects.

8.5 Choosing Appropriate Evaluation Waiting Times

As explained in Section 8.3.3, good choices of evaluation waiting time are generally affected by (1) the time it takes to find defects and (2) concept drift. Larger time to find defects would lead to the need for longer evaluation waiting times to reduce label noise, whereas more severe concept drifts would lead to the need for shorter evaluation waiting times. The choice of an appropriate evaluation waiting time depends on this trade-off between the time to find defects and the severity of concept drifts, being not straightforward. In particular, the reason for upward or downward trends in Figure 7 is the result of such trade-off. If the time to find defects takes over, the optimal evaluation waiting time should be larger, and the plot between the evaluation waiting time and the validity of the evaluation procedure would have an upward trend. If the impact of concept drift takes over, the optimal evaluation waiting time should be smaller, leading to a downward trend.

This analysis thus leads to the following additional questions: How to choose appropriate evaluation waiting times for a given project in order to obtain good validity of the evaluation procedure? What evaluation waiting times would typically be the best to adopt? This section discusses the answers to these questions based on the insights provided by RQ1∼3. To answer these two questions, we first provide an analysis of whether the values of the input features together with information on the time to find defects could inform the choice of evaluation waiting time (Section 8.5.1). We then provide an analysis to check whether concept drifts are more important than the time to find defects for the purpose of determining a good evaluation waiting time (Section 8.5.2). Finally, we provide an analysis of what evaluation waiting time typically leads to the best validity of the evaluation procedure across projects (Section 8.5.3).

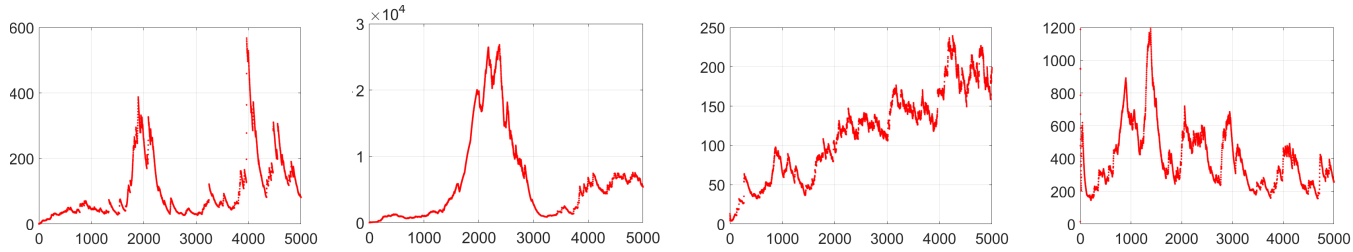
8.5.1 Factors That May Influence the Choice of Evaluation Waiting Time

The time it takes to find defects is potentially affected by various underlying factors, some of which are more or less controllable, and more or less objective. For example, a defect could be in an area of the code that is difficult to reach by test cases or that is not typically accessed by users. As a result, it takes longer for the defect to be found. Or, certain parts of the code could have been more tested or tested earlier than others, such that certain defects are found earlier. Or, certain parts of the code could have been tested by more experienced developers or may be more complex than others. As a result, the defect would be more or less difficult to find and fix.

Concept drift may also have multiple underlying causes. For example, a concept drift may be triggered by a new functionality being developed, a major refactoring of the code being conducted, the software maturing, the management strategy being used during the software development changing, or even more indirect factors such as software developers starting to work from home due to covid19, employees being over-tired due to another concurrent project reaching its final stages, or an economic crisis placing the company under extra stress, etc.

In this section, we conduct a systematic investigation of whether variations in the values of the input features (listed in Section 6) over time could inform the choice of evaluation waiting time. Such input features do not capture all possible aspects that could be related to the time to find defects or to concept drift. However, they are concrete and direct factors that could reflect concept drift and can be automatically collected from software repositories. Therefore, if these features can be used to inform the choice of evaluation waiting time, monitoring them could potentially be a feasible approach to help tuning evaluation waiting time in practice.

We randomly select six software projects for this analysis: Brackets, Broadleaf, Camel, Fabric8, jGroup and Nova. Given project, we compute the continuous values for each input feature by making use of a forgetting factor of 0.99



(a) Broadleaf's NUC (b) Broadleaf's EXP (c) Nova's NUC (d) Nova's EXP

Fig. 9. Time decayed mean values of features NUC and EXP for Nova and Broadleaf over time (in x-axis) with $\theta = 0.99$.

as previously done in Section 7. We then visually compare the plots of these input features over time for these projects to check which of them have relatively large variations. A project whose features have relatively large variations may have more severe concept drifts, requiring an evaluation waiting time that is smaller than its median time to find defects. Conversely, a project whose features have relatively small variations over time may require a evaluation waiting time that is larger than its median time to find defects.

Figure 9 gives an example of how to utilize input features to help with the choice of evaluation waiting time for Broadleaf and Nova. These figures show the mean values of the features NUC (number of unique changes to the modified files) and EXP (developer experience) over times. The median time to find defects in Broadleaf was 43 days [6]. Its NUC and EXP values have large variations over time (note the scale of the y-axis). Such large variations are likely to require evaluation waiting times that are even smaller than 43 days. Indeed, from Figure 7, the evaluation waiting time that led to the best validity of the performance evaluation procedure for this project was 15 days. Nova, on the other hand, had a larger median time to find defects of 97 days [6]. Moreover, the variations in NUC and EXP over time were much smaller for this project. Such larger median time to find defects, together with smaller variations in NUC and EXP, is likely to support the choice of larger evaluation waiting times. Indeed, from Figure 7, the best evaluation waiting time was 90 days.

Plots for other features and projects were omitted for space considerations. Brackets presents a similar behavior to Nova in the sense that the variations in the input features were small and its best evaluation waiting time was larger than its median time to find defects. For Camel and Fabric8, the variations are larger than those of Nova and Brackets, but not so large as those of Broadleaf. This makes it more difficult to use the input features to inform the choice of evaluation waiting time for these projects. Furthermore, for jGroup, the variations were similar to those in projects that did not require smaller evaluation waiting times, despite the best evaluation waiting times for this project being small.

Therefore, it is not easy to know how large the variations need to be to support smaller choices of evaluation waiting time. Moreover, even though large variations in the values of the features can potentially be used as indicative of the need for smaller evaluation waiting times, small variations do not always mean that larger evaluation waiting times should be used. This is because (1) the number possible factors potentially reflecting concept drift is very large, going well beyond just the input features being used for JIT-SDP, as discussed in the beginning of this section; (2) these factors may have complex interactions with each other; and (3) they may affect different pieces of code differently. Therefore, the choice of an appropriate evaluation waiting time is not straightforward, and one cannot always rely only on the variations in the input features to inform this choice.

Monitoring all possible factors that could influence the choice of waiting time and their interactions would be impractical. Nevertheless, if in practice a software manager knows that a significant change is likely to happen to the project, this could imply a severe concept drift. A shorter evaluation waiting time could thus be adopted. For example, if they know they will start a major refactoring or will implement a major new functionality, they may opt for a lower evaluation waiting time.

8.5.2 The Influence of the Time To Find Defects on the Choice of Evaluation Waiting Time

As explained in Section 8.5.1, monitoring input features over time is not always enough to inform the choice of waiting time. Therefore, it would be useful to know to what extent this choice could be made based on the time to find defects itself. This amount of time could potentially be monitored over time to inform the choice of waiting time. From the discussion on the results of RQ1∼3, we know that larger evaluation waiting times are associated to smaller label noise, and that smaller label noise is associated to better validity of the evaluation procedure. However, those results also show that concept drift is likely to play a more important role than label noise on the validity. We also know that label noise is caused by the effect of verification latency given the evaluation waiting time. This implies that concept drift plays a more important role than time to find defects when determining a good evaluation waiting time.

To double check if that is really the case, we have computed Spearman (nonlinear) and Pearson (linear) correlations to investigate the possible relationship between the best evaluation waiting time and the time to find defects in the evaluation data stream of each of the thirteen projects. The results did not show any high correlation. This confirms that knowledge of time to find defects does not really help in choosing the best evaluation waiting time. Therefore, concept drift is the primary aspect to be investigated when trying to tune the evaluation waiting time for a specific project. Monitoring the time to find defects is on its own not a good indicator of what waiting time to adopt.

8.5.3 Setting a Default Evaluation Waiting Time

Sections 8.5.1 and 8.5.2 suggest principles for choosing appropriate evaluation waiting time for a given project when information about the project is available from project managers and developers or when extensive numerical analyses can be conducted to tract potential concept drift of input features of JIT-SDP. However, when no further information about the project is available, one may wish to adopt an evaluation waiting time that is known to typically be the best one across a variety of projects.

Figure 7(n) shows the median impact of the evaluation waiting time on the validity across projects. We can see that a smaller evaluation waiting time generally corresponded to a better validity of the evaluation procedure, reflecting the trend that smaller evaluation waiting times were usually better for most projects. Therefore, even though 15 days is not always going to be the best evaluation waiting time, we suggest to use 15 days as the *default* because it is expected in general to lead to a good estimate of the true predictive performance up to the most recent evaluation time step.

9 Threats to Validity

Internal Validity. A potential threat is the fact that we may never gain access to true labels of some defect-inducing software changes if the defects induced by them are not found until the end of the data stream due to very large verification latency. To mitigate this threat, we have deliberately chosen open source projects covering a period of at least four years and eliminated software changes from the latter periods of the data stream, as described in Section 6.

Construct Validity. The evaluation metric (G-mean) used in the analyses has been carefully chosen for the characteristics of online JIT-SDP. In particular, this metric is adequate because it is insensitive to the level of class imbalance [32], being particularly important for problems that suffer from class imbalance evolution such as JIT-SDP [6]. We have also calculated this metric incrementally with fading factors as recommended for online learning studies [32], so that it is possible to track the fluctuations in predictive performance over time. The metric adopted in this study is the most widely used in online learning studies involving class imbalance [32]. Investigation of other metrics is left as future work. The continuous metrics to capture the amount of label noise and the validity of the performance evaluation procedure have also been designed based on a fading factor, being suitable for tracking variations over time [32].

External Validity. We have investigated 13 open source projects, with 4 levels of (training and evaluation) waiting time, in addition to 5 lengths for the data stream in RQ1 (see Section 8.1). These cover a range of different characteristics presented in previous JIT-SDP work. However, as with any study involving machine learning, results may not generalize to other projects, waiting times or data stream lengths. We have adopted `Commit Guru` as the data collection tool for this study. This is an easy-to-use tool that is readily available for practitioners. As with other data collection algorithms for JIT-SDP, this tool may itself result in some noise during the data collection process (see Section 3.4). Therefore, our conclusions regarding the impact of waiting time are valid when such tool is used, and may not generalize to other data collection tools that may inflict considerably different levels of noise. Also, we focus our study on OOB with Hoeffding trees, which have been previously adopted for online JIT-SDP [6], [7]. Other machine learning approaches could be investigated in the future.

10 Conclusion

We provided the first discussion on how to continuously evaluate predictive performance in JIT-SDP over time during the software development. We also provide the first analysis of the extent to which the validity of the continuous performance evaluation procedure can be affected by the inherent evaluation waiting time in JIT-SDP. Our key findings and their implications are as follows.

- Smaller evaluation waiting times were typically associated to considerably larger amount of label noise, whereas larger amount of evaluation label noise was associated to slightly worse validity of the performance evaluation procedure. In the absence of concept drift, this would mean that larger evaluation waiting times are recommended in practice to slightly improve such validity.
- Smaller evaluation waiting times were themselves associated to better validity of the performance evaluation procedure, probably due to concept drift. This means that, in general, the problems caused by label noise resulting from poor choices of waiting time are minor compared to the problems caused by concept drift. Even though the differences in the validity caused by different choices of waiting time were typically small (median of 3.73% difference), they were sometimes larger (up to 25%). Therefore, when choosing an evaluation waiting time in practice, special consideration should be taken to concept drift. If no information regarding concept drift is available in a given project or if we expect that concept drifts will happen often in the data stream of the project, we recommend smaller waiting times (e.g., 15 days), so that the evaluation is not performed with obsolete examples. If concept drifts are expected to be infrequent, larger waiting times should be preferred to prevent the slightly detrimental effect of label noise.
- Interestingly, verification latency was larger in latter portions of the data stream, suggesting that it takes longer to find defects induced by software changes as the projects mature. In the context of performance evaluation, this means that larger data streams (e.g., longer projects) are likely to result in more continuous label noise, which may in turn be more detrimental to the validity of the performance evaluation procedure.
- Despite leading to differences in estimated predictive performance, different evaluation waiting times usually did not change the rankings of the JIT-SDP models. Conclusions in terms of which models are ranked better are thus unlikely to change when using different evaluation waiting times, especially when investigating the ranking across projects. It is important to note that overlooking training waiting time is still a serious problem, as it leads to overoptimistic estimations of predictive performance [4].
- This paper is also the first to provide a mathematical formulation of the prediction, training and continuous evaluation procedures in online JIT-SDP. This formulation

should help future research studies and the development of tools to monitor the predictive performance of JIT-SDP in practice over time during the software development.

As future work, we propose to further investigate the use of concept drift detection methods for automatically tuning the evaluation waiting time for the JIT-SDP evaluation purposes. The impact of training waiting time on the quality of the resulting predictive models is also worthwhile to be studied. Future qualitative study to complement the findings that latter portions of data streams are typically associated to larger verification latency would also be desirable, to further investigate the reasons for this observation.

REFERENCES

[1] 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," *IEEE TSE*, vol. 39, no. 6, pp. 757–773, 2013.

[2] A. Mockus and D. M. Weiss, "Predicting risk of software change," *Bell Labs Technical Journal*, vol. 5, no. 2, pp. 169–180, 2000.

[3] E. Shihab, A. E. Hassan, B. Adams, and Z. M. Jiang, "An industrial study on the risk of software changes," in *FSE*, 2012, pp. 1–11.

[4] M. Tan, L. Tan, S. Dara, and C. Mayeux, "Online defect prediction for imbalance data," in *ICSE*, 2015, pp. 99–108.

[5] S. McIntosh and Y. Kamei, "Are fix-inducing changes a moving target? a longitudinal case study of just-in-time defect prediction," *IEEE TSE*, vol. 44, no. 5, pp. 412–428, 2018.

[6] G. Cabral, L. Minku, E. Shihab, and S. Mujahid, "Class imbalance evolution and verification latency in just-in-time software defect prediction," in *ICSE*, Monteal, Canada, 2019, pp. 666–676.

[7] S. Tabassum, L. L. Minku, D. Feng, G. G. Cabral, and L. Song, "An investigation of cross-project learning in online just-in-time software defect prediction," in *ICSE*, 2020, p. 554–565.

[8] G. Ditzler, M. Roveri, C. Alippi, and R. Polikar, "Learning in non-stationary environments: A survey," *IEEE CIM*, vol. 10, no. 4, pp. 12–25, 2015.

[9] S. Kim, J. E. J. Whitehead, and Y. Zhang, "Classifying software changes: Clean or buggy?" *IEEE TSE*, vol. 34, no. 2, pp. 181–196, 2008.

[10] J. Sliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" in *MSR*, 2005, pp. 1–5.

[11] J. Eyolfson, L. Tan, and P. Lam, "Do time of day and developer experience affect commit bugginess?" in *MSR*, 2011, pp. 153–162.

[12] A. T. Misirli, E. Shihab, and Y. Kamei, "Studying high impact fix-inducing changes," *EMSE)*, vol. 21, no. 2, pp. 605–641, 2016.

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

[14] T. Hoang, H. K. Dam, Y. Kamei, D. Lo, and N. Ubayashi, "DeepJIT: an end-to-end deep learning framework for just-in-time defect prediction," in *MSR*, 2019, pp. 34–45.

[15] T. Hoang, H. J. Kang, D. Lo, and J. Lawall, "CC2Vec: Distributed representations of code changes," in *ICSE*, 2020, pp. 518–529.

[16] H. M. Nguyen, E. W. Cooper, and K. Kamei, "Online learning from imbalanced data streams," in *International Conference of Soft Computing and Pattern Recognition*, 2011, pp. 347–352.

[17] N. Chawla, K. Bowyer, L. Hall, and W. Kegelmeyer, "SMOTE: Synthetic minority over-sampling technique," *JAIR*, vol. 6, pp. 321–357, 2002.

[18] T.-H. Chen, M. Nagappan, E. Shihab, and A. E. Hassan, "An empirical study of dormant bugs," in *MSR*, 2014, pp. 82–91.

[19] G. Antoniol, K. Ayari, M. D. Penta, F. Khomh, and Y. Gueheneuc, "Is it a bug or an enhancement? a text-based approach to classify change requests," in *CASCON*, 2008, pp. 23:304–23:318.

[20] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu, "Fair and balanced? bias in bug-fix datasets," in *FSE*, 2009, pp. 121–130.

[21] J. Aranda and G. Venolia, "The secret life of bugs: Going past the errors and omissions in software repositories," in *ICSE*, 2009, pp. 298–308.

[22] S. Kim, H. Zhang, R. Wu, and L. Gong, "Dealing with noise in defect prediction," in *ICSE*, 2011, pp. 481–490.

[23] K. Herzig, S. Just, and A. Zeller, "It is not a bug, it is a feature: How misclassification impacts bug prediction," in *ICSE*, 2013, pp. 392–401.

[24] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, A. Ihara, and K. Matsumoto, "The impact of mislabeling on the performance and interpretation of defect prediction models," in *ICSE*, vol. 1, 2015, pp. 812–823.

[25] S. Kim, T. Zimmermann, K. Pan, and E. James, "Automatic identification of bug-introducing changes," in *ASE*, 2006, pp. 81–90.

[26] 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 TSE*, vol. 43, no. 7, p. 641–657, 2017.

[27] E. C. Neto, D. A. da Costa, and U. Kulesza, "The impact of refactoring changes on the szz algorithm: An empirical study," in *SANER*, 2018, pp. 380–390.

[28] Y. Fan, X. Xia, D. Costa, D. Lo, A. Hassan, and S. Li, "The impact of mislabelled changes by szz on just-in-time defect prediction," *IEEE TSE*, vol. 47, no. 8, pp. 1559–1586, August 2021.

[29] J. Gama, R. Sebastiao, and P. P. Rodrigues, "On evaluating stream learning algorithms," *Journal of Machine Learning*, vol. 90, no. 3, pp. 317–346, 2013.

[30] J. Gama, R. Sebastião, and P. P. Rodrigues, "Issues in evaluation of stream learning algorithms," in *KDD*, 2009, pp. 329–338.

[31] S. Wang, L. L. Minku, and X. Yao, "Resampling-based ensemble methods for online class imbalance learning," *IEEE TKDE*, vol. 27, no. 5, pp. 1356–1368, 2015.

[32] ——, "A systematic study of online class imbalance learning with concept drift," *IEEE TNNLS*, vol. 29, no. 10, pp. 4802–4821, 2018.

[33] M. Kubat, R. Holte, and S. Matwin, "Learning when negative examples abound," in *ECML*, 1997, pp. 146–153.

[34] M. Kendall, "A new measure of rank correlation," *Biometrika*, vol. 30, no. 1-2, pp. 81–89, 1938.

[35] C. Rosen, B. Grawi, and E. Shihab, "Commit guru: analytics and risk prediction of software commits," in *FSE*, 2015, pp. 966–969.

[36] H. He and E. A. Garcia, "Learning from imbalanced data," *IEEE TKDE*, vol. 21, no. 9, pp. 1263–1284, 2009.

[37] J. Montiel, J. Read, A. Bifet, and T. Abdessalem, "Scikit-multiflow: A multi-output streaming framework," *JMLR*, vol. 19, no. 72, pp. 1–5, 2018.

[38] D. C. Montgomery, *Design and Analysis of Experiments*. Great Britain: John Wiley and Sons, 2004.

[39] E. C. Fieller and E. S. Pearson, "Tests for rank correlation coefficients: I," *Biometrika*, vol. 49, no. 1-2, pp. 185–191, 1962.

Liyan Song is a research assistant professor at Department of Computer Science and Engineering, Southern University of Science and Technology (China). Prior to that, she was a research fellow at School of Computer Science, University of Birmingham (UK). She received her Ph.D. degree in Computer Science from University of Birmingham (UK), and her bachelor and master degrees in Mathematics from Harbin Institute of Technology (China). Her main research interests are machine learning for predictive modeling in software engineering. She also has research interests in machine learning areas such as online learning, Bayesian learning, class imbalance learning and unsupervised learning.

Leandro L. Minku received the Ph.D. degree in computer science from the University of Birmingham, U.K., in 2010. He is currently an associate professor with the School of Computer Science, University of Birmingham. Prior to that, he was a lecturer of computer science with the University of Leicester, U.K., and a research fellow with the University of Birmingham. His research interests include machine learning for software engineering, data stream learning, class imbalance learning, ensembles of learning machines, and search-based software engineering. He is an associate editor-in-chief of the Neurocomputing journal, and an associate editor for IEEE Transactions on Neural Networks and Learning Systems, Empirical Software Engineering, and Journal of Systems and Software. He was the general chair for the 2019 and 2020 International Conference on Predictive Models and Data Analytics in Software Engineering, and co-chair for the Artifacts Evaluation Track of the 2020 International Conference on Software Engineering.