Rethinking Effort Awareness: A Novel Perspective on Applicable Just-In-Time Defect Prediction

Peter Bludau and Alexander Pretschner

***Abstract*—Reviewing software changes is crucial, as it mitigates the introduction of defects and thereby saves time and reduces costs. Just-in-time (JIT) defect prediction has emerged as an approach to support the review process by predicting the likelihood of defects in new commits. Effort-aware evaluations were proposed to better manage developers' limited time to review changes and analyze the applicability of JIT defect prediction approaches. However, current effort-aware approaches neglect the time-dependent nature of software engineering, thus overstating the performance and applicability of such approaches. Further, they do not reflect state-of-the-art software development practices. In this work, we discuss these limitations and propose a paradigm shift to evaluate JIT defect prediction more realistically and redirect the focus to saving effort under the condition that defective commits are still reviewed. Thus, we propose performance metrics that better represent applicable JIT defect prediction. We further analyze reliability techniques that adapt the prediction results and allow for a risk-based application of JIT defect prediction models. Taking this new perspective, we find that while still reviewing 95% of defective commits, on average, 46% of the non-defective commits are correctly identified by JIT defect prediction models and can be skipped; therefore, 20% of the total avoidable effort can be saved by employing JIT defect prediction models.**

***Index Terms*—Just-In-Time Defect Prediction, Effort Awareness, Reliability, Empirical Evaluation**

I. Introduction

When developing software, it is important to review changes in order to discuss arising issues and find defects early. To aid this process, just-in-time (JIT) defect prediction assigns a risk value to every new commit in the version control system to help developers decide if a review is needed [1]. In recent years, researchers have focused on effort-aware approaches to investigate the applicability of JIT defect prediction approaches under realistic conditions. Their primary objective is to identify as many defective commits as possible within a limited inspection budget. This involves ranking commits based on their probability to introduce defects and the associated effort needed to inspect these changes [2]. Based on this ranking, commits are reviewed until the predefined inspection budget is exhausted. Most studies assess the performance of their approaches by reporting the proportion of all defective commits that can be identified using 20% of the total inspection effort (e.g., [3]–[6]).

While state-of-the-art JIT defect prediction evaluations include the developers' effort, several assumptions of these effort-aware approaches contradict current development practices and call into question the value of such assessments. First, effort-aware approaches rank all changes to find a large number of defective commits with limited effort, thereby neglecting the chronological order of commits. This means that reviews are postponed until multiple changes are committed. This contradicts the intuition of JIT defect prediction, intending to evaluate new commits immediately and rendering current effort-aware approaches impractical [7]. Second, the predefined inspection budget used in the evaluation can only be determined after the development has ended. Third, best practices in software engineering mandate a lightweight code review for every commit, further questioning the general presupposition of a limited and fixed inspection budget. We argue that current effort-aware evaluations fail to represent applicable JIT defect prediction, and the concept needs to be reevaluated. A detailed description of effort awareness and its limitations is provided in Section II-B and Section III, respectively.

In order to measure the benefit of JIT defect prediction models in a realistic setting, we propose necessary adjustments. Given the need for immediate reviews in a just-in-time context [8], we align evaluations with the time dependency of JIT defect prediction, keeping the chronological order of commits and promptly assigning reviews after each change. Further, we propose performance metrics that put an emphasis on the saved effort and are not restrained by a fixed inspection budget. From this new perspective, we assess the potential value that a developer can gain applying JIT defect prediction systems and explore techniques to enhance JIT defect prediction applicability by answering the following questions:

- **RQ1**: What are the key metrics to evaluate applicable just-in-time defect prediction, and how do state-of-the-art prediction models perform in terms of these metrics?
- **RQ2**: To what extent can reliability measures be utilized to enhance the applicability of defect prediction approaches?

Our research shifts the focus from assessing the effectiveness of JIT defect prediction models to a balanced and efficiency-oriented approach. We aim to detect the majority of defective commits while optimizing for saved effort. We find that adjusting models for reliability allows the identification of most defects while preserving significant portions of effort. While these adapted models provide applicable JIT defect prediction, fine-tuning them remains a manual task.

II. Background and Related Work

In this Section, we describe the background and related work regarding JIT defect prediction approaches and their effort awareness as the foundation for this research.

Peter Bludau is with fortiss, Research Institute of the Free State of Bavaria, Munich, Germany. E-mail: bludau@fortiss.org

Alexander Pretschner is with Technical University of Munich, Munich, Germany. E-mail: alexander.pretschner@tum.de

This article has a replication package available at https://doi.org/10.24433/CO.8051628.v1, provided by the authors.

TABLE I
STATE-OF-THE-ART FEATURES FOR JIT DEFECT PREDICTION

	Feature	Description
Size	LA	Number of lines added by the change
	LD	Number of lines deleted by the change
	LT	Number of lines in the files before the change
Diffusion	NS	Number of modified subsystems
	ND	Number of modified directories
	NF	Number of modified files
	ENT	Spread of modified lines across modified files
History	NUC	Number of unique changes to all modified files
	NDEV	Number of past developers modifying the changed files
	AGE	Time between last and current change to modified files
Experience	EXP	Number of prior developer/reviewer changes
	REXP	Number of prior developer changes weighted by the age of the respective change
	SEXP	Number of prior changes to modified subsystems that an developer has participated in
	AWARE	Proportion of prior changes that an developer has participated in
Review	ITER	Number of reviews that were performed
	NREV	Number of persons, reviewing the changes
	NCOM	Number of review comments
	AGEREV	Time between the review request and the final approval

A. Just-in-Time Defect Prediction

JIT defect prediction, first introduced by Mockus and Weiss [1], aims to support developers by predicting whether a new change contains defects and should be investigated. This has several benefits compared to traditional defect prediction, where defects are predicted for whole modules or releases. First, reviews can be conducted shortly after the commits are pushed to the remote repository while changes are still tangible [8]. In addition, predicting defects at the level of individual changes reduces the amount of code that needs to be investigated to the updated lines within the revised files. This leads to more targeted reviews [2]. These benefits align well with modern code reviews used in software development today [9], [10].

In JIT defect prediction, a classifier determines if a change is defective (binary classification). Different approaches have been proposed in various studies, using different sets of features extracted from the software development history, such as size or diffusion of the change, history, or workflow-based features (e.g., [1], [2], [6], [11]–[13]) to learn a defect prediction model. Most studies use the set of well-established features for comparing the performance of different model approaches shown in Table I. Recently, Bludau and Pretschner [6] published an extended set of features that additionally contains workflow information (e.g., the number of changes in a merge request or the time needed to get a feature merged) and syntactic change information extracted from the changes in the abstract syntax tree (breadth or depth of the change and the types of changed nodes). They report increased performance in their experiments compared to solely using the features from Table I. For the experiments conducted in this work, we use their data set and features as a baseline since the work presented in this paper is not concerned with increasing the performance of JIT defect prediction models in general but rather investigates more effort awareness and applicability.

Researchers commonly employ different machine learning techniques such as Logistic Regression [1], [2], [14], Support Vector Machines [12], Naïve Bayes [15], Decision Trees and Random Forests [14], [16], [17], or Neural Networks [18]–[21]. In recent publications [6], [22], [23], Random Forests models reported the best performance. The most direct approach to decide if a change should be flagged as defective is directly utilizing the defect prediction model output. If the defect probability of the model for a particular commit exceeds a threshold, then that commit warrants a review. For experiments conducted in this work, we use the Random Forest algorithm as it already demonstrated good results.

McIntosh and Kamei [24] investigated the evaluation procedures used in JIT defect prediction studies. They discovered that cross-validation techniques do not accurately represent the temporal nature of software development. Splitting training and test data based on random sampling, newer commits could be used to train a model subsequently evaluated on preceding commits. This can result in overly optimistic performance results. They also point out the need to re-train models regularly. They propose timely data set splitting to address this issue, keeping the order of commits. They introduced short-term splitting, where data from the past six months is used to train a defect prediction model to predict the defectiveness of commits from the subsequent six months. In the wake of their research, numerous studies have adopted this approach (e.g., [6], [18]).

B. Effort awareness

Implementing defect prediction in an industrial context, where developers only have a limited time budget, necessitates applying effort-aware techniques for quality assurance and verification tasks [2], [25]. Kamei et al. [2] propose to evaluate the performance of JIT defect prediction approaches in general and consider the developers' effort expended during the review of changes. The objective of effort-aware JIT defect prediction is to optimize the review process by finding the majority of defects while conserving effort. Given that developers have limited time and about 20% of the code contains 80% of defects [26], Kamei et al. [2] suggest that JIT defect prediction models should be evaluated with a focus on this fixed available budget. They choose 20% of the total effort of all commits as a threshold for the limited effort developers can spend on reviewing changes. This threshold has been subsequently adopted by other studies evaluating effort awareness (e.g., [3], [5], [6], [27]). Many recent JIT defect prediction studies have incorporated an effort-aware evaluation [7]. Various case studies have also featured effort-aware evaluations (e.g., [28]). However, there is doubt that effort-aware evaluations realistically reflect the goals of JIT defect prediction, especially timely reviews [7]. In the remainder of this section, we will describe how developer effort is defined, how current effort-aware evaluations work in detail, and what metrics are used to assess effort awareness.

1) Effort: To assess the effort awareness of JIT defect prediction approaches, a measure is needed that quantifies the required effort to inspect a change. Kamei et al. [2] introduce the total number of changed lines as an approximate value for the effort needed to inspect a change. The intuition is that a commit with a large number of added or deleted lines would typically require more time to inspect. To the best of our knowledge, existing effort-aware JIT defect prediction studies adopt this measure. Throughout this paper, we also use this definition and denote it as *effort(c)* where c represents the commit in question.

They further define defect density as the ratio of the model's confidence ($m_{probability}$) in a commit being defective to the effort required to investigate it. Defect density, therefore, is a measure to adjust the model output by the effort needed to investigate a change, weighing the effort of a review and the potential value of a review. Equation 1 describes defect density for a commit c and a specific defect prediction model m.

$$defect\_density(m, c) = \frac{m_{probability}(c)}{effort(c)} \tag{1}$$

Equation 1 shows the tendency of effort-aware techniques to increase the defect density of small commits since the most effort can be saved, neglecting large commits. However, this ignores the fact that large commits tend to be more error-prone [29].

2) Effort-aware risk score and ranking: Effort-aware JIT defect prediction assigns an effort-aware risk score for each commit c (Equation 2). The score represents the perceived need to review a particular commit, factoring in its potential for defects and the effort required to review it. Afterward, all commits C are ranked based on this score, determining the order of investigation for identifying most defects with minimal effort (Equation 3).

$$risk(m, c) = \text{risk score of } c \text{ where } c \in C \tag{2}$$

$$rank(c_i) = \sum_{j \neq i} [\![risk(m, c_j) < risk(m, c_i)]\!] \tag{3}$$

Different approaches in the literature propose different risk scores to rank the commits [2]–[4], [20], [30], [31]. These scores rank commits based on factors like their likelihood of introducing defects based on the defect prediction model [20], the density of defects in the changes [2], the required effort [32], or a combination of these aspects [4]. Additionally, several unsupervised approaches were proposed in the literature (e.g., [5], [32]). They are based on the observation that small commits do not produce considerable effort and can be investigated without huge impact. For example, Liu et al. propose ranking the commits solely based on the total changed lines (CHURN) [32], equivalent to the effort notion used in supervised approaches. Unsupervised approaches perform similarly to supervised techniques due to the tendency of effort-aware evaluations to penalize large commits in general.

To the best of our knowledge, the described risk scores represent the state-of-the-art used in recent effort-aware JIT defect prediction studies. They all rank commits and stop the review process after a certain inspection budget was spent. Yet, JIT defect prediction focuses on timely predictions and notifying developers and reviewers early, which renders current effort-aware evaluation impractical [7].

3) Evaluation metrics: In this Section, we describe existing effort-aware performance metrics used to evaluate JIT defect prediction approaches regarding their efficiency. Effort-aware evaluation metrics are calculated based on the risk score and their ranking (Equation 3). According to existing studies and based on Kamei et al. [2], the effort budget is defined as 20% of the total effort needed to inspect all commits.

The most prominent effort-aware metric is the recall at 20% of spent effort (*R@20%*). It is used in most studies [4] and reports the fraction of found defects investigating 20% of changed lines. It is often also called *PofB20* (e.g., [2]), *Recall@20%LOC* (e.g., [33]) or P_{effort} (e.g., [34]).

With a similar intention, studies often report P_{opt} representing the area between a perfect effort-recall curve based on a perfect oracle and the effort-recall curve based on the classifications of a defect prediction model. This measure removes a strict effort budget cut-off value (20%) since a different cut-off value may lead to different results [2]. A larger value implies a smaller discrepancy between the optimal and the model results.

Huang et al. recently proposed two additional metrics for evaluating effort awareness [4]. First, they introduce the proportion of reviewed commits when limiting to 20% total effort (PCI). This metric accounts for the problem of frequent context switches, where a lower PCI value is preferable. Second, they introduce the number of initial false positives before the first correctly classified defective commit is predicted (IFA). A high IFA may cause developer frustration, as early wrong predictions lead to wasted reviewing effort [35]. This metric is tailored towards avoiding false positive predictions as they would spend effort from the small inspection budget and lower *R@20%* significantly.

In current effort-aware evaluations, the emphasis is on finding most defects with a limited inspection budget, and the presented metrics mirror this fact. Effectiveness is measured by recall, and the efficiency of the approach is factored in by applying the budget constraint. We propose to focus on evaluating the efficiency without the need for a fixed inspection budget by focusing on the performance of the model in the negative class.

III. Limitations of current approaches

Effort-aware evaluations focus on optimizing risk scores and reducing early spent effort. A typical approach, as detailed in Section II-B, ranks commits based on a calculated risk score, proceeding with code reviews based on this ranking. When reaching the effort budget, the review process is stopped. Although this method is sound in general, it presents several limitations and contradictions when applied to real-world scenarios. To illustrate the problems, we analyze projects from a recently published data set [36]. It contains defect information on six large open-source projects. The examples and graphs shown in the following are taken from the evaluations of the Kafka[1] project. The data set contains all commits from the

[1] https://github.com/apache/kafka

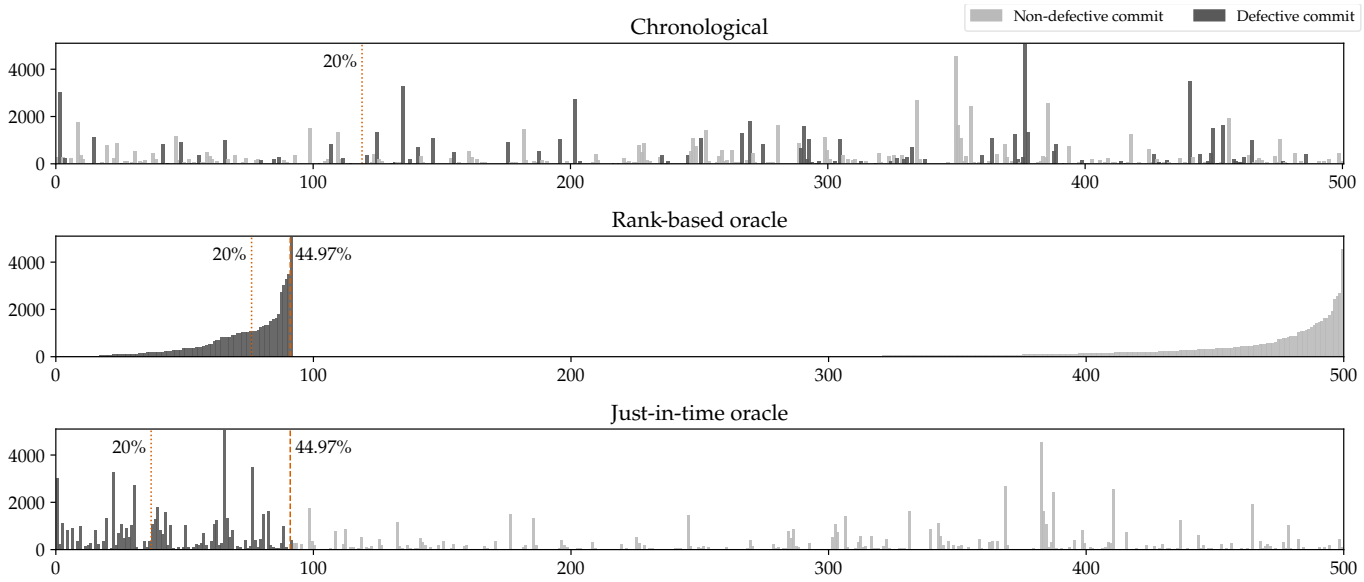

Fig. 1. Defect and effort distribution in the Kafka project. Each bar represents one commit with the height indicating the number of changed lines.

development history, their size, and a label that tells if the commit is considered bug-introducing. The evaluations for all other projects in this dataset show very similar results and can be found together with further analysis in the supplementary material provided alongside this work[2].

A. Ranking of commits

The most pressing issue of effort-aware evaluations is the missing time sensitivity. As pointed out by Çarka et al. [7], an immediate evaluation of each new commit is essential in JIT defect prediction. Therefore, the ranking of commits, a step that naturally delays evaluation, is neither practical nor realistic.

To illustrate this issue, we consider an oracle prediction model that predicts the correct class label for every commit. Figure 1 depicts three sub-graphs, each representing the first 500 commits (x-axis) in the development history of the Kafka project and their size, respectively, required review effort (y-axis). The black bars indicate defective commits, while the grey bars represent non-defective commits according to the defect data set. The upper graph chronologically orders the commits, the center graph represents a perfect *rank-based evaluation*, and the bottom subplot shows an perfect approach that keeps the chronological order of commits (*just-in-time evaluation*).

We see that large commits (high bars) are scattered across the whole commit history, while some are defective and some are not. In the *rank-based evaluation*, commits are ordered by label and effort, resulting in two distinct sections with increasing commit size. The dotted vertical line at 44.97% of effort depicts the total effort required to identify all defective commits. This is consistent in both the center and the bottom subplot, indicating that in a perfect world, both evaluation approaches would detect all defects investing 44.97% of the total effort. Committing effort beyond this line becomes unnecessary as all subsequent commits are non-defective and can be disregarded. In a realistic

[2]https://doi.org/10.24433/CO.8051628.v1

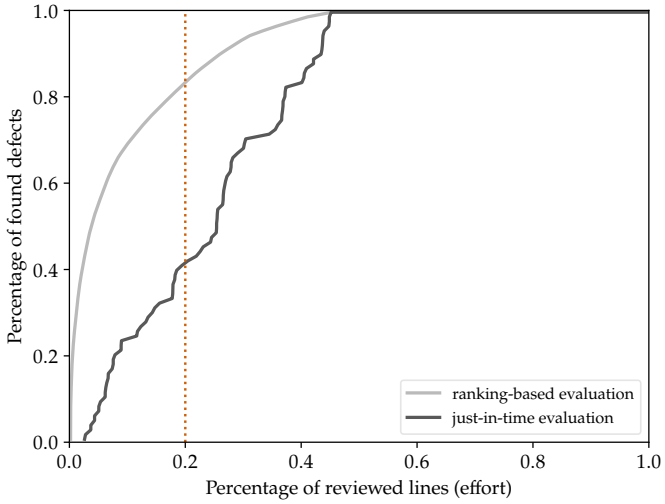

Fig. 2. Direct comparison of the effort-aware recall curve for both *rank-based evaluation* and *just-in-time evaluation* in the Kafka project.

JIT evaluation, however, large commits must be reviewed early. By ranking commits based on size, there is an overestimation of the number of defective commits that can be identified with minimal effort. Furthermore, by ranking commits, large defective commits are investigated later in the process; however, large commits tend to be more error-prone. This is also shown by the second dotted line at 20% of the effort. In the case of *rank-based evaluation*, more defective commits can be reviewed with 20% of effort than in the *just-in-time evaluation* setting, neglecting early large defective commits but having a huge effect on the applicability of the approach.

B. Effort budget

With a fixed budget of 20% of the total effort applied in today's research, not all defective commits can be investigated. Applying a timely evaluation, the number of defects that can be found using that budget is significantly lower, as depicted again

in Figure 2. Each line in Figure 2 illustrates the effort-aware recall curves represented by a grey line for the *rank-based evaluation* and a black line for the chronological *just-in-time evaluation*. The y-axis shows the percentage of found defects, and the x-axis shows the total effort percentage, indicating defects found per effort spent. The effort at 20% is indicated by the vertical orange dotted line.

The *just-in-time evaluation* curve rises more slowly, requiring more effort early on, and meets the post-development curve at 44.97% of effort. However, the area under the recall curve in the *just-in-time evaluation* setting is 14.3% smaller than in the *rank-based evaluation* setting, illustrating that more effort must be invested early and distributed more evenly, as large commits can happen at any time and need to be investigated directly afterward. This results in a lower recall at 20% of the effort. However, reviewing large commits should not be postponed in the development process just because they require more effort. This highlights a challenge in analyzing the applicability of JIT defect prediction models in an effort-aware context.

Ranking-based approaches give an illusion of efficient effort awareness, prioritizing commits that are easy to investigate but carry high risk. While this may reduce effort in the initial stages, it potentially skews the evaluation results. The inflated performance is particularly evident in unsupervised approaches [32], where low-effort commits are inspected first. Although this strategy identifies numerous defects, it neglects many defective commits in the early stages of development and results in a high rate of false negatives. Since the R@20% is the most used effort-aware metric in JIT defect prediction studies [4] and is often the only applied effort-aware metric [7], we find that the highlighted difference may have a significant impact on the validity of such studies and their applicability claim.

C. Modern code reviews

Current effort-aware evaluations in JIT defect prediction focus on minimizing effort due to budgetary constraints. The original purpose of JIT defect prediction - spotting defects early to prevent them from reaching production - is increasingly overshadowed by this emphasis on a limited budget. Current best practices in software development echo this disconnect. For instance, in open-source projects, it is standard to have all submitted code reviewed [37]. Modern code reviews are conducted in a lightweight process to have an effective bug detection process for every change [38], [39]. Nonetheless, most existing effort-aware studies operate under the assumption of a limited investigation budget. Except for P_{opt}, all prominent effort-aware evaluation metrics use the spent budget as a cut-off value for the investigation. We argue that with modern code reviews being a de-facto standard in software development, the applicable usage of the JIT defect prediction approaches needs to be re-evaluated.

IV. Approach

Given the limitations of effort-aware evaluations illustrated in Section III, we propose a new perspective on the evaluation of applicable JIT defect prediction to better align with real-world practices and objectives. Instead of focusing on developer frustration if too many commits are flagged by the defect prediction approach, the focus should be on conserving effort by thoroughly investigating defective commits while minimizing reviewing efforts for non-defective ones. To create realistic evaluations of JIT defect prediction studies, the ranking of commits based on an effort-aware risk score should be avoided. This includes the definition of a predefined inspection budget that can only be known in hindsight. Further, the evaluation metrics need to more clearly represent the potentially and correctly saved effort applying JIT defect prediction.

A. Timely evaluation

To allow a timely evaluation of JIT defect prediction approaches, the chronological order of commits needs to be ensured during the whole training and evaluation process. First, we advise constructing models only using past commits. The models should then be evaluated using only data from subsequent development activities, as suggested by McIntosh and Kamei [24]. This ensures that the models are evaluated in a temporally coherent manner. Thereby, genuine behavior during software development projects is imitated, maintaining the order of all commits and establishing a chronological sequence that includes the entire development history.

Current effort-aware approaches rank commits based on their effort-aware risk score. While this strategy might be justifiable from a post-project analytical perspective, without ranking commits we do not know which score value mandates a review. In standard JIT defect prediction, the model threshold is used to make the decision. A commit should be investigated if the model output exceeds a certain threshold. Such a threshold based on the effort-aware risk score is difficult to set and interpret. Moreover, with the current definition of effort as the sum of added and deleted lines, the effort is already used as a feature to train the defect prediction model, enhancing model performance significantly [6]. Therefore, the model decision is already implicitly based on effort. This leads us to question the necessity of making the decision function itself effort-aware, a step that may introduce redundancy and overlook other critical variables. Instead of making models effort-aware, we propose to craft evaluation metrics that incorporate the effort-saving perspective more judiciously. Thus, in our approach, we use the model thresholds to decide if a commit should be reviewed and evaluate the prediction based on its impact on saved effort.

B. Key performance indicator

The contingency table depicted in Figure 3 offers a comprehensive overview of potential outcomes within a classification task, particularly within the realm of effort-aware JIT defect prediction. Evaluating applicability of JIT defect prediction, the effectiveness - True Positive (TP) and False Negative (FN) segments - and efficiency - True Negatives (TN) and False Positives (FP) - need to be weighted and discussed separately. Further, the effort to review a commit varies based on the size of the commit and has an influence on the cost of unnecessarily investigated (FP) or missed defective (FN) commits. We elaborate on these considerations in the following.

	defective	non-defective
classified as defective review mandated	TP correctly spent effort	FP wasted effort
classified as non-defective review skipped	FN wrongly saved effort	TN saved effort

Fig. 3. Contingency table including effort notion.

1) Effectiveness and efficiency: When employing defect prediction to enhance the review process, the primary objective is to ensure that all defective commits are still identified and reviewed (TP), aiming for a high recall (R), as a not identified defect (FN) might increase debugging and fixing costs over time. This concept can be described as the **effectiveness** of the approach. To employ JIT defect prediction effectively, there should be minimal occurrences where defective commits evade review. Consequently, to ensure the effectiveness of JIT defect prediction, the number of unnecessarily investigated commits (FP) and hence precision (P) is not the major focus.

With effectiveness being the main concern, JIT defect prediction must optimize for saved effort to prove applicable. This concept can be described as **efficiency**. Without JIT defect prediction, all commits undergo review, leading to a maximum of unnecessarily investigated commits (FP). Applicable JIT defect prediction should strive to optimize the effort saved in the process by not reviewing non-defective commits (TN). To focus on efficiency and evaluate the potential to save effort, we evaluate the models' ability to correctly predict non-defective commits. To evaluate the precision and recall in the negative class, the negative predictive value (NPV) and the specificity (SPC) can be utilized. The NPV elucidates the fraction of accurate predictions out of all negative predictions. Drawing parallels to precision, NPV represents the precision in the negative class and, in our scenario, clarifies the proportion of correctly identified non-defective commits amongst all non-defective predictions. Thus, focusing on correctly saved effort while still reviewing most defective commits. Specificity is calculated as the fraction of TN out of all negative samples. It can, therefore, be interpreted as the recall in the negative class, telling how many of all real non-defective commits are identified by the model. Using specificity in the evaluation helps to show how many unnecessary reviews can be avoided by applying the model.

Current effort-aware performance metrics, as well as most metrics used to evaluate JIT defect prediction in general, such as precision, recall, or F-score, do not focus on TN samples and are more tailored to report on effectiveness. The Matthews Correlation Coefficient (MCC), which is increasingly being reported as a more balanced metric, considers all four elements of the contingency table but does not provide a fine-grained view of effectiveness and efficiency. We evaluate and discuss the proposed metrics in Section V-B. Our aim is to discern if the change in perspective indeed augments the applicability of JIT defect prediction.

2) Effort-based metrics: In general, performance metrics are based on the number of correct predictions and error occurrences. To highlight the effort, we propose to instead base the calculation of performance metrics on the review effort of each occurrence. Rather than counting TP, FP, FN, and TN occurrences, we substitute each occurrence count by its caused effort (sum of lines added and removed). For example, a commit with 100 changed lines is factored in as 100 instead of 1. While an equal effort distribution across all commits and both classes would render this distinction irrelevant, research indicates that larger commits tend to be more error-prone.

The resulting effort-based recall (R_{eff}) measures the correctly invested effort relative to the effort required to identify all defective commits. Therefore, identifying a large defective commit improves (R_{eff}) more than a small one. The intuition is that detecting larger defective commits is crucial, as they are more challenging to trace when causing production failures. Effort-based precision (P_{eff}) quantifies the wasted effort in reviewing commits. The cost of mistakenly investigating a small commit is not as significant as misjudging a large one. The effort-based NPV (NPV_{eff}) evaluates the fraction of the correctly saved effort. A missed large defective commit lowers NPV_{eff} more significantly than a missed smaller commit. Likewise, effort-based SPC (SPC_{eff}) assesses the fraction of correctly saved effort using the JIT defect prediction model from the total non-defective effort. A large reviewed non-defective commit negatively affects SPC_{eff} more than a small commit. In Section V-B, we reason which of these metrics helps to determine the applicability of JIT defect prediction.

C. Enhanced applicability

The question remains if current JIT defect prediction approaches can be applicable in practice. Various techniques could be applied to optimize the key performance indicators mentioned above to foster applicability. We, therefore, introduce the concept of a reliability threshold ($t_{reliability}$) for JIT defect prediction models that can be adjusted by developers to decide the balance between finding all defects (effectiveness) and saving effort (efficiency). This tuning parameter can be applied to take a more or less cautious stand and make the model predict more or fewer commits as defective. We propose two methods to augment applicability and assess their validity in Section V-C.

1) Threshold Tuning: To achieve an acceptable recall, a simple strategy is to directly adjust the model threshold and use it as the reliability threshold. By lowering the $t_{reliability}$, the model adopts a more cautious stance in predicting the negative class. Intuitively, a smaller threshold would lead to fewer FN, thereby enhancing recall. However, the repercussions of this adjustment on TN remains ambiguous. We evaluate if the rise in TP will be more rapid than the decline of FN in Section V-C.

2) Model Trust and Reliability: In domains like medicine, where the consequences of FN decisions can be exceedingly high, model reliability scores serve as a way to balance model predictions. These scores provide insights into the level of confidence a model holds regarding its predictions. One way to define such scores is by using the training data and describing how much the predicted label of a single sample is in agreement with the labels of nearby training samples. Multiple techniques

TABLE II
OVERVIEW OF USED DEFECT DATA SETS

Project	Commits	Inducing commits	Fixing commits
Airflow	7,311	1,582 (21.64%)	1,396 (19.09%)
Angular	16,050	2,545 (15.86%)	2,013 (12.54%)
Calcite	2,463	774 (31.43%)	1,182 (47.99%)
Jenkins	8,784	380 (4.33%)	750 (8.54%)
Kafka	5,812	1,377 (23.69%)	1,722 (29.63%)
Pulsar	4,127	1,016 (24.62%)	903 (21.88%)

exist to deduce reliability scores for a given prediction based on the proximity of training data in the feature space (e.g., [40]–[42]). For our evaluation, we use the trust score [41] as reliability scores. The trust score builds the ratio between the distance to the closest non-predicted class and the nearest predicted class training sample. It is a distance measure to the closest training sample with the same class. It has no upper bound, and the prediction is more trustworthy the higher the value. Therefore, it can be applied to existing models and does not need an adaption of the model architecture or training. If a model's trust score is lower than the predefined $t_{reliability}$, the commit gets reviewed regardless of the model prediction. Again, we investigate if applying the Trust Score will lead to a faster decline in FN compared to TN in Section V-C. Our aim is to discern whether it can be harnessed to modulate the frequency of FN and TN predictions during our evaluation, especially compared to the simpler threshold tuning approach.

V. RESULTS

We presented the drawbacks of current effort-aware approaches in Section III and proposed a novel perspective to evaluate JIT defect prediction in Section IV. In the following, we analyze the presented approach to reflect on its applicability and implications. We describe the study design in the subsequent section.

A. Study design

In order to answer the research questions posed in the Introduction of this work and to evaluate our approach, we use a recently published data set [36] as ground truth for defect-inducing commits. This data set contains defect information from six open-source projects, shown in Table II. The commits contained in the data set were created between January 2015 to January 2020. For every commit in the development history of these projects, the data set contains a binary label that tells if the commit introduced a defect. In order to account for the time-dependent nature of software development and changing environments, we build defect prediction models for each of the six projects based on a six-month short-term strategy as proposed by McIntosh et al. [24]. Thus, we do not use randomized cross-validation to learn and test the models but rather split the development history of every project into chunks of six months. Splitting the data in this fashion results in ten time frames for each project.

For each project and for every commit within each time frame, we collect the prediction features described in Section II-A. Using these features, a JIT defect prediction model is learned for every time frame and evaluated on the subsequent one. We train random forest models, as the classification algorithm performs best on the chosen data set [6]. To learn the random forest models, we use the standard implementation provided by scikit-learn[3]. We concatenate the results of every time frame to form one continuous and chronologically ordered development history per project that contains each commit, the respective defect label from the ground truth data set, and a defect-inducing probability based on the output of the defect prediction model as well as the effort needed to inspect the respective commit.

Based on this information, we calculate all discussed and proposed performance metrics for each project (**RQ1**) and analyze the results in Section V-B. To evaluate if the applicability could be increased using reliability measures (**RQ2**), we perform the proposed threshold tuning and trust score evaluation. To evaluate threshold tuning, we adjust the predicted label per commit based on the model output directly by applying different threshold levels to decide if a commit should be investigated. To calculate the trust score, we use the model training data from each time frame and calculate a trust score value for each commit from the subsequent 6-month time period. We report the results in Section V-C. Finally, we discuss implications and provide an outlook in Section V-D.

B. Performance metrics (RQ1)

In this section, we evaluate key metrics for applicable JIT defect prediction. The reported performance values are shown in Table III. For all metrics, we report the effort-based results in brackets. For example, the prevalence (PR) value is based on occurrence count, while the value inside parentheses is calculated using the effort of every instance.

Generally speaking, both the standard and the effort-based values show wide ranges across metrics and projects. As stated in existing research, this shows that JIT defect prediction performance highly varies based on the project context. The prevalence value shows the imbalance of the data set, with defective commits being the minority class in all investigated projects. Together with defect size (bracketed values of TP and FN), they significantly determine the reported performance. A low prevalence inherently deflates P and inflates NPV due to how these metrics are calculated [43]. Re-basing PR on the commit size always leads to an increased effort-aware PR, reflecting on the P_{eff} and NPV_{eff} values similarly. This shows that defective commits, by nature, are often larger than non-defective changes, a trend consistent across all projects but with varying magnitudes. It is evident that the model picks up this fact, and size/effort influences the model decision [6]. Large non-defective commits are often predicted as defective (FP), and small defective commits are predicted as non-defective (FN). FN commits are smaller than FP commits in all projects and, on average, span 120 compared to 301 lines. The number of FN commits is comparably low, and large FP commits lead to high wasted effort in the review process. Consequently, for every incorrect prediction made by the model, developers are

[3]https://scikit-learn.org/stable/

TABLE III
PERFORMANCE METRICS FOR THE DIFFERENT PROJECTS. EFFORT-BASED VALUES ARE SHOWN IN BRACKETS FOR EACH COLUMN.

Project	PR	TP	FN	TN	FP	R	P	NPV	SPC
Airflow	0.23 (0.83)	925 (252089)	359 (17141)	3308 (145179)	1090 (180863)	0.72 (0.94)	0.46 (0.58)	0.90 (0.89)	0.75 (0.45)
Angular	0.17 (0.70)	1734 (1148670)	460 (85052)	8052 (600002)	2928 (1152483)	0.79 (0.93)	0.37 (0.50)	0.95 (0.88)	0.73 (0.34)
Calcite	0.33 (2.06)	488 (342581)	142 (14903)	943 (61865)	333 (112385)	0.77 (0.96)	0.59 (0.75)	0.87 (0.81)	0.74 (0.36)
Jenkins	0.04 (0.10)	224 (67173)	102 (9814)	5507 (465476)	1441 (338288)	0.69 (0.87)	0.13 (0.17)	0.98 (0.98)	0.79 (0.58)
Kafka	0.25 (1.38)	1061 (722685)	195 (16652)	3099 (282234)	752 (254991)	0.84 (0.98)	0.59 (0.74)	0.94 (0.94)	0.80 (0.53)
Pulsar	0.26 (1.00)	743 (477429)	173 (35112)	1823 (270703)	724 (241779)	0.81 (0.93)	0.51 (0.66)	0.91 (0.89)	0.72 (0.53)

investing a substantial amount of effort in reviewing these sizeable FP commits and miss investigating these smaller FN commits. This inequitable distribution of effort has implications, especially for the efficiency of the review process and the overall cost of defect identification.

With this in mind, we investigate whether the additional performance metrics, reported in Table III, give meaningful insight to evaluate the effectiveness and efficiency of JIT defect prediction approaches. In Section IV, we propose recall as the main metric for effectiveness, as it describes the fraction of still identified defective commits applying JIT defect prediction. The recall ranges from 0.69 (Jenkins) to 0.84 (Kafka). However, the reported recall values might still be too low for applicable defect prediction. The results for R_{eff} show that wrongly saved small commits do not significantly affect the metric increasing R_{eff}; however, these commits could still introduce severe bugs. This renders R_{eff} impractical. Precision is low in all investigated projects, with a maximum value of 0.60 for Kafka. P_{eff} sharply rises compared to P as a small wrongly investigated commit (FP) is not as effort-intense as investigated large commits. This information supplements standard precision by indicating that when investigating small FP commits, their impact on the total expended effort is limited. P_{eff} might help to set precision into perspective; however, in applicable JIT defect prediction, P is not a primary concern. Reporting the F-Score in highly imbalanced data sets while focusing on recall and not precision leads inherently to lower values. Consequently, with this perspective the F1-Score has no additional value.

Looking at the efficiency of JIT defect prediction and the values of NPV and SPC, we see that effort can be saved and high numbers of TN commits can be detected. The NPV ranges from 0.87 (Calcite) to 0.98 (Jenkins), while SPC ranges from 0.72 (Pulsar) to 0.80 (Kafka). High NPV and SPC values in all projects show that the models perform well in the negative class. Most predicted non-defective commits are correctly classified (NPV), and many non-defective commits are detected (SPC). This may lead to applicable JIT defect prediction approaches in the outlined context. NPV_{eff} shows lower values because if a larger defective commit is not investigated, it has a higher influence on the metric than a smaller wrongly saved commit. The decrease in NPV_{eff} compared to NPV can be but is not always significant. The decline is more pronounced in projects with a high ratio of large defective commits. Missing already one large defective commit (FN) severely influences NPV_{eff}. SPC_{eff} shows that large commits have a high impact when wrongly predicted defective, lowering SPC_{eff} considerably. SPC_{eff} drops significantly in all projects, showing the model's tendency to predict large non-defective commits as defective, missing out on saveable effort.

Introducing new metrics has provided a new perspective on JIT defect prediction, emphasizing the effectiveness demonstrated through recall and the efficiency illustrated by NPC and SPC. These metrics offer insights into the negative class, highlighting potential effort savings. Beyond just offering an alternative viewpoint, the effort-aware metrics pave the way for evaluating the applicability of JIT defect prediction more proficiently. The effort-aware metrics generally provide additional insights but are tightly coupled with their counterparts and PR. That said, from economic aspects, SPC_{eff} sheds additional light on the extent of avoidable effort conserved. We conclude that recall, paired with NPV, SPC, and SPC_{eff}, constitute the cornerstone of performance metrics for investigating effort-saving defect prediction. Nevertheless, the reported results, particularly regarding effectiveness (R), challenge the feasibility of applying JIT defect prediction with current approaches. There is a pressing need to refine models for higher recall and assess whether meaningful effort savings remain achievable under such conditions.

C. Reliability measures (RQ2)

In this section, we investigate whether adapting the model threshold (*Threshold*) or calculating and applying a trust score (*Trust Score*) is a viable option to fine-tune JIT defect prediction model applicability.

As recall is the main metric for deciding if a model is effective enough, we base all further evaluations on this metric. First, we want to evaluate if *Threshold* and *Trust Score* can be used to tune recall in a consistent way. Figure 4 shows the reliability threshold $t_{reliability}$ (y-axis) for *Threshold* (orange) and *Trust Score* (blue) compared to the achieved recall (x-axis) for one exemplary project, namely Kafka. Looking at all other projects from the data set, a very similar picture emerges. The evaluations for the other projects are contained in the supplementary material provided alongside this work. Looking at Figure 4, we see that recall rises with decreasing model threshold. On the contrary, for *Trust Score*, we see that the recall rises with higher trust score values, as expected. Looking at these values, generally speaking, setting the $t_{reliability}$ to a lower *Threshold* or a higher *Trust Score* continuously leads to higher recall values. The *Threshold* curve falls more gradually, whereas the *Trust Score* curve has a more extreme slope at the lower and upper recall bounds and is not as straightforward to interpret. Moreover, the $t_{reliability}$ for model threshold has a fixed value range between 0 and 1, whereas trust scores do not

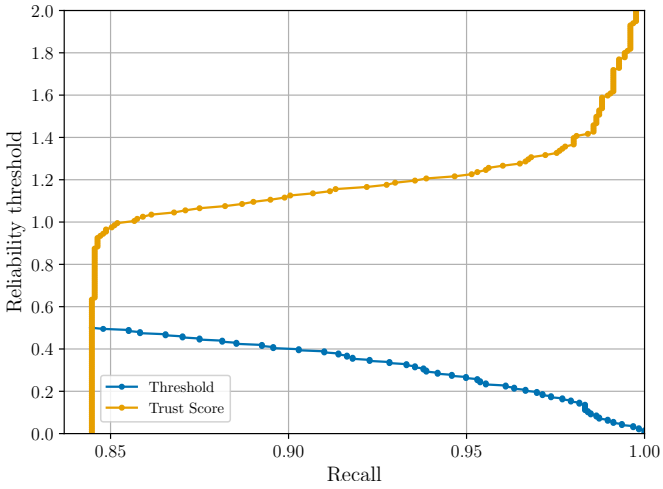

Fig. 4. Threshold and Trust Score values required to reach certain recall levels. Exemplary depicted for the Kafka project.

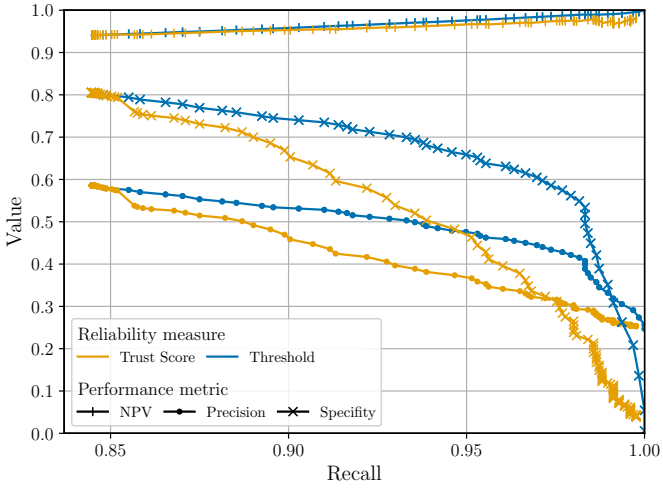

Fig. 5. Comparison of different performance metrics based on Threshold and Trust Score tuning. Exemplary depicted for the Kafka project.

have an upper bound, which makes setting the $t_{reliability}$ more ambiguous.

Since we now know that recall can be tuned using the two options, we evaluate the model performance regarding NPV, P, and SPC. The value for each metric is depicted on the y-axis depending on the tuned recall on the x-axis. Figure 5 shows results again for the Kafka project. Again, the results are very similar for all other projects and all evaluated performance metrics, including the effort-aware metrics. We can see that the NPV (marked as '|'), P (marked as '·'), and SPC (marked as '×') are higher tuning *Threshold* (blue) than they are for *Trust Score* (blue) for most recall values. This relationship breaks only in a few cases and at the bounds of the recall range (e.g., at recall levels higher than 0.99 for the Pulsar project). However, in these cases, SPC is always very low, which means that nearly no effort can be saved setting $t_{reliability}$ in such an extreme way. This shows that under the presumption of a certain recall level, the *Threshold*-tuned models do outperform *Trust Score*-tuned models in all important performance metrics. This leads us to conclude that in the case of JIT defect prediction and our

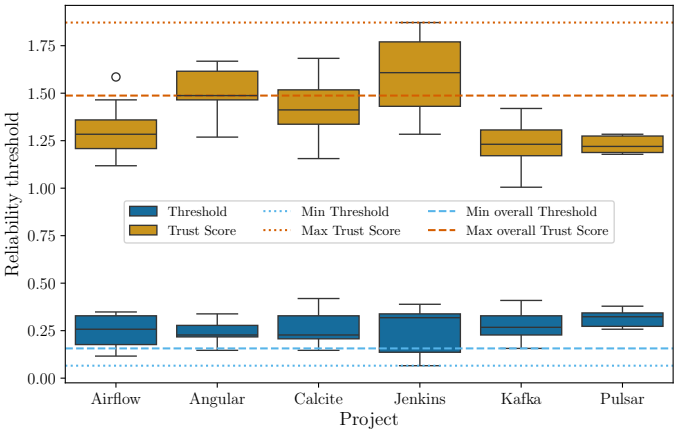

Fig. 6. Boxplot of $t_{reliability}$ for Threshold and Trust Score measures to maintain a recall of at least 0.95 in the different time frames by project.

evaluated models, the simpler *Threshold* based approach to fine-tune JIT defect prediction models performs better concerning the investigated key performance metrics.

For applicable JIT defect prediction, we want to optimize NPV and SPC while maintaining high recall values. To apply fine-tuned defect prediction models in an industrial context, a $t_{reliability}$ needs to be set by the user. The user would want to reach a certain level of security to find most defective commits, obtaining a high recall, by setting $t_{reliability}$. We investigate if the $t_{reliability}$ is stable across time frames (different model training periods) and if certain $t_{reliability}$ values can be set to ensure a certain recall. Figure 6 shows the $t_{reliability}$ that is necessary to reach a recall value of 0.95 in different time frames. We choose 0.95 as an example recall value that represents a high-security approach. The box size describes the variation in $t_{reliability}$ across the time frames for *Threshold* (blue) and *Trust Score* (orange). We see that for every project, there is variation in suitable $t_{reliability}$ between time frames. Furthermore, the $t_{reliability}$ from one time frame cannot be easily transferred to the subsequent time frames. Moreover, the optimal $t_{reliability}$ value varies between projects. Nonetheless, there are bounds for $t_{reliability}$ (dotted lines) that ensure a certain recall for all time frames and all projects. Looking at the full project lifespan and not a specific time frame, a $t_{reliability}$ (0.16 for *Threshold*, 1.49 for *Trust Score*) can be selected that on average maintains a 0.95 recall level for all projects over all time frames (dashed lines).

The above evaluations show that tuning the model threshold results in better effort-saving approaches and is easier to interpret than a *Trust Score*-based adaption. This is true for all calculated performance metrics and all investigated projects. One possible explanation for why the *Trust Score* performs worse is that the most important features learned by the defect prediction models tend to be related to the size of a commit [6]. If we have a large non-defective commit, the surrounding training data will be mostly defective, leading to low trust. Moreover, if we have a large defective commit, the model already has a high chance of correctly predicting. Because of the distribution of the data in the training space, the trust score does not hold additional valuable information that would help

TABLE IV
MEAN PERFORMANCE METRICS FOR DIFFERENT $T_{RELIABILITY}$ VALUES FOR THRESHOLD TUNING ACROSS PROJECTS. THE NUMBER IN BRACKETS REPRESENTS THE STANDARD DEVIATION FOR THE DIFFERENT PROJECTS.

$t_{reliability}$	TP	TN	FN	FP	R	NPV	SPC	SPC_{eff}
0.1	1088.33	1219.00	12.67	3781.00	0.99 (0.00)	0.99 (0.01)	0.27 (0.12)	0.09 (0.09)
0.2	1065.33	2035.00	35.67	2965.00	0.97 (0.01)	0.98 (0.01)	0.42 (0.10)	0.18 (0.11)
0.3	1025.83	2678.33	75.17	2321.67	0.93 (0.02)	0.97 (0.02)	0.54 (0.08)	0.26 (0.10)
0.4	965.50	3233.00	135.50	1767.00	0.87 (0.04)	0.95 (0.03)	0.65 (0.06)	0.35 (0.10)
0.5	862.50	3788.67	238.50	1211.33	0.77 (0.06)	0.93 (0.04)	0.76 (0.04)	0.46 (0.10)

with the reliability of the model.

D. Implications and Outlook

In the previous sections, we showed that additional metrics provide meaningful insights on the applicability of defect prediction models and that tuning these models can increase the number of identified defects while still saving effort. In the best-case scenario, where the $t_{reliability}$ is set optimally for every project to maintain a recall of 0.95, we find that, on average 97.25% (min: Calcite: 95.77%; max: Jenkins: 99.33%) of skipped commits are correctly skipped non-defective commits leading to 46.31% (min: Jenkins: 34.04%; max: Kafka: 65.20%) of the non-defective commits being correctly identified and, thus, on average 19.70% (min: Angular: 6.52%; max: Kafka: 39.49%) of the avoidable effort is saved.

Based on risk and security aspects, JIT defect prediction could be applied and adjusted in a risk-based manner. Developers can set $t_{reliability}$ based on their preferences and environment. However, knowing the optiomal $t_{reliability}$ for one's circumstances upfront is impossible. Choosing the $t_{reliability}$ based on past development activities is not stable. Nonetheless, a static $t_{reliability}$ over the whole project lifespan yields promising results for all investigated projects. Table IV shows the average performance values for different $t_{reliability}$ values, with the number in parentheses representing the standard deviation across projects. We can see that the recall is quite stable with a low standard deviation across all $t_{reliability}$-levels. With higher $t_{reliability}$, the standard deviation increases. Furthermore, a lower $t_{reliability}$ leads to higher recall (effectiveness) but also to lower NPV and SPC values (efficiency). The table also shows that setting a static $t_{reliability}$ value to a rather restrictive value of, for example, 0.2 guarantees a recall of at least 0.95 for all projects. We find that applying this $t_{reliability}$, while still reviewing 97% (min: Jenkins: 94%; max: Calcite: 97%) of defective commits on average 97.91% (min: Airflow: 96.58%; max: Jenkins: 99.38%) of skipped commits are correctly skipped non-defective commits leading to 42.27% (min: Airflow: 32.72%; max: Kafka: 60.50%) of the non-defective commits being correctly identified and, thus, on average 18.82% (min: Angular: 4.19%; max: Kafka: 36.54%) of the avoidable effort is saved. Setting a $t_{reliability}$ of 0.1, in all projects, 99% of all defective commits are identified while still being able to save reviews for, on average, 1219 non-defective commits.

In conclusion, the promise of JIT defect prediction lies in its effectiveness and the ability to directly integrate it into the development workflow. Using the presented approach and setting a $t_{reliability}$ can save effort under the condition of effectiveness (high recall). When considering current best practices, it is obvious that every TN means a reduction in developer workload. A trade-off between effectiveness and efficiency is visible and must be factored in depending on the specific context. Further research is needed to understand the relations in a broader context.

VI. DISCUSSION

This section discusses further limitations and assumptions that should be considered in future studies.

Presentation of results: One of the significant challenges for applicable JIT defect prediction is the potential backlash from developers when faced with incorrect predictions. As mentioned by Kochhar et al. [35], false predictions can lead to developer frustration, potentially diminishing trust. Therefore, the presentation of predictions is crucial. Instead of overwhelming developers with a barrage of warnings, JIT defect prediction tools might be more effective if they point out commits that are likely safe to ignore. This way, developers can prioritize review efforts without getting distracted. Furthermore, integrating explainability techniques, as, for example, done by Khanan et al. [44], can enhance the acceptance. When developers can understand the reasoning behind a prediction, they are more likely to trust and act upon it.

Notion of effort: The approximation of effort as the number of changed lines is a popular measure used in many software engineering studies. The rationale seems straightforward: the more changed lines of code, the higher the effort needed from the developer. However, this definition may not capture the full spectrum of developer effort. For instance, a complex algorithmic change might require a high cognitive load but might only result in a few changed lines. Conversely, simple refactoring or boilerplate code can change a large number of lines with relatively low reviewing effort. Another dimension is the latent effort associated with defective commits. When defects are not detected early and linger until later development stages, the cost of the investigation and rectification process increases rapidly [45]. For example, a defect detected during production puts pressure on the development team due to the operational urgency and demands effort to trace back and comprehend the origin, especially if a significant time has elapsed since its introduction. More research is needed to fully understand the intricate nature of the needed effort.

Review process: Current JIT defect prediction approaches assume that every review finds all defective code. Yet, it is more likely that the success of a review is influenced by its thoroughness. A shallow code review might miss subtle defects,

whereas a thorough, effort-intense review should have a higher likelihood of uncovering defects. This trade-off is central to understanding the actual effort spent in the review process. In JIT defect prediction, review thoroughness varies. No review means no chance of finding defects. A standard review has a variable chance of identifying mistakes, which might shift if guided by predictions. Apart from review thoroughness, the reviewer's experience and code complexity also influence review efficacy. These factors are crucial for understanding how defect prediction methods perform in practical situations and should be considered in future research.

VII. Threats to Validity

Threats to **internal validity** may arise due to faulty assumptions in the evaluation process. We carefully evaluated the data set and its characteristics to ensure that the shown limitations are valid for all projects. We propose to apply well-established metrics and use standard libraries to compute them. We publish all code and evaluations in the supplementary material to provide reproducible results and to mitigate this risk.

Threats to **external validity** relate to the generalizability of the study results. We used six open-source projects, covering five years of development and varying in size. The projects belong to different application domains and contain different programming languages [36]. Nonetheless, the data set may be incomplete. This may affect the accuracy of our experimental results. Furthermore, we evaluate our proposed approach on a limited number of six projects. Since the data is collected from open-source projects, we are not able to provide insights into the transferability of the results to industrial projects. To generalize the results, further studies analyzing more projects and investigating the application in industrial contexts are needed.

Threats to **construct validity** refer to the soundness and suitability of our study design and evaluation. The time-dependent validation of models leads to data set splits that are not randomized to be more realistic. We evaluate all scenarios for all projects and compare their results in order to provide a complete picture. However, due to the splitting of the data set, characteristics of commits may determine the results of singular time periods more directly.

VIII. Conclusion

In this study, we discussed the limitations of current effort-aware JIT defect prediction and made a case for focusing on a more realistic evaluation setting. We described key metrics to evaluate the effort-saving potential and introduced reliability approaches to adapt model predictions. We evaluated the approaches in the context of six projects from a publicly available data set.

We found that a combination of recall, negative predictive value, and specificity, as well as effort-based specificity, are valid measures to evaluate the effort-saving potential of JIT defect prediction models. Moreover, adapting the model threshold is a viable option to make defect prediction more applicable in practice by ensuring high levels of recall while still helping developers save effort. However, the threshold needs to be set and adapted manually depending on the project and its environment. We evaluated different reliability levels that can be used as a first reference to set a suitable threshold.

In summary, this work describes a shift in perspective to enable applicable JIT defect prediction that is effective and efficient and contributes to the ongoing exploration of JIT defect prediction approaches. In future, the results of this study need to be evaluated on a larger set of study projects and validated in industry. Moreover, the evaluation of JIT defect prediction studies should focus on closely mirroring realistic software development processes. In this context, the considerations set out in section VI can provide valuable impetus.

References

[1] A. Mockus and D. M. Weiss, "Predicting risk of software changes," *Bell Labs Technical Journal*, vol. 5, no. 2, pp. 169–180, Aug. 2002. Available: https://doi.org/10.1002/bltj.2229

[2] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi, "A large-scale empirical study of just-in-time quality assurance," *IEEE Transactions on Software Engineering*, vol. 39, no. 6, pp. 757–773, Jun. 2013. Available: https://doi.org/10.1109/tse.2012.70

[3] Q. Huang, X. Xia, and D. Lo, "Supervised vs unsupervised models: A holistic look at effort-aware just-in-time defect prediction," in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, Sep. 2017.

[4] ——, "Revisiting supervised and unsupervised models for effort-aware just-in-time defect prediction," *Empirical Software Engineering*, vol. 24, no. 5, pp. 2823–2862, Oct. 2019. Available: https://doi.org/10.1007/s10664-018-9661-2

[5] Y. Yang, Y. Zhou, J. Liu, Y. Zhao, H. Lu, L. Xu, B. Xu, and H. Leung, "Effort-aware just-in-time defect prediction: simple unsupervised models could be better than supervised models," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, Nov. 2016. Available: https://doi.org/10.1145/2950290.2950353

[6] P. Bludau and A. Pretschner, "Feature sets in just-in-time defect prediction: An empirical evaluation," in *Proceedings of the 18th International Conference on Predictive Models and Data Analytics in Software Engineering*, ser. PROMISE 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 22–31. Available: https://doi.org/10.1145/3558489.3559068

[7] J. Çarka, M. Esposito, and D. Falessi, "On effort-aware metrics for defect prediction," *Empirical Software Engineering*, vol. 27, pp. 1–38, 11 2022. Available: https://link.springer.com/article/10.1007/s10664-022-10186-7

[8] E. Shihab, A. E. Hassan, B. Adams, and Z. M. Jiang, "An industrial study on the risk of software changes," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering - FSE '12*. ACM Press, 2012. Available: https://doi.org/10.1145/2393596.2393670

[9] P. C. Rigby and C. Bird, "Convergent contemporary software peer review practices," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013. New York, NY, USA: Association for Computing Machinery, 2013, p. 202–212. Available: https://doi.org/10.1145/2491411.2491444

[10] C. Sadowski, E. Söderberg, L. Church, M. Sipko, and A. Bacchelli, "Modern code review: A case study at google," in *International Conference on Software Engineering, Software Engineering in Practice track (ICSE SEIP)*, 2018.

[11] L. Aversano, L. Cerulo, and C. D. Grosso, "Learning from bug-introducing changes to prevent fault prone code," in *Ninth international workshop on Principles of software evolution in conjunction with the 6th ESEC/FSE joint meeting - IWPSE '07*. ACM Press, 2007. Available: https://doi.org/10.1145/1294948.1294954

[12] S. Kim, E. J. Whitehead, and Y. Zhang, "Classifying software changes: Clean or buggy?" *IEEE Transactions on Software Engineering*, vol. 34, no. 2, pp. 181–196, Mar. 2008. Available: https://doi.org/10.1109/tse.2007.70773

[13] O. Kononenko, O. Baysal, L. Guerrouj, Y. Cao, and M. W. Godfrey, "Investigating code review quality: Do people and participation matter?" in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, Sep. 2015. Available: https://doi.org/10.1109/icsm.2015.7332457

[14] Y. Kamei, T. Fukushima, S. McIntosh, K. Yamashita, N. Ubayashi, and A. E. Hassan, "Studying just-in-time defect prediction using cross-project models," *Empirical Software Engineering*, vol. 21, pp. 2072–2106, 10 2016, from Duplicate 1 (Studying just-in-time defect prediction using cross-project models - Kamei, Yasutaka; Fukushima, Takafumi; McIntosh, Shane; Yamashita, Kazuhiro; Ubayashi, Naoyasu; Hassan, Ahmed E.)ACM. Available: http://link.springer.com/10.1007/s10664-015-9400-x

[15] T. Jiang, L. Tan, and S. Kim, "Personalized defect prediction," in *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '13. IEEE Press, 2013, p. 279–289. Available: https://doi.org/10.1109/ASE.2013.6693087

[16] C. Pornprasit, C. Tantithamthavorn, J. Jiarpakdee, M. Fu, and P. Thongtanunam, "Pyexplainer: Explaining the predictions of just-in-time defect models," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2021, pp. 407–418.

[17] T. Fukushima, Y. Kamei, S. McIntosh, K. Yamashita, and N. Ubayashi, "An empirical study of just-in-time defect prediction using cross-project models," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, ser. MSR 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 172–181. Available: https://doi.org/10.1145/2597073.2597075

[18] T. Hoang, H. K. Dam, Y. Kamei, D. Lo, and N. Ubayashi, "DeepJIT: An end-to-end deep learning framework for just-in-time defect prediction," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, May 2019. Available: https://doi.org/10.1109/msr.2019.00016

[19] T. Hoang, H. J. Kang, D. Lo, and J. Lawall, "Cc2vec: Distributed representations of code changes," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 518–529. Available: https://doi.org/10.1145/3377811.3380361

[20] X. Yang, D. Lo, X. Xia, Y. Zhang, and J. Sun, "Deep learning for just-in-time defect prediction," in *2015 IEEE International Conference on Software Quality, Reliability and Security*. IEEE, Aug. 2015.

[21] Z. Zeng, Y. Zhang, H. Zhang, and L. Zhang, "Deep just-in-time defect prediction: how far are we?" in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, Jul. 2021.

[22] D. Bowes, T. Hall, and J. Petrić, "Software defect prediction: do different classifiers find the same defects?" *Software Quality Journal*, vol. 26, no. 2, pp. 525–552, Feb. 2017. Available: https://doi.org/10.1007/s11219-016-9353-3

[23] H. Osman, M. Ghafari, O. Nierstrasz, and M. Lungu, "An extensive analysis of efficient bug prediction configurations," *Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering - PROMISE*, 2017.

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

[25] E. Arisholm, L. C. Briand, and E. B. Johannessen, "A systematic and comprehensive investigation of methods to build and evaluate fault prediction models," *Journal of Systems and Software*, vol. 83, no. 1, pp. 2–17, Jan. 2010.

[26] T. Ostrand, E. Weyuker, and R. Bell, "Predicting the location and number of faults in large software systems," *IEEE Transactions on Software Engineering*, vol. 31, no. 4, pp. 340–355, Apr. 2005. Available: https://doi.org/10.1109/tse.2005.49

[27] L. Qiao and Y. Wang, "Effort-aware and just-in-time defect prediction with neural network," *PLOS ONE*, vol. 14, no. 2, pp. 1–19, 02 2019. Available: https://doi.org/10.1371/journal.pone.0211359

[28] M. Yan, X. Xia, Y. Fan, D. Lo, A. E. Hassan, and X. Zhang, "Effort-aware just-in-time defect identification in practice: a case study at alibaba," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, Nov. 2020. Available: https://doi.org/10.1145/3368089.3417048

[29] R. Moser, W. Pedrycz, and G. Succi, "A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction," in *Proceedings of the 13th international conference on Software engineering - ICSE '08*. ACM Press, 2008. Available: https://doi.org/10.1145/1368088.1368114

[30] W. Fu and T. Menzies, "Revisiting unsupervised learning for defect prediction," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, Aug. 2017. Available: https://doi.org/10.1145/3106237.3106257

[31] W. Li, W. Zhang, X. Jia, and Z. Huang, "Effort-aware semi-supervised just-in-time defect prediction," *Information and Software Technology*, vol. 126, p. 106364, 2020. Available: https://www.sciencedirect.com/science/article/pii/S0950584920301324

[32] J. Liu, Y. Zhou, Y. Yang, H. Lu, and B. Xu, "Code churn: A neglected metric in effort-aware just-in-time defect prediction," in *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, Nov. 2017. Available: https://doi.org/10.1109/esem.2017.8

[33] C. Pornprasit and C. K. Tantithamthavorn, "Jitline: A simpler, better, faster, finer-grained just-in-time defect prediction," *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pp. 369–379, 5 2021. Available: https://ieeexplore.ieee.org/document/9463103/

[34] M. D'Ambros, M. Lanza, and R. Robbes, "Evaluating defect prediction approaches: a benchmark and an extensive comparison," *Empirical Software Engineering*, vol. 17, pp. 531–577, 8 2012.

[35] P. S. Kochhar, X. Xia, D. Lo, and S. Li, "Practitioners' expectations on automated fault localization," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 165–176. Available: https://doi.org/10.1145/2931037.2931051

[36] P. Bludau and A. Pretschner, "Pr-szz: How pull requests can support the tracing of defects in software repositories," in *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2022, pp. 1–12.

[37] A. Bacchelli and C. Bird, "Expectations, outcomes, and challenges of modern code review," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. IEEE Press, 2013, p. 712–721.

[38] S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan, "The impact of code review coverage and code review participation on software quality: A case study of the qt, vtk, and itk projects," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, ser. MSR 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 192–201. Available: https://doi.org/10.1145/2597073.2597076

[39] P. S. Kochhar, T. F. Bissyandé, D. Lo, and L. Jiang, "An empirical study of adoption of software testing in open source projects," in *2013 13th International Conference on Quality Software*, 2013, pp. 103–112.

[40] J. Leonard, M. Kramer, and L. Ungar, "A neural network architecture that computes its own reliability," *Computers & Chemical Engineering*, vol. 16, no. 9, pp. 819–835, 1992, an International Journal of Computer Applications in Chemical Engineering. Available: https://www.sciencedirect.com/science/article/pii/0098135492800358

[41] H. Jiang, B. Kim, M. Guan, and M. Gupta, "To trust or not to trust a classifier," in *Advances in Neural Information Processing Systems*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds., vol. 31. Curran Associates, Inc., 2018. Available: https://proceedings.neurips.cc/paper_files/paper/2018/file/7180cffd6a8e829dacfc2a31b3f72ece-Paper.pdf

[42] P. Schulam and S. Saria, "Can you trust this prediction? auditing pointwise reliability after learning," in *The 22nd International Conference on Artificial Intelligence and Statistics, AISTATS 2019, 16-18 April 2019, Naha, Okinawa, Japan*, ser. Proceedings of Machine Learning Research, K. Chaudhuri and M. Sugiyama, Eds., vol. 89. PMLR, 2019, pp. 1022–1031. Available: http://proceedings.mlr.press/v89/schulam19a.html

[43] A. Luque, A. Carrasco, A. Martín, and A. de las Heras, "The impact of class imbalance in classification performance metrics based on the binary confusion matrix," *Pattern Recognition*, vol. 91, pp. 216–231, 2019. Available: https://www.sciencedirect.com/science/article/pii/S0031320319300950

[44] C. Khanan, W. Luewichana, K. Pruktharathikoon, J. Jiarpakdee, C. Tantithamthavorn, M. Choetkiertikul, C. Ragkhitwetsagul, and T. Sunetnanta, "Jitbot: An explainable just-in-time defect prediction bot," in *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2020, pp. 1336–1339.

[45] B. W. Boehm. Software engineering economics. *IEEE Transactions on Software Engineering*, SE-10(1):4–21, 1984. Available: http://dx.doi.org/10.1109/TSE.1984.5010193