On the Validity of Retrospective Predictive Performance Evaluation Procedures in Just-In-Time Software Defect Prediction

Liyan Song · **Leandro L. Minku**[*] ·
Xin Yao[*]

Abstract Just-In-Time Software Defect Prediction (JIT-SDP) is concerned with predicting whether software changes are defect-inducing or clean. It operates in scenarios where labels of software changes arrive over time with delay, which in part corresponds to the time we wait to label software changes as clean (waiting time). However, clean labels decided based on waiting time may be different from the true labels of software changes, i.e., there may be label noise. This typically overlooked issue has recently been shown to affect the validity of continuous performance evaluation procedures used to monitor the predictive performance of JIT-SDP models during the software development process. It is still unknown whether this issue could potentially also affect evaluation procedures that rely on retrospective collection of software changes such as those adopted in JIT-SDP research studies, affecting the validity of the conclusions of a large body of existing work. We conduct the first investigation of the extent with which the choice of waiting time and its corresponding label noise would affect the validity of retrospective performance evaluation procedures. Based on 13 GitHub projects, we found that the choice of waiting time did not have a significant impact on the validity and that even small waiting times resulted in high validity. Therefore, (1) the estimated predictive performances in JIT-SDP studies are likely reliable in view of different waiting times, and (2) future studies can make use of not only larger (5k+ software changes),

Liyan Song (songly@sustech.edu.cn) · Xin Yao (xiny@sustech.edu.cn)
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.

Leandro L. Minku (L.L.Minku@bham.ac.uk)
School of Computer Science, the University of Birmingham, Edgbaston, Birmingham, B15 2TT, UK

[*] the corresponding authors.

but also smaller (1k software changes) projects for evaluating performance of JIT-SDP models.

Keywords Just-in-time software defect prediction · performance evaluation procedures · verification latency · online learning · concept drift · label noise

1 Introduction

Just-In-Time Software Defect Prediction (JIT-SDP) is concerned with predicting whether software changes are defect-inducing or clean upon commit time (i.e., just-in-time) based on machine learning approaches (Kamei et al. 2013). In practice, JIT-SDP operates in an online learning scenario, where software changes are produced and labeled over time for the purpose of training and evaluating JIT-SDP models. In particular, each software change must be predicted as defect-inducing or clean at commit time. Then, only when the label (defect-inducing or clean) becomes available, this software change can be used as a data example to evaluate and update (train) JIT-SDP models.

It takes time for the true labels of software changes to be revealed in the real-world process of JIT-SDP (Cabral et al. 2019; Ditzler et al. 2015; Song and Minku 2023). As a result, examples need to be produced based on *observed* labels rather than the true labels of software changes. Specifically, a software change is labeled to produce a defect-inducing example when a defect is found to be induced by it; in contrast, it is labeled as clean when no defect has yet been found to be induced by it and enough time has passed for one to be confident that this software change is really clean. Such length of time is referred to as *waiting time* (Cabral et al. 2019; Song and Minku 2023) and can be considered as a pre-defined parameter W of the data collection process. The observed clean label resulting from such waiting time may or may not be the same as the true label of this software change. Whenever it is not the same, a noisy example is produced. Such label noise caused by the waiting time may affect not only the training of JIT-SDP models, but also the validity of procedures used to evaluate them.

Song and Minku (2023) discussed how to evaluate predictive performance continuously over time during the software development process. The purpose of the *continuous performance evaluation procedure* is to track the most recent performance status of JIT-SDP models during the software development. Therefore, in this evaluation procedure, each software change is used to update the predictive performance as soon as it can be labeled, given a waiting time W. This is necessary in practice because the predictive performance of JIT-SDP models may fluctuate over time as a result of changes in the underlying defect generating process (Cabral and Minku 2022 (in press); Cabral et al. 2019; McIntosh and Kamei 2018; Tabassum et al. 2020) and it is important for practitioners to be alerted of any performance deterioration as early as possible. The study found that waiting time had a significant impact on the validity of such kind of evaluation procedure. In particular, if inappropriate waiting times are used, the results of the evaluation procedure become invalid.

Another kind of evaluation procedure is the *retrospective performance evaluation procedure*, where software changes are collected and labeled retrospectively rather than continuously over time. The purpose of this evaluation procedure is to check how well JIT-SDP models would have performed in practice if they had been predicting (and potentially learning) those labeled software changes over time. Such procedure can be used to help practitioners to decide which kind of JIT-SDP approach to adopt in their company, rather than for the purpose of monitoring the performance of a currently adopted JIT-SDP model during the software development process. For instance, research papers typically collect and label software changes to retrospectively evaluate how well different JIT-SDP approaches would have performed on those past software changes, rather than monitoring the predictive performance of such models on software changes that are currently being developed in a project. The results of such evaluation procedure are used to determine which kind of JIT-SDP approach is more promising to be adopted in practice. Once adopted in practice, the corresponding JIT-SDP model should then have its predictive performance monitored continuously over time based on continuous evaluation procedures such as the one proposed in Song and Minku (2023), to alert software engineers if/when its performance start deteriorating.

Retrospective performance evaluation procedures do not need to collect the label of a software change as soon as possible after this software change is committed. Instead, all labels can be collected at the same time moment when one decides to trigger this evaluation procedure. Such labeling process also relies on a waiting time parameter. However, this waiting time refers to the *minimum* amount of time we wait to label a software change as being clean, rather than the *exact* amount of time used in continuous evaluation procedures. In other words, it corresponds to the age of the newest software change that can be labeled as clean to produce an example. All other clean labeled examples are produced with older software changes. The older the software change, the more time we will have waited to observe its clean label, potentially leading to a more reliable label. Due to these differences between the waiting time used in continuous and retrospective performance evaluation procedures, it is unknown whether the validity issues found to affect continuous performance evaluation procedures (Song and Minku 2023) also affect retrospective performance evaluation procedures.

If the impact of waiting time on the validity of retrospective performance evaluation procedures is significant, it could seriously affect the validity of a large number of existing research studies in JIT-SDP, especially considering that many of them implicitly assume that label noise is non-existent for evaluation purposes. If such impact is not significant, it would mean that the predictive performances obtained in existing research studies are likely reliable in view of different choices of waiting time, and could potentially be used to inform practitioners about which kind of JIT-SDP approach is more promising to adopt in practice.

Therefore, the aim of this paper is to systematically investigate whether and to what extent the conclusions of JIT-SDP retrospective performance

evaluation procedures (and thus also the conclusions of a large body of JIT-SDP research studies) are (in)valid in view of the fact that observed labels rather than the true labels of software changes are being used for performance evaluation. This would not only lead to an insight into the validity of the conclusions drawn in existing work that overlooks the role of waiting time on evaluation procedures in JIT-SDP, but also inform future JIT-SDP work on how waiting time should be considered for evaluation purposes.

This study can be seen as a conceptual replication of Song and Minku (2023) aiming to check whether the findings obtained for continuous evaluation scenarios would also occur in retrospective evaluation scenarios. For this, some adjustments need to be done in the methodology that was kept as similar as possible to that of Song and Minku (2023). The datasets investigated in this work are also the same as Song and Minku (2023), but their processing also had to be adjusted for the retrospective evaluation scenario. We answer three of the Research Questions (RQs) from Song and Minku (2023), but in the context of retrospective performance evaluation procedures rather than in continuous performance evaluation procedures[1]:

[RQ1] How large is the amount of label noise caused by different waiting times in retrospective JIT-SDP data collection? The effect of waiting time on label noise may be reduced when evaluating JIT-SDP through retrospective evaluation procedures compared to the continuous evaluation procedure required during the software development process. This is because the waiting time is only used to determine what is the most recent software change that can be used in the retrospective performance evaluation procedure. All other changes will be older than this one, such that more time would have passed to detect their true labels, potentially reducing the amount of label noise. However, it is unknown how large the amount of label noise caused by different waiting times in retrospective data collection is.

[RQ2] To what extent is the validity of retrospective performance evaluation procedures impacted by label noise resulting from waiting time? The label noise resulting from waiting time investigated in RQ1 may or may not be large enough to have a significant impact on the validity of retrospective performance evaluation procedures. This investigation will enable us to check how reliable the estimated performance of current JIT-SDP studies is in view of the label noise caused by waiting time. In other words, it will determine whether conclusions in terms of how well different JIT-SDP approaches perform (and thus which ones are recommended for adoption in practice) are reliable in view of the label noise caused by waiting time.

[RQ3] To what extent is the validity of retrospective performance evaluation procedures impacted by different waiting times? As in Song and Minku (2023), part of RQ3 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 retrospective

---

[1] Previous work (Song and Minku 2023) investigated 4 research questions, but the fourth one becomes irrelevant in the context of our work given the results obtained for the first three research questions.

performance evaluation procedures (RQ2), waiting time may have significant impact on the validity through the label noise it generates. However, waiting time could potentially have further impact on the validity of retrospective performance evaluation procedures that cannot be captured by label noise on its own, possibly intensifying or moderating the impact mediated by label noise. RQ3 complements the study to check whether the choice of waiting time as a while does have an impact on the validity.

To answer these RQs, we conduct experimental studies based on the same 13 GitHub software projects and statistical methodologies in Song and Minku (2023). We find that different waiting times used in retrospective performance evaluation procedures can cause significantly different amounts of label noise (RQ1). Similar to Song and Minku (2023)'s results in the continuous evaluation scenario, we find that such amounts of label noise also have a statistically significant impact on the validity of retrospective performance evaluation procedures in JIT-SDP (RQ2). However, the differences between estimated and true predictive performance are smaller than those found in Song and Minku (2023), being always smaller than 3% and having a median of less than 1% across datasets. Different from Song and Minku (2023)'s results on the continuous evaluation scenario, when investigating the direct impact of waiting time on the validity (RQ3), we found that such impact is moderated and becomes insignificant for retrospective performance evaluation procedures. Therefore, different waiting times are unlikely to change the conclusions on whether JIT-SDP is accurate enough to be worthy of adoption in practice, especially when conducing studies using multiple datasets.

Our results also show that even waiting times as small as 15 days led to high validity of retrospective performance evaluation procedures. This is very encouraging, as it means that research studies can evaluate JIT-SDP models not only on large projects (with more than 5k software changes) but also with smaller projects (with 1k software changes). Accordingly, it is not necessary to remove a large portion of the most recent software changes to increase the validity of retrospective performance evaluation procedures. This result is particularly relevant given that many software companies develop projects that are much shorter in length than many of the existing open source projects that have been running for many years.

The remainder of this paper is organized as follows. Section 2 motivates and briefly explains two evaluation scenarios of JIT-SDP – a continuous evaluation scenario as in Song and Minku (2023) and a retrospective evaluation scenario, which will be investigated in this paper. Section 3 discusses background and related work. Section 4 explains our notation system and formulates the validity of the retrospective performance evaluation procedures. Section 5 describes the design of our experiments, and our RQs are answered in Section 6 by analyzing the results of experiments. Threats to validity is discussed Section 7 and Section 8 concludes the paper.
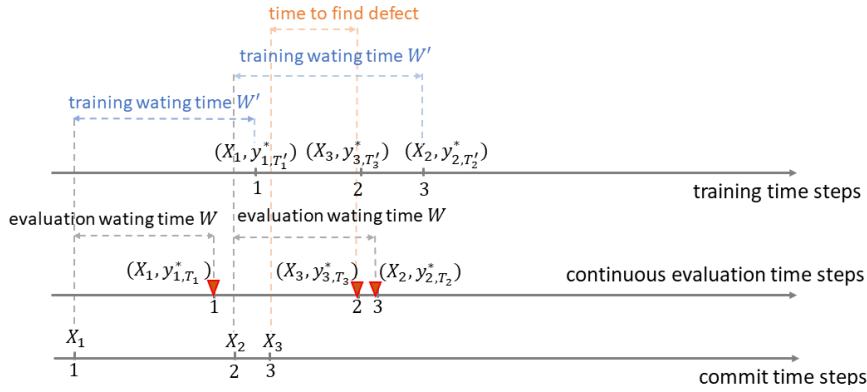
Fig. 1 Illustration of labeled examples used for training and the continuous performance evaluation investigated in Song and Minku (2023). Examples (X_u, y^*_{u,T_u}) used in the continuous performance evaluation scenario have their labels collected individually at Unix timestamp T_u, each of which would be W days after the commit time or when a defect is found to be associated to it. Training examples (X_u, y^*_{u,T'_u}) have their labels individually collected at Unix timestamp $T_{u'}$. This timestamp corresponds to the time when X_u is labeled for training purposes. Especially, evaluation waiting time W should be no larger than training waiting time W', guaranteeing the online learning principle that each example should be first used for evaluation and then used for training (Gama et al. 2013).

2 Continuous and Retrospective Performance Evaluation Scenarios

This section provides motivating scenarios for using continuous and retrospective performance evaluation procedures and discusses the general differences between these two kinds of procedure. Figures 1 and 2 give illustrative examples of the continuous and the retrospective performance evaluation scenarios to facilitate such distinctions. As will be explained in Sections 2.1 and 2.2, the different purposes of these two evaluation procedures result in the labels of their evaluation examples to be collected at different moments in time, potentially resulting in different levels of label noise and validity issues. A more detailed mathematical formulation of the continuous and retrospective predictive performance evaluation procedures can be found in Song and Minku (2023) and in Section 4, respectively.

2.1 Continuous Performance Evaluation Scenario

As a software project is developed, new software changes are produced over time. JIT-SDP models are used to predict these incoming software changes as being clean or defect-inducting. However, it has been shown that variations in the underlying data-generating process (i.e., concept drifts (Ditzler et al. 2015)) can cause the predictive performance of JIT-SDP models to fluctuate over time (Cabral and Minku 2022 (in press); Cabral et al. 2019; McIntosh and Kamei 2018; Tabassum et al. 2020). Therefore, it is important for practitioners to *continuously* monitor such predictive performance over time *during* the

software development process. Continuous monitoring enables practitioners to identify any time periods when the JIT-SDP model becomes unreliable and should not be trusted.

To continuously evaluate the predictive performance of a given JIT-SDP model during the software development process, as soon as a past software change becomes labeled, it should be immediately used to update the estimate of the *current* predictive performance of the JIT-SDP model. Therefore, the evaluation process is triggered at several different Unix timestamps T_u during software development, each of which corresponding to the moment when a software change is labeled as clean or defect-inducing. The middle timeline of Figure 1 gives an illustrative example of such continuous labeling process. Examples (X_u, y^*_{u,T_u}) are produced individually and continuously over time throughout the evaluation process. In particular, each software change X_u is labeled at Unix timestamp T_u (annotated by the inverted red triangles on the timeline), leading to an observed label y^*_{u,T_u}. This timestamp is either W days after the commit time of X_u (e.g., (X_1, y^*_{1,T_1}) and (X_2, y^*_{2,T_2}) in the figure), or when a defect is found to be associated to it (e.g., (X_3, y^*_{3,T_3}) in the figure). As such, commit and evaluation time steps may differ in the continuous performance evaluation scenario. For instance, (X_3, y^*_{3,T_3}) comes prior to (X_2, y^*_{2,T_2}) in this illustrative example because a defect was found to be induced by X_3 at an earlier moment in time.

The value chosen for the waiting time parameter W affects both the amount of label noise generated when producing labeled examples and the obsolescence of the examples used to estimate the current predictive performance of a model. In particular, a too long waiting time means that the example is already old by the time it is used to update the estimation of the current predictive performance, as it may not reflect well the current defect generating process anymore. Waiting too little means that not enough time is used to find defects that are potentially associated to the software change, leading to more label noise (examples labeled as clean when they are actually defect-inducing). Song and Minku (2023) found that the choice of waiting time has a significant impact on the validity of continuous performance evaluation procedures, and that smaller waiting times should typically be preferred to obtain a better validity of such evaluation procedure.

2.2 Retrospective Performance Evaluation Scenario

Many machine learning algorithms can potentially be used for creating JIT-SDP models. Consider a practitioner who does not yet use JIT-SDP in their software company, but is interested in starting to use. This practitioner needs to decide which algorithm to adopt to build a JIT-SDP model for their company. For that, they can potentially use historical software changes that have already been produced in their software projects to investigate which machine learning algorithm would likely lead to the best predictive performance in the context of their software development company. These historical software
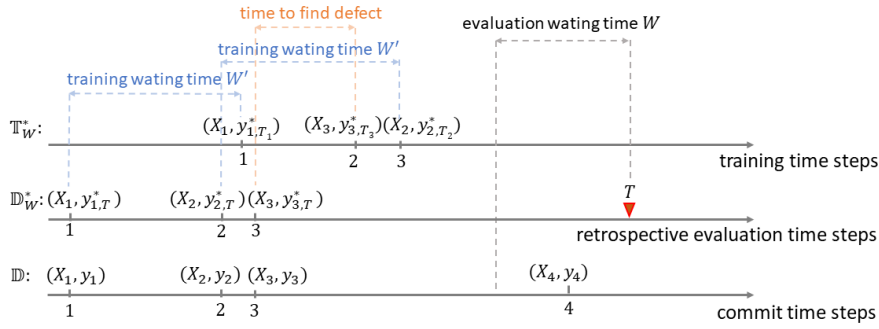
Fig. 2 Illustration of data streams $\mathbb{D}$, $\mathbb{D}_W^*$ and $\mathbb{T}_{W'}^*$ in the retrospective performance evaluation scenario. Unix timestamp T corresponds to the time of the retrospective data collection, leading to the labels $y_{u,T}^*$ used for the retrospective evaluation data stream. Examples $(X_u, y_{u,T}^*)$ used for evaluation can only be collected up to Unix time stamp $T - W$, where W is the waiting time used for the retrospective data collection. Examples (X_u, y_{u,T_u}^*) used for training have their labels collected at Unix timestamp T_u. This timestamp corresponds to the time when a software change X_u is labeled for training purposes (W' days after the commit or when a defect is found, where W' is the training waiting time).

changes can be collected retrospectively and used to evaluate the predictive performance that a JIT-SDP model would have achieved if it had been used for predicting those changes when they were produced. Similarly, a researcher who is interested in evaluating and comparing several machine learning algorithms for JIT-SDP would also retrospectively collect software changes that have already been produced in a software project. Therefore, at the moment of the retrospective data collection, labels of software changes can be assigned based on the most up-to-date knowledge about defects in the software. In other words, the practitioner could simulate the exact online learning process that would have been used to create and update predictive models in practice based on historical data, but evaluate these predictive models using the most current knowledge about defects on such data.

In particular, if the moment when we run the evaluation procedure is depicted by Unix timestamp T, all labels will have been collected with the most up-to-date knowledge available at T. The middle timeline of Figure 2 gives an illustrative example of the retrospective labeling process. All examples $(X_u, y_{u,T}^*)$ have their labels collected at Unix timestamp T (annotated by an inverted red triangle). Each clean labeled example was committed at some timestamp up to Unix timestamp $T - W$. Even though one clean labeled example can possibly correspond to a software change committed exactly at Unix timestamp $T - W$, all other clean labeled examples correspond to software changes committed at Unix timestamps $U < T - W$. As more than W days would have passed since the commit of the software changes that are being labeled as clean, such labels used for retrospective evaluation procedures may be less noisy than the labels used for continuous evaluation procedures.

It is also worth noting that commit and evaluation time steps are equivalent in the retrospective performance evaluation scenario. The order with which

examples are used for evaluation is the same as the order with which they were committed. This is because the model that would have been available at Unix timestamp T_u can be evaluated with the software change that was committed exactly at Unix timestamp T_u. This is different from the continuous evaluation procedure, where a model available at time T_u needs to be evaluated with older software changes (produced earlier than T_u).

Due to such differences, the impact of waiting time on retrospective evaluation procedures may be different from that on continuous performance evaluation procedures. Our work investigates what this impact is.

> **Summary of Differences Between Continuous and Retrospective Evaluation Procedures**
>
> In continuous predictive performance evaluation procedures, we use the most recently labeled past software change to immediately update the estimation of the current predictive performance of a JIT-SDP model during the software development process. At the time of the estimation, a clean software change is exactly W days (waiting time) old. In retrospective predictive performance evaluation procedures, we use historical software changes to estimate the predictive performance that a JIT-SDP model would have achieved in past moments in time. At the time of the estimation, a clean software change may be more than W days old, meaning that more time is used to collect its label.
>
> By knowing the predictive performance that JIT-SDP approaches would have obtained through a retrospective performance evaluation procedure, a practitioner could make a decision on which kind of JIT-SDP models to start adopting in their company. Once they start adopting a model, they should monitor its predictive performance based on a continuous performance evaluation procedure to detect any potential performance deterioration over time.

3 Related Work

3.1 Software Defect Prediction

Software defect prediction (SDP) can be used to identify code modules that are likely to be defective (i.e., defect-prone modules). Based on that, quality assurance teams can effectively focus their limited resources on testing, reviewing or debugging such defect-prone modules, reducing the time required to find defects. Most existing work has investigated SDP models that are created using a fixed and pre-existing dataset (Hassan 2009; Nam and Kim 2015; Wang and Yao 2013), but some studies have also investigated the effect of learning additional data received over time (Ekanayake et al. 2012; Harman et al. 2014; Kabir et al. 2019).

However, conventional SDP usually predicts bugs at the module level. Such coarse granularity can cause disadvantages as it may be difficult for practitioners to find where exactly the defect is. Such disadvantage can be alleviated by defect prediction in a finer granularity, such as software defect prediction at the software change level (Hassan 2009).

3.2 Just-In-Time Software Defect Prediction (JIT-SDP)

Software defect prediction at the change level, a.k.a., Just-in-Time SDP (JIT-SDP), aims to predict whether software changes are likely to induce defects (Mockus and Weiss 2000; Shihab et al. 2012). JIT-SDP can be considered as a binary classification task, where a JIT-SDP model is constructed based on training examples of software changes that are labeled as *defect-inducing* or *clean*. Several studies have investigated different input features that can be used to describe software changes (Eyolfson et al. 2011; Kim et al. 2008; Misirli et al. 2016; Shihab et al. 2012; Śliwerski et al. 2005). Kamei et al. (2013) conducted a large-scale empirical study to investigate 14 features extracted from commits and bug reports for JIT-SDP models, which can be grouped into five dimensions of diffusion, size, purpose, history and experience. They showed these features to be good indicators for achieving high predictive performance. Many subsequent studies have been conducted based on these features (Cabral et al. 2019; Kamei et al. 2016; McIntosh and Kamei 2018; Misirli et al. 2016).

Different machine learning algorithms have been used to build JIT-SDP models, e.g., logistic regression (Kamei et al. 2013; McIntosh and Kamei 2018), random forests (Kamei et al. 2016), support vector machines (Kim et al. 2008), and deep learning (Hoang et al. 2019, 2020). Tree-based and logistic regression-based methods are among the most popular and have shown potential in yielding good performance for JIT-SDP. Some studies (Kamei et al. 2013, 2016; Tan et al. 2015) have adopted techniques such as random under-sampling, random over-sampling (Nguyen et al. 2011) and SMOTE (Chawla et al. 2002) to help JIT-SDP in identifying defect-inducing software changes despite the relatively small number of defect-inducing training examples compared to the number of clean ones.

Most studies overlooked the chronology of software changes, where software changes arrive sequentially in order over time. Overlooking the chronology was shown to lead to defect prediction models with deceptively better predictive performance than they could achieve in practice when chronology must be respected (Tan et al. 2015). Other studies have also reported that predictive performance of JIT-SDP models can deteriorate over time (McIntosh and Kamei 2018), possibly as a result of concept drift. Therefore, online learning algorithms able to update JIT-SDP models with new examples over time have been recommended (Cabral et al. 2019; Tabassum et al. 2020; Tan et al. 2015).

3.3 Verification Latency in SDP

When respecting the chronology of JIT-SDP, one needs to take into account the fact that labels of software changes only become available long after the software changes are committed, an issue referred to as *verification latency* in machine learning (Ditzler et al. 2015). As the bug-fix software change or bug issue report required to identify a bug-introducing software change (Śliwerski et al. 2005) comes after this software change, it by nature takes time for the true label of a software change to be revealed. Therefore, the labels of software changes are typically revealed months or even years after their commit time (Cabral et al. 2019). Consequently, people would have to wait enough time to confidently label a given software change as clean.

A similar study in module-based SDP is discussed by (Chen et al. 2014). They use the term *dormant bugs* to refer to defects introduced in a version of the software system that are found only in later versions. Based on 20 open-source software projects from Apache Software Foundation, they found that typically 33% of the defects introduced in a version were reported in future versions as dormant bugs and performance evaluation that ignored 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 indicates that defects induced by software changes may take time to be revealed. Later, based on 10 open-source software projects, Cabral et al. (2019) found in the context of JIT-SDP that the time it took to find defects induced by software changes varied from 1 to 4,210 days after their commit time, with a median of 90 days. Pornprasit and Tantithamthavorn (2021) corroborated this by also mentioning that the true labels of defect-inducing software changes can only be collected with a delay.

However, most existing JIT-SDP studies overlook the fact that such label delay happens not only in the training process of JIT-SDP but also in the evaluation process. Many studies implicitly assume that true labels of software changes are available in their retrospective performance evaluation procedures. So far, there have been very few related studies that proposed to delete a good portion of the end of a data stream of software changes (e.g., those committed in the last three months of the developing process) (Cabral et al. 2019; McIntosh and Kamei 2018; Tabassum et al. 2020; Tan et al. 2015) in an attempt to reduce label noise. However, no one really knows how large this portion should be in order to avoid problems in the validity of retrospective performance evaluation procedures. Therefore, it is desirable to know to what extent different waiting times and their resulting label noise would impact the validity of retrospective performance evaluation procedures.

A previous study (Song and Minku 2023) has investigated whether and to what extent waiting time can affect the validity of continuous performance procedures of JIT-SDP models. Our study can be considered as a conceptual replication of Song and Minku (2023) to check whether the findings obtained for continuous performance evaluation scenarios would also occur in retrospective performance evaluation scenarios.

The smaller the waiting time, the more label noise is likely to be produced, as there would be less time to find defects associated to software changes before producing corresponding examples. So far, no existing study has investigated how to choose an adequate waiting time value automatically, while making remaining software changes not to be too obsolescent for the purposes of model training and evaluation. The amount of label noise produced as a result of different waiting times is also unknown, as well as the impact that waiting time may have on the validity of retrospective performance evaluation procedures in JIT-SDP. This work is the first to provide such analyses.

3.4 Label Noise in SDP

As explained in Section 1, this paper is related to the label noise that results from waiting time (Song and Minku 2023). Some studies have investigated label noise resulting from other aspects of the data collection process that are unrelated to the waiting time (Antoniol et al. 2008; Aranda and Venolia 2009; Bird et al. 2009; Kim et al. 2011). They found that issue reports can often be mislabeled. For instance, reports describing defects can be mislabeled as "enhancements" and such mislabeling can influence the issue tracking system and version control system records based on which source code modules are labeled as defective or clean. Some studies reported that such mislabeling can lead to a negative impact on predictive performance (Herzig et al. 2013; Yatish et al. 2019), while others concluded that this rarely causes a severe problem to SDP (Tantithamthavorn et al. 2015).

Yatish et al. (2019) investigated the label noise arising from post-release window periods through a case study with 32 releases and found that such post-release label noise had large impact on the model-based SDP performance, leading to misleading predictive accuracy of many studies that were based on such labeling of post-release window periods.

Related literature in JIT-SDP usually focused on the mislabeling resulting from the original SZZ algorithm (Śliwerski et al. 2005) and its variants (Da Costa et al. 2017; Kim et al. 2006; Neto et al. 2018), which were designed for the identification of defect-inducing software changes. Śliwerski et al. (2005) proposed B-SZZ (Basic SZZ) based on the built-in `annotation` command from the version control system. Kim et al. (2006) replaced the `annotate` command used in B-SZZ with the *annotation graph* – a tool for tracing change history, forming the variant AG-SZZ (Annotation Graph SZZ). To mitigate the noise caused by branch / merge changes and property changes of AG-SZZ, Da Costa et al. (2017) proposed the variant MA-SZZ (Meta-change Aware SZZ). Built upon MA-SZZ, Neto et al. (2018) proposed RA-SZZ (Refactoring Aware SZZ) to further deal with the refactoring modifications in software changes.

These algorithms are typically used in the literature for the collection of datasets for training and evaluating JIT-SDP models. Herbold et al. (2022) found that among studies that adopted SZZ algorithms, B-SZZ is the most popular variant with 38% identified literature; only 14% of the literature spec-

ified their adoption of other SZZ variants; the remaining 40% literature did not specify which SZZ algorithms they have adopted.

Fan et al. (2019) investigated the impact of label noise arising from the SZZ labeling process on predictive performance of JIT-SDP models based on four popular SZZ variants. RA-SZZ was the most recent SZZ algorithm and thus JIT-SDP models that were trained on examples labeled by RA-SZZ was used as the baseline. They found that the SZZ-related label noise caused by AG-SZZ can cause a significant performance reduction; in contrast, label noise caused by B-SZZ and MA-SZZ were unlikely to cause a severe problem to JIT-SDP. A more recent study was conducted by Herbold et al. (2022) to investigate the severe problem of the data labeling process using SZZ based on a dataset that was constructed under manual inspection. They concluded that a large amount of noise was produced by SZZ when labeling software changes, and manual work was recommended to inspect the quality of the data.

However, these studies focused on the impact of label noise on predictive performance of module-based SDP or JIT-SDP models, rather than on the validity of performance evaluation procedures as will be done in this work. Moreover, label noise caused by waiting time was not investigated in these studies, which is the main focus of this work. The only previous work investigating the impact of label noise resulting from waiting time on the validity of performance evaluation procedures was Song and Minku (2023), but it did not consider retrospective evaluation procedures. Hereafter, whenever we refer to label noise, we mean the label noise associated to waiting time, unless otherwise specified.

4 Problem Formulation

This section mathematically formulates the retrospective performance evaluation procedures for JIT-SDP. Notations used in the formulation are summarized in Table 1, and we invite readers to refer to this table for the explanations.

4.1 True and Observed Evaluation Data Streams

Given a Unix timestamp T, it would be ideal to evaluate the predictive performance of JIT-SDP based on the *predicted* and *true* (not observed) labels of the evaluation examples up to this Unix timestamp. The true evaluation data stream consists of all software changes with their true labels until T, where T is the Unix timestamp when the retrospective data collection is conducted. These software changes are ordered based on their commit Unix timestamp. We refer to the natural number representing the order of commit as an *evaluation* or *commit time step* [2].

---

[2] Song and Minku (2023) made a distinction between commit and evaluation time steps. However, commit and evaluation time steps are equivalent in this paper.

Table 1 Summary of notations used in the problem formulation of the validity of retrospective performance evaluation procedures in JIT-SDP.

Notation	Description
T, U	uppercase letters used to represent Unix timestamps
t, u	lowercase letters used to represent time steps corresponding to the Unix timestamps above, respectively*
X_u	the input features describing the software change produced at commit (also evaluation) time step u
y_u	the true label of software change X_u, being 1 for a defect-inducing software change and 0 for a clean software change
$y_{u,t}^*$ / $y_{u,T}^*$	the observed label for X_u decided at time step t / Unix timestamp T
$\widehat{y}_u$	a predicted label of X_u at evaluation time step t
$\mathbb{D}$	the true evaluation data stream that is produced based on true labels
$\mathbb{D}_W^*$	the observed evaluation data stream that is collected based on observed labels retrospectively given waiting time W
E	the true predictive performance based on the true evaluation data stream
E_W^*	the estimated predictive performance based on the observed evaluation data stream $\mathbb{E}_W^*$
Δ	the validity of retrospective performance evaluation procedures
η_W	the amount of label noise of the observed evaluation data stream given a waiting time W in the retrospective performance evaluation scenario

* Note that in the formulations used in this paper, we will frequently need to convert between time steps and actual Unix timestamps. Uppercase letters such as T are used to indicate Unix timestamps, and corresponding lowercase letters such as t are used to indicate the corresponding time steps in a data stream. Lowercase and uppercase letters will be used interchangeably whenever we need to emphasize time steps or Unix timestamps.

The true evaluation data stream can be formulated as

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

where the mathematical bold $\mathbb{D}$ denotes a data stream used for evaluation, X_u denotes the features describing the software change produced at commit time step u, y_u denotes the true label of X_u (1 for defect-inducing and 0 for clean), and t denotes the time step corresponding to the Unix timestamp T [3].

However, the true labels y_u are unknown when collecting the data stream. In reality, given a waiting time W, the data stream is collected based on observed labels, which can be formulated as

$$\mathbb{D}_W^* = \{(X_u, y_{u,T}^*)\}_{u=1}^{t_W} \cup \{(X_u, 1)\}_{u=t_W+1}^t,$$

where the superscript $*$ is used to indicate that this is a data stream produced with observed (rather than the true) labels of software changes, $y_{u,T}^*$ denotes an observed label for X_u decided at Unix timestamp T, and t_W is the time step corresponding to the Unix timestamp $T_W = T - W$. The time step t_W corresponds to the number of evaluation time steps of this data stream up to the last moment when a clean software change can be labeled given waiting

---

[3] Lowercase and uppercase letters will be used interchangeably in our notation system to represent a time step and its corresponding Unix timestamp, respectively, as explained in the note of Table 1.

time W. No clean software change can be labeled after T_W because we would not have waited enough time to be confident that this software change is really clean. Examples $\{(X_u, 1)\}^t_{u=t_W+1}$ correspond to software changes found to be defect-inducing between Unix timestamps T_W and T.

The two bottom timelines of Figure 2 give an illustrative example of observed and true evaluation data streams for a software project. The observed evaluation data stream $\mathbb{D}^*_W$ has software changes X_u whose labels have been collected as $y^*_{u,T}$, whereas the true evaluation data stream $\mathbb{D}$ has software changes X_u with their true labels y_u. $\mathbb{D}^*_W$ has only three examples in this illustration, whereas $\mathbb{D}$ has four. This is because labels for $\mathbb{D}^*_W$ can only be collected until $T_W = T - W$ (except for software changes found to be defect-inducing within the waiting time), whereas X_4 is a clean change that was committed after T_W. $\mathbb{D}$ includes the additional X_4 committed after T_W and its true label y_4. This is because $\mathbb{D}$ is an ideal (though impractical) data stream that does not require to wait for labeling the examples. It is worth noting that the observed evaluation data stream D^*_W used for retrospective evaluation is different from the one used in continuous performance evaluation procedures (Song and Minku 2023). In continuous performance evaluation, the observed label for a given software change X_u would be $y^*_{u,U+W}$ rather than $y^*_{u,T}$.

When all defect-inducing software changes are found before Unix timestamp T_W ($\{(X_u, 1)\}^t_{u=t_W+1} = \emptyset$), time step t_W is actually the last evaluation time step in the observed data stream. When there are defect-inducing software changes found between T_W and T ($\{(X_u, 1)\}^t_{u=t_W+1} \neq \emptyset$), the last time step where a defect-inducing software change is found is the last evaluation time step in $\mathbb{D}^*_W$. Without loss of generality and for the sake of simplicity, we will assume that t_W is the last evaluation time step of the observed evaluation data stream when writing formulas in the remaining of this section. For this reason, we can also use t_W to denote the number of evaluation time steps used by the retrospective performance evaluation procedure.

4.2 Computing Label Noise

For RQ1, we need to determine the impact of waiting time on the level of label noise. We formulate the amount of label noise of the evaluation data stream $\mathbb{D}^*_W$ given a waiting time W as

$$\eta_W = \frac{\sum_{u=1}^{t_W} |y^*_{u,T} - y_u|}{\sum_{u=1}^{t_W} y_u}, \tag{1}$$

where T is the Unix timestamp when the observed evaluation data stream was retrospectively collected and time step t_W corresponds to the Unix timestamp $T_W = T - W$, denoting the number of evaluation time steps (see the last paragraph of Section 4.1). Please refer to Table 1 for other notations.

As the investigated label noise in JIT-SDP can only happen to defect-inducing software changes, the numerator of Eq. (1) is not influenced by the

status of examples that are truly clean. As a clean software change is labeled "0", the denominator actually counts the total number of defect-inducing software changes in the data stream. Therefore, the label noise $\eta(W)$ measures the proportion of noisy examples over the total number of defect-inducing software changes. A large $\eta(W)$ shows a more severe level of label noise induced by waiting time in the retrospective data collection process. We can also see that examples at the tail (end) of the evaluation data stream are more likely to suffer from label noise because they are closer to the last Unix timestamp T, so that there is less time for defects induced by them to be found.

4.3 True and Estimated Performance

In retrospective evaluation, one is typically interested in determining how well JIT-SDP performs on each and every software change that has been labeled up to a given Unix timestamp T, where T represents the moment of data collection. The true predictive performance can be computed based on all examples in the true evaluation data stream $\mathbb{D}$. We define it as the average predictive performance across all examples produced up to time step t based on the true labels of the evaluation examples as

$$E = \frac{1}{t} \sum_{u=1}^{t} ||\widehat{y}_u - y_u||_G, \tag{2}$$

where t is the corresponding time step of T that also represents the number of evaluation time steps and $||\cdot||_G$ represents some performance metric.

However, as explained previously, the true performance is not accessible in reality as $\mathbb{D}$ is actually absent due to verification latency. One has to wait a certain amount of time to produce observed labels of software changes and estimate the predictive performance accordingly. In a retrospective performance evaluation procedure, the estimated performance that can be computed in reality is formulated based on the observed labels of software changes in the evaluation data stream $\mathbb{D}_W^*$ as

$$E_W^* = \frac{1}{t_W} \sum_{u=1}^{t_W} ||\widehat{y}_u - y_{u,T}^*||_G, \tag{3}$$

where the superscript $*$ is used to indicate that this is the estimated performance based on observed (not the true) labels and t_W is the time step corresponding to the Unix timestamp $T_W = T - W$ that also denotes the number of evaluation time steps of this data stream.

4.4 Validity of Retrospective Performance Evaluation Procedures

For RQ2 and RQ3, we need to determine the impact of label noise and waiting time on the validity of retrospective performance evaluation procedures,

respectively. The validity at Unix timestamp T given a waiting time W (and an amount of label noise) can be measured based on the difference between the true and estimated predictive performance, which can be formulated as

$$\Delta = 1 - |E - E_W^*|, \tag{4}$$

where E and E_W^* denote the true performance and the performance estimated retrospectively as defined in Eqs. (2) and (3), respectively. A larger value for Δ indicates a better validity of retrospective performance evaluation procedures.

4.5 Training Process

The retrospective evaluation procedure described in Sections 4.1 and 4.3 can be used to evaluate any JIT-SDP model. To avoid analyzing this procedure with JIT-SDP models that would not have been possible to produce in practice, we adopt online JIT-SDP (Cabral et al. 2019) to fully respect the chronology of the training data in this study.

The training data stream used to build our online JIT-SDP models consists of all software changes labeled until T. The software changes are labeled following the procedure proposed by Cabral et al. (2019). A software change X_u committed at Unix timestamp U is labeled as clean at Unix timestamp $U + W'$ if no defect was found to be associated to it until this timestamp, where W' is the waiting time used for the collection of training examples [4]. It is labeled as defect-inducing at the Unix timestamp corresponding to the first defect found to be associated to it. Whenever a software change is labeled, it becomes a training example in the training data stream. The training examples in the training data stream are ordered based on the Unix timestamp when they were labeled. We refer to the natural number representing the order with which training examples are produced as *training time step.*

In particular, a training data stream $\mathbb{T}_{W'}^*$ can be formulated as

$$\mathbb{T}_{W'}^* = \{(X_u, y_{u,T_u}^*)\}_{u=1}^{t_{W'}} \cup \{(X_u, 1)\}_{u=t_{W'}+1}^{t},$$

where the mathematical bold $\mathbb{T}$ is used to denote a data stream used for training committed, W' is a waiting time used to produce training examples, and $t_{W'}$ is the time step corresponding to the Unix timestamp $T_{W'} = T - W'$ denoting the number of training time steps up to the last moment when a clean software change can be labeled based on waiting time W'. Examples $\{(X_u, 1)\}_{u=t_{W'}+1}^{t}$ correspond to software changes found to be defect-inducing between Unix timestamps $T_{W'}$ and T.

The top timeline in Figure 2 shows an illustrative example of a training data stream $\mathbb{T}_{\mathbb{W}'}^*$. Software change X_1 is labeled at Unix timestamp $T_1 = U_1 + W'$, where U_1 is the Unix timestamp of its commit. Similarly, software change X_2 is labeled at Unix timestamp $T_2 = U_2 + W'$. However, software change X_3

---

[4] The training waiting time W' may or may not have the same value as the evaluation waiting time W discussed in Section 4.1.

Table 2 Summary of the datasets investigated in this work. Source: Song and Minku (2023). "%Defect-inducing" represents the percentage of defect-inducing software changes over the total number of extracted ones, and "Retained at 99%-quantile" denotes the number of retained software changes that have at least 99% confidence on the observed labels.

Datasets (Projects)	Total Changes	%Defect-inducing	Retained 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
Fabric	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

is labeled at Unix timestamp $T_3 < U_3 + W'$, as a defect was found to be induced by it before the end of the training waiting time W'. As a result, the training example corresponding to software change X_3 appears before the one corresponding to X_2 in the training data stream. Therefore, the training time step corresponding to each software change are not the same as its commit / evaluation time step.

To build JIT-SDP models, we simulate a scenario where, whenever a training example is produced, it is immediately used for updating the JIT-SDP model. This scenario is consistent with the online learning scenario that one would be able to adopt in practice. In particular, as training examples are produced W' days after the commit and the retrospective evaluation procedure uses software changes to evaluate predictive models at their commit time, no software change is ever used for training before it is used for evaluation. No data from the future is used to train / update JIT-SDP models at present either. Specifically, whenever a software change is produced at commit timestamp U, it is predicted by the most up-to-date available JIT-SDP model that has been trained on all (and only) training examples that could be labeled before U. This software change becomes a training example and is used for training only some time after its commit.

One can refer to Song and Minku (2023) for a more thorough mathematical formulation of the online learning process for JIT-SDP.

5 Experimental Setup

Being a conceptual replication of Song and Minku (2023), we adopt the same 13 GitHub open source projects as that work to investigate the three research questions of this paper. These projects were chosen for having more than 4 years of duration (most with more than 8 years duration), rich history (>10k commits) and a wide range of defect-inducing changes ratios (from 2% to

45%). The datasets were collected using `Commit Guru` (Rosen et al. 2015), which implements the original and most popular B-SZZ algorithm (Śliwerski et al. 2005) when an issue tracking system is available and its approximation otherwise. The statistic summary of the projects is shown in Table 2. We will investigate each research question by the corresponding statistical analysis performed across these 13 datasets in Table 2.

The large duration enables us to calculate a measure of predictive performance which reflects the true performance with high confidence, so that we can compute Eqs. (2) and (4). Following Song and Minku (2023), the 99%-quantile of the time it takes to find the labels of defect-inducing software changes is calculated, and software changes committed more recently than this 99%-quantile would be eliminated. For instance, if this quantile is two years, all software changes committed within the past two years were eliminated. As a result, the remaining software changes were committed for at least more than two years, having at least 99% confidence that they are really clean if no defect has been found to be induced by them so far. Defect-inducing labels are always noise-free in this study, as they cannot involve label noise due to inadequate waiting time. As discussed in Section 3.4, the main aim of this paper is to systematically investigate whether and to what extent waiting time and the label noise resulting it can have on the validity of retrospective performance evaluation procedures. Label noise that is not induced by waiting time is out of the scope of this study, which may have different effects on the validity of performance evaluation procedures and could be investigated as a future work.

All projects have at least 5,000 software changes for which we are confident of their labeling. As in Song and Minku (2023), we retain the first 5,000 time steps of each project to answer our research questions, so that all projects investigated in this paper would have the same data stream length. This is because the impact of the data stream length will also be investigated in our analyses. Since most projects actually contain considerably more than 5,000 software changes in reality, the confidence in their true labels is higher than 99%. When computing the predictive performance using the retrospective evaluation procedure, we consider the moment of the data collection T to be the Unix timestamp of the 5,000th example. We then contrast the performance estimated based on the labels obtained through this procedure against the true performance by calculating the validity of the retrospective performance evaluation procedure using Eq. (4).

As in Song and Minku (2023), G-mean is adopted to implement performance metric $|| \cdot ||_G$. However, the equations to evaluate the predictive performance based on the G-mean (Eqs. (2) and (3)) are different from those in Song and Minku (2023), as our paper analyzes the validity of retrospective rather than continuous evaluation procedures. G-mean is the geometric mean between sensitivity (a.k.a. recall) and specificity (one minus the false positive rate) (Kubat et al. 1997). Unlike performance metrics such as F-measure (Yao and Shepperd 2021), G-mean was adopted for being robust against class imbalance, being particularly important for JIT-SDP where class imbalance

Table 3 Summary of the statistical methodology adopted for answering each RQ. Within-subject factors and the response are in line with ANOVA; Independent and dependent variables are in line with the linear regression statistics.

RQ	Statistical Methodology	Within-subject factors (Independent variables)	Response (Dependent variable)
RQ1	ANOVA	1) Waiting time W 2) Length of data stream t	The amount of label noise η_W in Eq. (1)
RQ2	Linear regression analysis	1) Evaluation label noise 2) Training label noise	The validity of performance evaluation in Eq. (4)
RQ3	Linear regression analysis	1) Evaluation waiting time 2) Training waiting time	The validity of performance Evaluation in Eq. (4)

often takes place (Cabral et al. 2019; He and Garcia 2009; Wang et al. 2018). Larger G-mean values represent better predictive performance.

Being a conceptual replication of Song and Minku (2023), we adopt the same machine learning algorithm. Oversampling Online Bagging (OOB) with Hoeffding trees (Wang et al. 2015) to update / train the JIT-SDP model whenever a training example is produced, without requiring retraining on past examples. This machine learning algorithm has been shown to work well for JIT-SDP due to its ability to tackle class imbalance evolution (Cabral et al. 2019; Tabassum et al. 2020). We conducted a grid search based on the first 500 (out of the total 5,000) software changes in the data stream of a software project for parameter tuning based on G-mean. As in Song and Minku (2023), the parameters consisted of the decay factor $\in \{0.9, 0.99\}$ and the ensemble size $\in \{5, 10, 20\}$. Given a software project, the parameter setting achieving the best G-mean (calculated in Eq. (2)) at the first 500 time steps across 30 runs was chosen. The predictive performance of the JIT-SDP model was then calculated based on the whole data stream using the best parameter setting. Hoeffding trees adopted the default parameter settings provided by the Python package *scikit-multiflow* (Yao and Shepperd 2021), following previous studies in JIT-SDP (Cabral et al. 2019; Song and Minku 2023; Tabassum et al. 2020). All analyses and statistical tests were conducted based on the mean performance across 30 runs with the chosen parameter setting. The code and data used for our experiments is released as open source source at `https://github.com/sunnysong14/jit-sdp-retrospective-pf-validity`.

5.1 Statistical Methodology for RQ1

RQ1 investigates impacts of waiting time on the amounts of label noise. Waiting time W varied among four levels (15, 30, 60 and 90 days), following the previous work (Song and Minku 2023).

The investigation for RQ1 will also take into account different lengths of the data stream (1000, 2000, 3000, 4000 and 5000 evaluation time steps), where the moment of the data collection T used by the retrospective performance evaluation procedure corresponds to the Unix timestamp of the 1000th, 2000th, 3000th, 4000th and 5000th example, respectively. The data stream length is

investigated as the proportion of noisy examples could be relative to the size of the data stream. In particular, the "tail" of the data stream could potentially contain more noise than the rest of the data stream because it is composed of more recent software changes (closer to the moment of data collection T), for which less time has passed to find defects. Therefore, for instance, if we have a larger stream length such as 5000 commits, the proportion of noisy examples is likely to be smaller, as the "tail" of the data stream is relatively small compared to the size of the data stream as a whole. Conversely, if we have a smaller stream length such as 1000 commits, the proportion of noisy examples is likely to be larger. Therefore, the impact of the length of the data stream is investigated as part of the analysis in RQ1.

We will perform Analysis of Variance (ANOVA) (Montgomery 2017) with the significance level 0.05 to analyze the impact of waiting time and data stream length on the amount of label noise in the evaluation data stream $\mathbb{D}_W^*$, following the prior work (Song and Minku 2023). The null hypothesis states that there is no difference among group means and is rejected when the p-value is smaller than the significance level 0.05. ANOVA is used instead of non-parametric statistical tests such as Friedman because it enables us to investigate multiple factors. Sphericity is an important assumption made by the repeated measures ANOVA design. Mauchly's test (Mauchly 1940) is adopted to assess the statistical assumption of sphericity when using ANOVA. When the test yields a p-value less than the significance level 0.05, we consider that the assumption has been violated. The Greenhouse-Geisser correction is then used to correct for this violation.

As shown in Table 3, the within-subject factors under investigation include the waiting time W and the data stream length t. The response variable is the amount of label noise η_W in Eq. (1).

5.2 Statistical Methodology for RQ2

RQ2 investigates impacts of label noise on the validity of retrospective performance evaluation procedures. Following the prior work (Song and Minku 2023), we will perform linear regression analyses with the significance level 0.05 for this purpose. The linear regression approach adopted Ordinary Least Squares to learn the model. As shown in Table 3, in addition to the label noise of evaluation examples (evaluation label noise), we will also consider the label noise of training examples (training label noise) as independent variables, for a more thorough analysis. Different from the waiting time used for evaluation purposes (the main concern of this paper), the training waiting time is used to produce the data stream for training JIT-SDP models. Different training waiting times can lead to different levels of noise in the training data and consequently produce different JIT-SDP models. Both the training and evaluation waiting times used to compute the amount of noise varied among 15, 30, 60 and 90 days. As the waiting time used for evaluation purposes is the main topic of this work, whenever using the term "waiting time" on its own,

we mean the "evaluation waiting time"; whereas the training waiting time will always be explicitly referred to as "training waiting time".

Including training label noise would enable the analysis to consider to what extent different JIT-SDP models could impact the conclusions of this study. The dependent variable is the validity of retrospective performance evaluation procedures formulated in Eq. (4). The p-value of each independent variable tests the null hypothesis that the corresponding coefficient equals to zero (no effect on the dependent variable). The linear regression statistical test is considered significant if its p-value is smaller than the significance level 0.05. ANOVA, which was used to answer RQ1, is not viable for answering RQ2. This is because the independent variables are continuous but not ordinal, so one cannot set up the levels of within-subject factors (Montgomery 2017).

5.3 Statistical Methodology for RQ3

RQ3 investigates impacts of waiting time on the validity of retrospective performance evaluation procedures. Following the prior work (Song and Minku 2023), we will perform linear regression analyses with the significance level 0.05 for that. The linear regression approach adopted Ordinary Least Squares to learn the model. As shown in Table 3, we will consider the evaluation waiting time and the training waiting time as the two independent variables in the linear regression analyses for enabling a more thorough analysis of the validity. Both have values varying among 15, 30, 60 and 90 days. The dependent variable is the validity of retrospective performance evaluation procedures in Eq. (4). ANOVA adopted for answering RQ1 is not viable for RQ3, because there is a constraint between the two independent variables: the evaluation waiting time should be no larger than the training one to follow the principles of the online learning procedure, as explained in Song and Minku (2023).

6 Experimental Results

6.1 RQ1: Impact of Waiting Time on the Amount of Label Noise

In this section, ANOVA is used to analyze the influence of each of the two factors (the waiting time and the data stream length) on the response variable (the amount of label noise). Mauchly's tests show that the sphericity assumptions on the waiting time W and the length of data stream t are violated with p-values 4.42E-11 and 7.40E-5, respectively. Greenhouse-Geisser corrections are thus adopted to account for such violations.

ANOVA with Greenhouse-Geisser corrections reports that W has significant impact on the amount of label noise $\eta(W)$ with p-value 0.016, but t and the interaction $W * t$ are not found to have significant impact on it with p-values 0.868 and 0.184, respectively. Effect size in terms of partial eta-square is 0.447 for W, which is large compared to 0.009 for t, and 0.163 for the interaction $W * t$. Pairwise comparisons with post-hoc Bonferroni between different

waiting times find significant differences between 15 vs 30 days with p-value 0.024 and between 15 vs 60 days with p-value 0.033. No significant difference was found between the other values.

We originally conjectured that the data stream length might have an impact on label noise, because a larger proportion of the shorter data streams would likely be affected by label noise. However, the statistical analysis shows that the data stream length does not have significant impact on the amount of label noise. This suggests that, for different data stream lengths over and including 1000, one does not need to be concerned with the impact of the data stream length on label noise in the retrospective performance evaluation scenario. A potential reason is that length of 1000 has been already large enough to avoid such impact.

Figure 3 shows impact of waiting time on the amount of label noise. Corresponding numeric values can be found in the supplementary material. Each value in Figure 3 represents the average label noise across different lengths of data stream. Averaging across the data stream length is reasonable because it has no significant impact on the label noise. The plot for the impact of the data stream length on label noise is not shown as this impact is not significant. As shown in Figures 3(a)∼3(m), even though individual plots per dataset were different from each other, larger waiting times usually lead to smaller label noise (except for jGroup). This is reasonable since a larger waiting time would allow for more opportunities to find defects induced by software changes, potentially contributing to smaller amounts of label noise. Figure 3(n) shows the plot of median label noise across all datasets. We can see that a larger waiting time of 90 days causes a median drop of 14.93% in the proportion of defect-inducing examples labeled as clean across datasets. Therefore, different waiting times can typically cause considerable difference in the amount of label noise, even though such differences can be smaller for some datasets (minimum of around 4% in Broadleaf), and larger for others (maximum of 45.86% in wp-Calypso).

Altogether, larger waiting time led to significant reduction on the amount of label noise; the data stream length did not have significant impact on label noise. Section 6.2 will investigate whether such amounts of label noise would be large enough to have a significant impact on the validity of retrospective performance evaluation procedures.

📎 **Answer to RQ1**

Smaller waiting times were found to be associated with significantly larger amount of label noise. The proportion of defect-inducing examples labeled as clean increased by up to 45.86% as a result of smaller waiting time. The data stream length did not have significant impact on the amount of label noise.
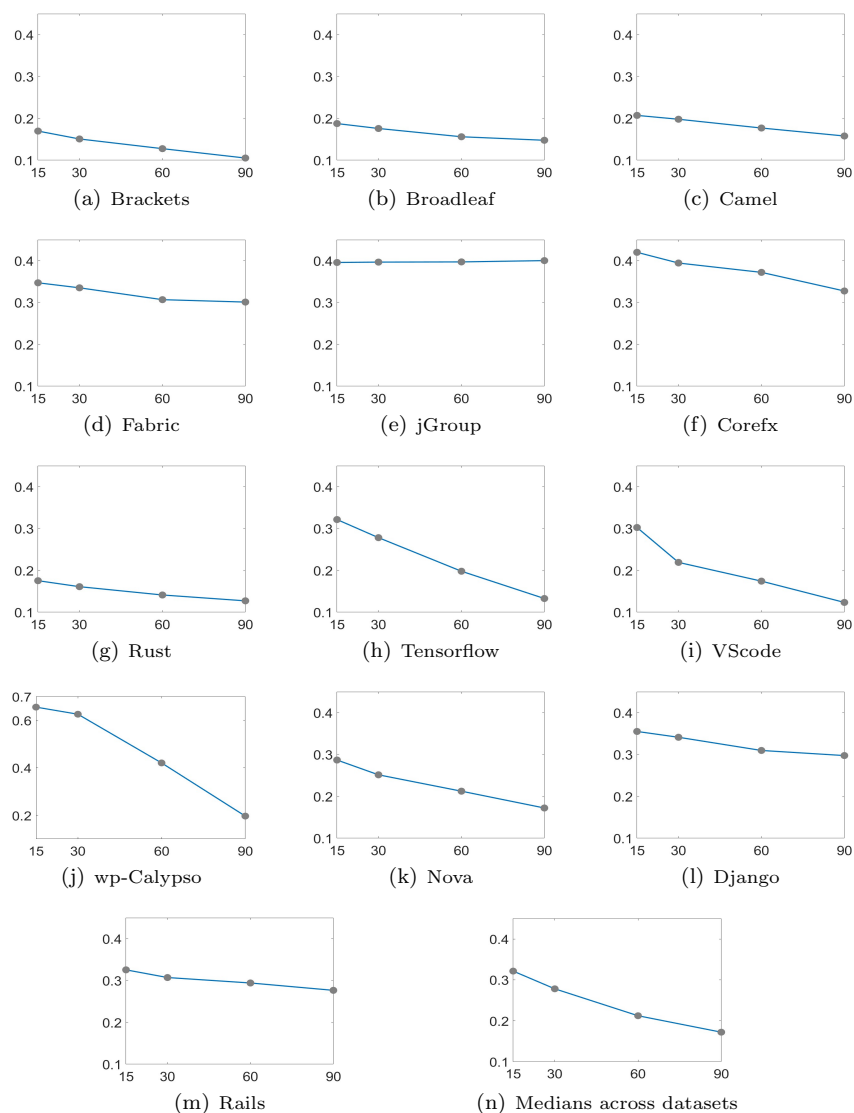
Fig. 3 RQ1: Impact of waiting time (x-axis) on the amount of label noise (y-axis) in the retrospective performance evaluation scenario. The impact of the length of data stream on label noise is not shown as this impact is not significant. Values in the y-axis of Figure 3(n) are the medians of the amount of label noise across all datasets. We show the range of y-axis between 0.1 and 0.45 to facilitate visualization for all datasets except for wp-Calypso.

6.2 RQ2: Impact of Label Noise on the Validity of Retrospective
Performance Evaluation Procedures

The linear regression analysis conducted for RQ2 shows that the linear relationship between the two independent variables and the dependent one is significant with p-value 6.1834E-11. So, we can continue to investigate the model coefficients. Both the evaluation and the training label noise were found to have significant impact on the validity of retrospective performance evaluation procedures with p-values 0.023215 and 7.2162E-10, respectively.

The standardized coefficient for the amount of evaluation label noise was -0.172135, showing a significant negative impact of the evaluation label noise on the performance validity. This means that larger evaluation label noise typically associates to significantly worse validity of the retrospective performance evaluation procedure. The standardized coefficient for the amount of training label noise was -0.499430, showing a significant negative impact of the training label noise on the validity of retrospective performance evaluation procedures. This means that larger training label noise typically means significantly worse validity of retrospective performance evaluation procedures.

Figure 4 shows the relationship between the evaluation (training) label noise and the validity of retrospective performance evaluation procedures in orange dotted (blue solid) lines for all datasets. Corresponding numerical values can also be found in the supplementary material. Each reported orange (blue) value represents the average performance validity across all training (evaluation) label noises arising from different waiting times investigated (15, 30, 60 and 90 days).

As shown by the orange dot lines in Figures 4(a)∼4(m), even though the effects of evaluation label noise on the validity in individual datasets were different, they typically showed decreasing trends (except for Fabric, jGroup, and wp-Calypso). In this sense, larger evaluation label noise typically associates to worse performance validity. Nevertheless, in most datasets, the drops in the magnitude of the validity related to larger evaluation label noise were small, with the validity differing by less than 1%, though some datasets might also suffer from drops of around 2% (Nova). As shown by the orange dot line in Figure 4(n), in general, a larger evaluation label noise caused a median drop of only 0.3987% in the validity across datasets. Therefore, larger evaluation label noise typically meant worse validity of retrospective performance evaluation procedures, but the magnitude of the drop in the validity was small.

As shown by the blue solid lines of Figures 4(a)∼4(m), even though the effects of training label noise on the validity in individual datasets were different, they typically presented decreasing trends (except for Bracket, Fabric, Corefx, and Rails). In this sense, larger training label noise typically means significantly worse validity of retrospective performance evaluation procedures. Nevertheless, in most datasets, the drops in the magnitude of the validity caused by larger training label noise were small, with the validity differing by less than 1%, though some datasets might also suffer from drops in validity of up to around 3% (Rust). As shown by the blue solid line of Figure 4(n),
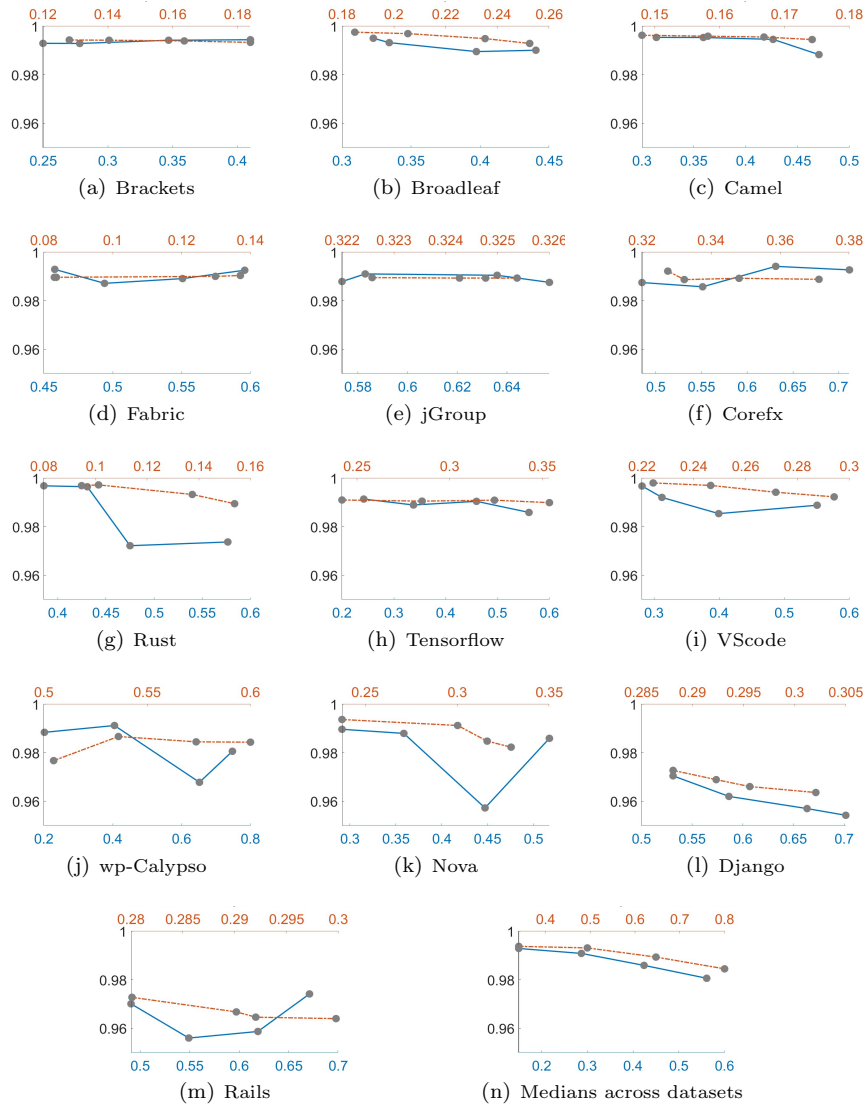
Fig. 4 RQ2: Impact of the training label noise (blue solid lines in the lower x-axis) and the evaluation label noise (orange dotted lines in the upper x-axis), arising from different waiting times, on the validity of retrospective performance evaluation procedures (in y-axis). Reported values in Figure 4(n) are the medians across all datasets to demonstrate overall impacts on the performance validity. Note that the ranges of the x-axis may differ with respect to the training and the evaluation label noise in order to facilitate visualization.

in general, a larger training label noise caused a median drop of only 0.4935% in the validity across datasets. Therefore, larger training label noise would

typically associate to worse validity of retrospective performance evaluation procedures, but the magnitude of the drop in the validity would be small.

> 📎 **Answer to RQ2**
>
> Both evaluation and training label noise had significant negative impact on the validity of retrospective performance evaluation procedures. However, the magnitude of the changes in the validity caused by different amounts of label noise was small (varying up to around 2% for evaluation label noise and 3% for training label noise, but most of the time being less than 1%).

6.3 RQ3: Impact of Waiting Time on the Validity of Retrospective Performance Evaluation Procedures

RQ1 and RQ2 have investigated the impact of waiting time on the validity of retrospective performance evaluation procedures via label noise as a mediator. However, different choices of waiting time might also affect the validity that cannot be captured by label noise. Such impact could either increase or reduce the effect of label noise on the validity. RQ3 is designed to investigate this.

The linear regression analysis conducted for RQ3 shows that there is a significant linear relationship between the two independent variables and the dependent variable in the retrospective performance evaluation scenario with p-value 0.015. So, we move to the analysis of the impact of each independent variable on the dependent one.

No significant impact of the evaluation waiting time on the validity of retrospective performance evaluation procedure was found with p-value 0.564. Therefore, even though the label noise caused by different evaluation waiting times has a significant (but small) impact on the validity of retrospective performance evaluation procedures, other factors associated to different choices of waiting time are likely to moderate this effect, resulting in evaluation waiting time not having a significant impact on the validity. A discussion on such factors is provided in Section 6.4.2.

Training waiting time was found to have a significant impact on the validity of retrospective performance evaluation procedures with p-value 0.028. The standardized coefficient was 0.22, showing that the training waiting time had positive impact on the performance validity, i.e., larger training waiting times are associated to better validity.

Figure 5 shows the relationship between training waiting time and the validity of retrospective performance evaluation procedures in the blue solid lines for all datasets. Corresponding numerical values can also be found in the supplementary material. Each reported value represents the average validity of retrospective performance evaluation procedures across all evaluation waiting times investigated (15, 30, 60 and 90 days). Averaging across evaluation
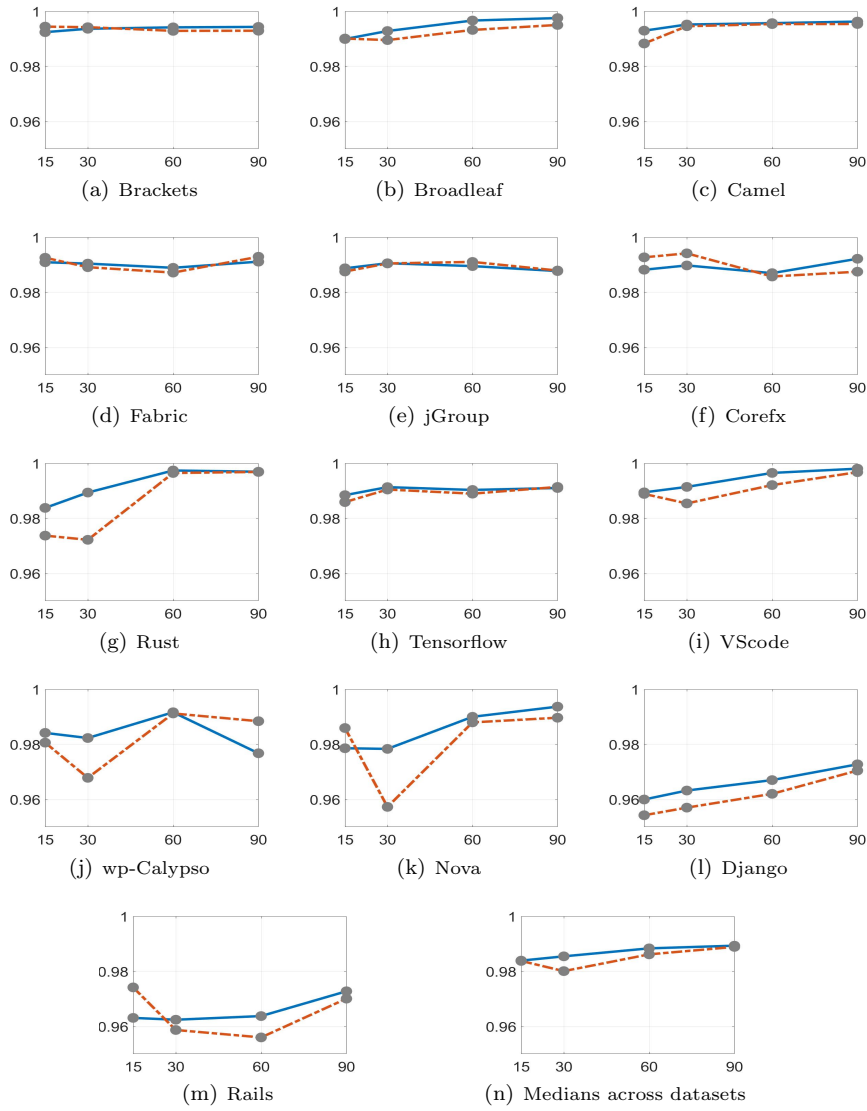
Fig. 5 RQ3: Impact of the training waiting time (blue solid lines in x-axis) and the evaluation waiting time (orange dotted lines in x-axis) on the validity of retrospective performance evaluation procedures (in y-axis). Reported values in Figure 5(n) are the medians across all datasets to demonstrate overall impacts on the performance validity.

waiting times is reasonable because the evaluation waiting time has no significant impact on the validity. Despite that, we also show the validity plots for different evaluation waiting times in orange dotted lines to demonstrate that the validity of the retrospective performance evaluation procedure was high in all datasets.

As shown in Figures 5(a)~5(m), even though smaller training waiting times were sometimes associated to better validity of performance evaluation procedures (jGroup and wp-Calypso), large training waiting times typically positively impact the validity of retrospective performance evaluation procedures. This means that, depending on the actual JIT-SDP model maintained for a given project and being evaluated, the validity of retrospective performance evaluation procedures may be better or worse. However, despite the significant impact of training waiting time on the validity, the increases in the magnitude of the validity resulting from larger training waiting times were not large, with the validity differing by less than 1 percentage point in most datasets; the differences were of at most around 2% (Django and Rust). Figure 5(n) shows the plot of median validity of performance evaluation procedures across datasets, for different waiting times. We can see that a larger evaluation waiting time of 90 days causes a median increase of even less than 0.5% in the validity compared to that of 15 days, being of small magnitude. Therefore, such impact is unlikely to be practically relevant when evaluating JIT-SDP models.

> **✎ Answer to RQ3**
>
> No significant impact of evaluation waiting time was found on the validity of retrospective performance evaluation procedures. Training waiting time had a significant impact on the validity of retrospective performance evaluation procedures, meaning that the validity may be better or worse depending on the actual JIT-SDP model being evaluated. However, the changes in validity were small (typically less than 1%, and up to around 2%), and so this impact is unlikely to be practically relevant.

6.4 Discussion and Implications

6.4.1 High Validity of Retrospective Performance Evaluation Procedures

From the analyses presented in Sections 6.1 to 6.2, we know that, despite different evaluation waiting times leading to significantly different amounts of label noise and such amounts of label noise having a significant impact on the validity of retrospective performance evaluation procedures, the magnitude of differences in the validity was rather small. When investigating the impact of evaluation waiting time on the validity in Section 6.3, such impact was not significant. This indicates that other factors related to evaluation waiting time are likely to moderate the effect of label noise, canceling it out. Therefore, the choice of evaluation waiting time among 15, 30, 60 and 90 days investigated in this study is unlikely to matter when evaluating JIT-SDP in the retrospective performance evaluation scenario.

Such conclusion would be irrelevant if all of these waiting times had led to equally poor (low) performance validity. In particular, that would have meant that larger waiting time values might need to be investigated, to see if they

would lead to significantly better validity. However, Figure 5 shows that the validity was indeed very high, being close to 1 (100%) in all datasets. This result is very encouraging as such high validity values mean that not only the results of existing JIT-SDP studies that have not cut a large portion of data streams to prevent the validity issues are likely to remain valid, but also that researchers can make use of relatively small or recent projects with just 1000 software changes in their studies when adopting retrospective evaluation procedures. Such studies can be conducted by adopting an evaluation waiting time as small as 15 days, leading to a relatively small portion of the software changes of the project having to be eliminated. Future work could investigate whether even smaller projects are also possible. It is worth noting that, if the waiting times investigated in this study had not been large enough to achieve high validity, this would have meant that studies to evaluate JIT-SDP could only be performed with much larger projects, so that a very large portion of the tail of their corresponding data streams could be removed, as done in this work for computing the "true" predictive performance with high confidence.

Such high validity values also mean that the predictive performances being reported in research studies are close to the predictive performances that would have been obtained if verification latency was not an issue in JIT-SDP. This is also very encouraging. It means that practitioners can more confidently use knowledge of such estimated performances to decide whether or not they would like to adopt or investigate a given JIT-SDP model in their companies.

It is worth noting that our study investigated the impact of waiting time on the validity of retrospective performance evaluation procedures. This is different from investigating the impact of waiting time on the predictive performance of JIT-SDP models. The latter is about measuring the impact of waiting time on the capability of JIT-SDP models to correctly predict labels of software changes. The former is about measuring the impact of waiting time on the procedure used to evaluate what the predictive performance of such models is. We propose the former to be investigated as future work.

6.4.2 Contrasting Validity Results of Retrospective vs Continuous Evaluation Scenarios

Being a conceptual replication of Song and Minku (2023), it is important to compare the results obtained by our paper in the context of retrospective performance evaluation procedures with those obtained by Song and Minku (2023) in the context of continuous performance evaluation procedures. The results of this study are different from those obtained in Song and Minku (2023), despite having some similarities. The main difference is that we found that evaluation waiting time did not have a significant impact on the validity in the context of retrospective performance evaluation procedures, whereas Song and Minku (2023) concluded that it had a significant impact on the validity in the context of continuous performance evaluation procedures. The potential reasons for such differences are as follows.

Even though both our study and Song and Minku (2023) found that waiting time had a significant impact on label noise, the magnitude of this impact was smaller in the retrospective scenario than in the continuous scenario. In particular, even though the amount of label noise generated by waiting time was somewhat large in the retrospective scenario (around 0.3 for the waiting time of 15 days), it was even larger in the continuous evaluation scenario (around 0.5 for the waiting time of 15 days (Song and Minku 2023)). This may have contributed to a larger magnitude of the impact of label noise on the validity of continuous performance evaluation scenario (by up to around 3% in Song and Minku (2023)), compared to that of the retrospective evaluation scenario (by usually less than 1%).

Moreover, Song and Minku (2023) found that the obsolescence of past examples (which are used to estimate the current performance in the continuous evaluation scenario) played a significant role in affecting the validity of continuous performance evaluation procedures, resulting in smaller waiting times leading to higher validity (despite it leading to larger amounts of label noise). The role of the obsolescence of the past examples was strong, because it affected every example in the continuous evaluation data stream.

In the context of retrospective performance evaluation, the obsolescence of examples is also a factor that can influence the impact of waiting time on the validity. This is because the true predictive performance is computed with examples obtained until Unix timestamp T, whereas the observed retrospective performance evaluation procedure uses only examples obtained until $T - W$. Therefore, the larger the waiting time W, the more obsolete the observed evaluation data stream $\mathbb{D}_W^*$ is compared to the true $\mathbb{D}$. However, the number of newer examples in $\mathbb{D}$ that are not present in $\mathbb{D}_W^*$ is small compared to the whole data stream size. Therefore, the role of the obsolescence becomes less impactful than in the continuous performance evaluation scenario. It becomes a moderator of the already small impact of label noise on the validity, such that the impact of waiting time on validity became insignificant. This role was thus that of a moderator that reduces the impact of waiting time on the validity, rather than changing the impact from negative (lower waiting time leading to lower validity) to positive (lower waiting time leading to better validity) as in Song and Minku (2023).

7 Threats to Validity

This section discusses the threats to validity of our study, which are similar to the threats to validity of Song and Minku (2023).

Construct Validity. We carefully chose G-mean as the evaluation metric whenever the performance of JIT-SDP was required to compute in the analyses of this study. Adopting G-mean is adequate due to its insensitivity to the class imbalance issue (Wang et al. 2018), which is particularly important for JIT-SDP that typically suffers from the class imbalance issue (Cabral et al. 2019). G-mean is the most widely used metric in online class imbalance learning

studies (Wang et al. 2018). We used grid search based on an initial portion of the data stream to tune parameters of the machine learning algorithms used in this study. Random search might find better parameter values than grid search (Bergstra and Bengio 2012). However, whether or not this is the case in data stream learning is still an open question, as the best values for the initial portion of the data stream are not necessarily the best for the remaining of the data stream due to concept drift, which is frequently occurred in JIT-SDP (Cabral and Minku 2022 (in press); Cabral et al. 2019; McIntosh and Kamei 2018). Moreover, this paper is concerned with investigating the validity of the performance evaluation procedures rather than with improving predictive performance of JIT-SDP. The specific choice of parameter tuning method is less relevant in this context than in studies targeted at improving predictive performance of JIT-SDP models.

Internal Validity. A potential threat of the internal validity is that the true labels of some defect-inducing software changes may never be accessible when the defects induced by them are not induced until the end of the data stream due to very large verification latency. To mitigate this threat, we used open source projects covering a period of at least four years and eliminated software changes from the latter periods of data streams.

External Validity. We have investigated 13 open source projects, with 4 levels of waiting time, and 5 lengths for data streams, covering a range of different characteristics in previous JIT-SDP studies. However, as with any study involving machine learning, results may not generalize to other contexts. Moreover, our study focuses on OOB with Hoeffding trees, which have been previously adopted for online JIT-SDP (Cabral et al. 2019; Song and Minku 2023; Tabassum et al. 2020). Being a conceptual replication of Song and Minku (2023), we adopt the same machine learning approach as in that paper. Other types of machine learning approaches could be investigated as future work following the same investigation procedures and statistical methodologies for answering the same RQs in future work. The conclusions of our study are in the context of noise resulting from waiting time when using SZZ for data collection. Label noise not caused by waiting time may have different effects on the validity and could be investigated as a future work. Similarly, different conclusions may be obtained regarding the impact of waiting time if a different algorithm from SZZ is adopted for data collection.

8 Conclusion

We conducted the first analysis of the extent with which the conclusions of JIT-SDP research studies are (in)valid in view of the fact that observed labels rather than the true labels of software changes are being used for conducting retrospective performance evaluation procedures. We conduct our investigation by answering three research questions as below.

RQ1. How large is the amount of label noise caused by different waiting times in retrospective JIT-SDP data collection? We found

that smaller waiting times were associated to significantly larger amount of label noise. The proportion of noisy defect-inducing examples labeled as clean increased by up to 45.86% as a result of smaller waiting time.

RQ2. To what extent is the validity of retrospective performance evaluation procedures impacted by label noise resulting from waiting time? We found that both the evaluation and the training label noise had significant negative impact on the validity of retrospective performance evaluation procedures. However, the magnitude of the changes in the validity was typically small, varying up to around 2% for evaluation label noise and 3% for training label noise, but most of the time being less than 1%.

RQ3. To what extent is the validity of retrospective performance evaluation procedures impacted by different waiting times? No significant impact of the evaluation waiting time was found on the validity of retrospective performance evaluation procedures. Training waiting time had a significant impact on the validity, meaning that the validity of performance evaluation procedures may be better or worse depending on the actual JIT-SDP model being evaluated. However, the changes in validity were small (up to around 2%), and so this impact is unlikely to be relevant.

Besides the investigation of the three research questions, our results also report that the validity of retrospective performance evaluation procedures was high in magnitude even when using small evaluation waiting times. This is an encouraging result, as it means that future studies can make use of not only larger (with 5k+ software changes) but also smaller (with 1k software changes) software projects for evaluating predictive performance of JIT-SDP models. This is particularly important in terms of having a validated performance evaluation, as many software companies have projects of short duration compared to some of the existing open source projects that have run for several years. With this in mind, people would feel safe to trust the estimated performance even for smaller software projects in the retrospective performance evaluation scenario.

As future work, other performance metrics, machine learning algorithms, and sources of label noise can be investigated. The impact of waiting time on the predictive performance of JIT-SDP models could also be investigated.

Data Availability Statements

A replication package can be found in the JIT-SDP-Retrospective-PF-Validity repository, `https://github.com/sunnysong14/jit-sdp-retrospective-pf-validity`. The datasets generated during and/or analyzed during the current study are available in the JIT-SDP-DATA repository, `https://github.com/sunnysong14/jit-sdp-data`.

Declarations

Funding and/or Conflicts of interests/Competing interests:

References

Antoniol G, Ayari K, Di Penta M, Khomh F, Guéhéneuc YG (2008) Is it a bug or an enhancement? a text-based approach to classify change requests. In: Conference of the Center for Advanced Studies on Collaborative Research: Meeting of Minds, Ontario, Canada, pp 304–318

Aranda J, Venolia G (2009) The secret life of bugs: Going past the errors and omissions in software repositories. In: International Conference on Software Engineering, Vancouver, BC, Canada, pp 298–308

Bergstra J, Bengio Y (2012) Random search for hyper-parameter optimization. Journal of Machine Learning Research 13:281–305

Bird C, Bachmann A, Aune E, Duffy J, Bernstein A, Filkov V, Devanbu P (2009) Fair and balanced? bias in bug-fix datasets. In: International Symposium on the Foundations of Software Engineering, Amsterdam, The Netherlands, pp 121–130

Cabral GG, Minku LL (2022 (in press)) Towards reliable online just-in-time software defect prediction. IEEE Transactions on Software Engineering DOI 10.1109/TSE.2022.3175789

Cabral GG, Minku LL, Shihab E, Mujahid S (2019) Class imbalance evolution and verification latency in just-in-time software defect prediction. In: International Conference on Software Engineering, Montreal, QC, Canada, pp 666–676

Chawla NV, Bowyer KW, Hall LO, Kegelmeyer WP (2002) SMOTE: Synthetic minority over-sampling technique. Journal of Artificial Intelligence Research 16:321–357

Chen TH, Nagappan M, Shihab E, Hassan AE (2014) An empirical study of dormant bugs. In: Conference on Mining Software Repositories, Hyderabad, India, pp 82–91

Da Costa DA, McIntosh S, Shang W, Kulesza U, Coelho R, Hassan AE (2017) A framework for evaluating the results of the SZZ approach for identify-

ing bug-introducing changes. IEEE Transactions on Software Engineering 43(7):641–657

Ditzler G, Roveri M, Alippi C, Polikar R (2015) Learning in non-stationary environments: A survey. IEEE Computational Intelligence Magazine 10(4):12–25

Ekanayake J, Tappolet J, Gall HC, Bernstein A (2012) Time variance and defect prediction in software projects. Empirical Software Engineering 17(4):348–389

Eyolfson J, Tan L, Lam P (2011) Do time of day and developer experience affect commit bugginess? In: International Workshop on Mining Software Repositories, Waikiki, Honolulu, HI, USA, pp 153–162

Fan Y, Xia X, Da Costa DA, Lo D, Hassan AE, Li S (2019) The impact of mislabeled changes by SZZ on just-in-time defect prediction. IEEE transactions on software engineering 47(8):1559–1586

Gama J, Sebastiao R, Rodrigues PP (2013) On evaluating stream learning algorithms. Journal of Machine Learning 90(3):317–346

Harman M, Islam S, Jia Y, Minku LL, Sarro F, Srivisut K (2014) Less is more: Temporal fault predictive performance over multiple hadoop releases. In: Symposium on Search-Based Software Engineering (SSBSE'14), Lecture Notes in Computer Science Volume 8636, Cham, pp 240–246

Hassan AE (2009) Predicting faults using the complexity of code changes. In: International Conference on Software Engineering, Vancouver, BC, Canada, pp 16–24

He H, Garcia EA (2009) Learning from imbalanced data. IEEE Transaction on Knowledge and Data Engineering 21(9):1263–1284

Herbold S, Trautsch A, Trautsch F, Ledel B (2022) Problems with SZZ and features: An empirical study of the state of practice of defect prediction data collection. Empirical Software Engineering 27(2)

Herzig K, Just S, Zeller A (2013) It is not a bug, it is a feature: How misclassification impacts bug prediction. In: International Conference on Software Engineering, San Francisco, CA, USA, pp 392–401

Hoang T, Khanh Dam H, Kamei Y, Lo D, Ubayashi N (2019) DeepJIT: an end-to-end deep learning framework for just-in-time defect prediction. In: International Conference on Mining Software Repositories, Montreal, QC, Canada, pp 34–45

Hoang T, Kang HJ, Lo D, Lawall J (2020) CC2Vec: Distributed representations of code changes. In: International Conference on Software Engineering, Seoul, South Korea, pp 518–529

Kabir MA, Keung JW, Bennin KE, Zhang M (2019) Assessing the significant impact of concept drift in software defect prediction. In: Computer Software and Applications Conference, Milwaukee, WI, USA, vol 1, pp 53–58

Kamei Y, Shihab E, Adams B, Hassan AE, Mockus A, Sinha A, Ubayashi N (2013) A large-scale empirical study of just-in-time quality assurance. IEEE Transactions on Software Engineering 39(6):757–773

Kamei Y, Fukushima T, Mcintosh S, Yamashita K, Ubayashi N, Hassan AE (2016) Studying just-in-time defect prediction using cross-project models.

Empirical Software Engineering 21(5):2072–2106

Kim S, Zimmermann T, Pan K, Jr Whitehead EJ (2006) Automatic identification of bug-introducing changes. In: Automated Software Engineering, Tokyo, Japan, pp 81–90

Kim S, Whitehead EJ, Zhang Y (2008) Classifying software changes: Clean or buggy? IEEE Transactions on Software Engineering 34(2):181–196

Kim S, Zhang H, Wu R, Gong L (2011) Dealing with noise in defect prediction. In: International Conference on Software Engineering, Honolulu, HI, USA, pp 481–490

Kubat M, Holte R, Matwin S (1997) Learning when negative examples abound. In: European Conference on Machine Learning, Berlin, Heidelberg, pp 146–153

Mauchly JW (1940) Significance test for sphericity of a normal n-variate distribution. The Annals of Mathematical Statistics 11(2):204–209

McIntosh S, Kamei Y (2018) Are fix-inducing changes a moving target? a longitudinal case study of just-in-time defect prediction. IEEE Transactions on Software Engineering 44(5):412–428

Misirli AT, Shihab E, Kamei Y (2016) Studying high impact fix-inducing changes. Empirical Software Engineering Journal 21(2):605–641

Mockus A, Weiss DM (2000) Predicting risk of software change. Bell Labs Technical Journal 5(2):169–180

Montgomery DC (2017) Design and analysis of experiments. John wiley & sons

Nam J, Kim S (2015) Clami: Defect prediction on unlabelled datasets. In: International Conference on Automated Software Engineering, Lincoln, NE, USA, pp 452–463

Neto EC, da Costa DA, Kulesza U (2018) The impact of refactoring changes on the SZZ algorithm: An empirical study. In: International Conference on Software Analysis, Evolution and Reengineering, Campobasso, Italy, pp 380–390

Nguyen HM, Cooper EW, Kamei K (2011) Online learning from imbalanced data streams. In: International Conference of Soft Computing and Pattern Recognition, Dalian, China, pp 347–352

Pornprasit C, Tantithamthavorn CK (2021) JITLine: a simpler, better, faster, finer-grained just-in-time defect prediction. In: International Conference on Mining Software Repositories, Madrid, Spain, pp 369–379

Rosen C, Grawi B, Shihab E (2015) Commit guru: analytics and risk prediction of software commits. In: International Symposium on the Foundations of Software Engineering, pp 966–969

Shihab E, Hassan AE, Adams B, Jiang ZM (2012) An industrial study on the risk of software changes. In: International Symposium on the Foundations of Software Engineering, Cary, North Carolina, pp 1–11

Śliwerski J, Zimmermann T, Zeller A (2005) When do changes induce fixes? ACM sigsoft software engineering notes 30(4):1–5

Song L, Minku LL (2023) A procedure to continuously evaluate predictive performance of just-in-time software defect prediction models during software

development. IEEE Transactions on Software Engineering 49(2):646–666

Tabassum S, Minku LL, Feng D, Cabral GG, Song L (2020) An investigation of cross-project learning in online just-in-time software defect prediction. In: International Conference on Software Engineering, Seoul, South Korea, p 554–565

Tan M, Tan L, Dara S, Mayeux C (2015) Online defect prediction for imbalance data. In: International Conference on Software Engineering, Florence, Italy, pp 99–108

Tantithamthavorn C, McIntosh S, Hassan AE, Ihara A, Matsumoto K (2015) The impact of mislabelling on the performance and interpretation of defect prediction models. In: International Conference on Software Engineering, Florence, Italy, vol 1, pp 812–823

Wang S, Yao X (2013) Using class imbalance learning for software defect prediction. IEEE Transaction on Reliability 62(2):434–443

Wang S, Minku LL, Yao X (2015) Resampling-based ensemble methods for online class imbalance learning. IEEE Transaction on Knowledge and Data Engineering 27(5):1356–1368

Wang S, Minku LL, Yao X (2018) A systematic study of online class imbalance learning with concept drift. IEEE Transaction on Neural Networks and Learning Systems 29(10):4802–4821

Yao J, Shepperd M (2021) The impact of using biased performance metrics on software defect prediction research. Information and Software Technology 139:106664

Yatish S, Jiarpakdee J, Thongtanunam P, Tantithamthavorn C (2019) Mining software defects: Should we consider affected releases? In: International Conference on Software Engineering, Montreal, QC, Canada, pp 654–665