Enhancing Just-In-Time Defect Prediction Models with Developer-Centric Features

Emanuela Guglielmi, Andrea D'Aguanno, Rocco Oliveto, and Simone Scalabrino
DEVISER @ University of Molise, Italy

***Abstract*—Ensuring high software quality in development cycles with frequent updates is critical, especially in Agile and CI/CD environments. Just-In-Time Software Defect Prediction (JIT-SDP) has emerged as a promising solution for finding bugs early, as it enables immediate identification of changes prone to defects. JIT-SPD models based on Machine Learning focus primarily on project- and change-specific features, such as number of lines added and number of files modified in the change. Recent research has started to investigate developer-related features for defect prediction. However, these studies overlook information about developers' work habits and cross-project activities. In this paper, we try to fill this gap by introducing a set of developer-centric features for JIT-SDP, which span through temporal aspects (when do developers usually make commits?), change-related aspects (how do developers usually make commits?), and project-related aspects (how are the contributions distributed among different repositories?). We conducted an empirical evaluation to understand if such features allow to improve ML-based JIT-SPD models and evaluated the importance of developer-centric features on the performance of the model. Our results show that integrating developer-centric features improves model performance. We observed a +15.48% precision and +10.47% recall in a within-project evaluation and +14.59% precision and even +85.83% recall in cross-project evaluation.**

***Index Terms*—Just-in-Time Software Defect Prediction, Developer Centric, Mining software repositories**

I. Introduction

In software system development, maintaining high quality while keeping up with the frequent iterative changes during software maintenance and evolution is a crucial challenge [1]. This becomes especially important in Agile and CI/CD environments, where changes to the codebase are frequent and might automatically trigger releases, thus requiring an immediate evaluation of a given change in terms of quality [2]. In software maintenance and evolution developers frequently integrate their code into shared repositories, triggering testing pipelines to detect issues early. In this process, defects— *i.e.,* deviations from specifications or expectations — might be introduced and can cause malfunctions and security issues, with potentially harmful consequences for end-users. Some defects may remain latent and surface under specific conditions, leading to system failures [3]. To tackle this problem, previous work defined approaches to automatically predict to what extent a given component is defective (Software Defect Prediction, or SDP in short). SDP aims to automatically identify potentially defective code early, providing developers with timely feedback to prevent problems before they impact the end user [1]. Given a source code artifact (*e.g.,* a file/class or a function/method), SDP consists in predicting if it is *buggy* or *non-buggy*. In time, SDP research has gained significant attention, particularly thanks to the advancements in machine learning (ML) techniques [2], [4]. Traditional approaches for SDP are based on supervised Machine Learning (ML) models. Specifically, approaches focus on product-, process- and developer-related features, such as CK metrics [5], entropy of changes [6], and scattering [7].

Just-In-Time Software Defect Prediction (JIT-SDP) [8] has emerged as a more effective application of the concepts behind SDP. Instead of predicting the faultiness of software artifacts at a given revision, JIT-SDP models aim at identifying *changes* (*i.e.,* commits) that might lead to the introduction of bugs. CI/CD pipelines could be configured to hold the release of a new version when possible bug-introducing changes are detected and developers could be forced to review such changes.

Many JIT-SDP techniques have been proposed and evaluated in recent years [9]–[12]. Some of them are based on traditional ML techniques (such as Naive Bayes and Random Forest) [13], [14], while more recent ones explored the use of Deep Learning and Large Language Models as well [15]–[18]. However, recent surveys by Zhao *et al.* [2] and Alnagi *et al.* [4] have shown that the first ones (Random Forest models, specifically) tend to outperform other methods. These studies also highlight limitations in current approaches: JIT-SDP models typically rely on features that are extracted based on the information available for the specific project at hand, such as the number of lines added or the number of the specific change or the number of commits made by the developer (*i.e.,* their experience). Such project-centric aspects are certainly important, but neglect the fact that developers contribute to other projects as well. Developer experience, habits, and workload distribution might play a crucial role in error introduction as well [19]–[23]. For instance, if developers work on many projects in the same period, they may disperse attention and increase the risk of mistakes, while focusing on a single project may reduce this risk [24].

Developer-related information are kept into account in SDP models. Di Nucci *et al.* [7] introduced features aimed at capturing developers' focused activity areas and showed that they allow to improve existing models. However, these metrics are constrained to the specific project under analysis, capturing a narrow view of developer behavior without accounting for cross-project activity or long-term work patterns. Laudato *et al.* [23] recently explored the potential of using developers' activities to predict the likelihood that they introduce bugs.

However, measuring such features requires that developers use specialized equipment while working, which might not always be possible or accepted. Nevertheless, to the best of our knowledge, all the available developer-related features collected are still project-centric.

In this paper, we aim to provide a complementary point of view by introducing and evaluating a new set of developer-centric features for JIT-SDP, *i.e.,* metrics that can be used to fully characterize developers based on their near-past commits. The concept behind all such metrics is simple: Given a commit, we retrieve the activities performed by the developers in the near-past period (*e.g.,* 2 months) and we extract useful information from them throughout *all* the project they worked on. We identified three categories of aspects of interest: (i) *temporal* aspects, which are related to *when* developers worked (*e.g.,* did they work during the night or on Sundays?); (ii) *change*-related aspects, which capture how typical commits made by the developer look like (*e.g.,* did they make many commits?); (iii) project-related aspects, which regard how the contribution of the developer is spread across multiple projects (*e.g.,* did they focus on a single project or did they work on many projects?).

We conducted an empirical study to assess the effectiveness of our developer-centric features in JIT-SDP. We compared a model with classic project-centric JIT-SPD features from the literature and a model that combines such features with our newly introduced features. We performed such evaluations both in a within-project and in a cross-project evaluation scenario. Our results show that integrating developer-centric features improves model performance. We observed that up to +15.48% precision can be achieved in a within-project evaluation and even +85.83% recall and +4.85% AUC in a cross-project evaluation. We observed that features aimed at capturing temporal aspects are generally more important than others. For example, we observed that a higher percentage of work in the afternoon is associated with a higher buggy rated.

Our findings provide a clear message for future researchers interested in this field: Developer-related information is important for predicting bugs, and the context of the project might be insufficient to fully characterize them. Besides, our findings pave the way for models that only rely on developer-specific information to assess the risk that a given developer will introduce bugs *before* they start working (Before Time Defect Prediction), that could help better allocate developers' time in the future.

II. Related Work

We discuss the related literature focusing on JIT-SDP models, and developer-related features.

A. JIT-SDP Model

Developing various methods to build JIT-SDP (Just-In-Time Software Defect Prediction) models is a relevant aspect already explored in the literature [8], [25]. Mockus *et al.* [25] introduced one of the first studies to explore the possibility of leveraging system changes for detecting software defects. The authors developed an early supervised model for failure prediction based on the concept of identifying software defects by analyzing the changes.

Kamei *et al.* [8] formalize the concept of "Just-In-Time" defect prediction, highlighting its distinction from traditional SDP (Software Defect Prediction) approaches, which typically predict defects at the code artifact level (*e.g.,* file or package) [6], [26]–[31]. Instead, JIT-SDP focuses on continuous quality control, identifying potentially defective changes immediately after a commit, enabling timely interventions.

As a result of the introduction of JIT-SDP models in the literature, studies have been introduced to explore various algorithms in order to improve their accuracy [2], [4]. As described by Zhao et al. [2] and later by Alnagi *et al.* [4], the Random Forest classifier algorithm currently exhibits the best performance compared to Logistic Regression and Naive Bayes models in the literature [13], [14]. Fukushima *et al.* [32] conducted a study on predicting cross-project defects with Random Forest models. Their findings showed that model performance improves significantly when trained on accurately selected data from projects with similar correlations between predictors and variables. Kamei *et al.* [33] confirmed these findings, also observing that Random Forest ensemble models excel in cross-project contexts with accurate data, particularly when historical project data are limited. Tourani *et al.* [34] proposed a new JIT-SDP model that integrates traditional features with software release discussions. The authors show that the addition of contextual information improves defect prediction. Recently, Zeng *et al.* [35], explored how adding more features to datasets impacts the improvement of JIT-SDP predictive performance. The results show that it is important to consider the context of features as adding more features to datasets does not necessarily lead to improvements.

B. Features in JIT-SDP

The features considered in ML models play a crucial role in Just-In-Time Defect Prediction by identifying and quantifying code changes that may introduce defects. The state-of-the-art features for defect prediction are presented in Table I and are geared to take into account the main aspects that affect the life cycle of a software project [2], [8]. As highlighted by Alnagi *et al.* [4], commonly used JIT-SDP metrics focus on aspects like code diffusion, history, modification, developer activity, and purpose. *Diffusion* refers to how widely changes are distributed across different modules or components of the project. *History* involves the modification history, including quantitative and temporal data, as well as the frequency and extent of past changes. *Code modification* encompasses actual changes to the source code, such as additions, deletions, or rewrites [8], [36]. The *Developer* aspect focuses on the individual making the change, particularly their experience with the specific subsystem. Lastly, the *purpose* behind the change is analyzed, such as whether it was made to fix a bug. In fact, for each change which defines a row within the dataset, a series of numerical values are measured [37], [38].

Feature	Description
NS	Number of subsystems modified by the change
ND	Number of sub-directories modified by the change
NF	Number of files modified by the change
Entropy	Distribution across the modified files
LA	Lines of code added
LD	Lines of code deleted
LT	Lines of code in total
AGE	Average time gap to the last change over files
NUC	Number of unique last changes to the files
NDEV	Number of developer
EXP	Developer experience, i.e. number of changes
REXP	Recent developer experience, i.e. weighted by ages
SEXP	Developer experience over a subsystem
FIX	Whether the change is for fixing a previous defect

TABLE I: Description of the features used in the literature.

Issue Tracking System data and static analysis metrics have become emerging areas of interest in Just-In-Time Software Defect Prediction (JIT-SDP). Data from such systems typically include both textual content and metadata from software development tools like GitHub Issues, which tracks issue reports, code reviews, and developer discussions. These data sources capture rich information on how defects are discussed, prioritized, and resolved within a project, providing insights into the development process [34], [39].

Ning Li *et al.* [41] evaluated state-of-the-art JIT methods from a software reliability perspective. The authors start selecting several projects and calculated a set of features to train 11 of the most commonly used JIT-SDP models. They focused on two key aspects: the effectiveness of JIT techniques in preventing early exposed defects and the improvement in long-term software reliability resulting from these techniques. The dataset was built entirely on the state-of-the-art features described in Table I.

In recent years, JIT-SDP studies have incorporated developer-related metrics to capture human factors associated with defect introduction [22], [46]. Eyolfson *et al.* [46] showed that developers with more experience introduce fewer errors, while Bird et al. [22] pointed out that strong code ownership reduces bug rates. Posnett *et al.* [24] highlighted that developers focused on specific areas of a system introduce fewer defects. Di Nucci *et al.* [7] proposed a model that considers developers' focused activity areas, reporting superior results in defect prediction compared to traditional models. In JIT-SDP, developer experience metrics—often approximated by the number of changes a developer has made—have been used based on the assumption that more experienced developers are less likely to introduce defects [19]–[21]. However, these metrics are typically limited to the context of the project at hand. They capture elements of the developer's immediate development process, such as code changes or commit frequency. This is the main difference with our work, in which we consider the overall near-past history of contribution of a developer, also in other projects

III. Defining and Mining Developer-Centric Features

We discuss in the following the developer-centric features we devised to capture developer-related information and integrate them into JIT-SDP models. In particular, we detail in the following: (i) the set of features and the theoretical motivation behind them, and (ii) the mining procedure required for extracting computing them.

A. Developer-centric Features

All the features we devised are meant to be computed based on the global near-past contribution by the developer. Specifically, given a developer *d*, author of the commit c_t for which the JIT-SDP prediction needs to be performed, the features are computed based on the sequence of commits C_r by the same author made in *all* the repositories $r \in R$ they work on in the *X* days that come before the day in which c is performed. We discuss the *X* we adopt for our experiment in the experimental design. Each commit $c \in C_r \forall r \in R$ is associated to the push event $p(c)$ that contained it, *i.e.,* the repository synchronization request between the local repository of the developer and the central one. Note that each push event contains one or more commits and has its own creation time (might differ from the one of the commits it includes).

We identified three categories of features. The first one regards *temporal* aspects of the contributions made by the developer. The second one regards aspects concerning the specific *change* performed. Finally, the third one regards aspects related to the *projects* modified. We provide a detailed description of the categories and of the specific features below. Table II provides an overview the categories and features.

Note that, in the following descriptions, we always exclude from C_r duplicate commits, *i.e.,* commits with the same ID, which typically appear when developers first contribute to forks and then to the original repository.

1) Temporal Aspects: Time related aspects are characteristics that concern the part of the day and the weekday in which a developer works. Previous work has shown that such aspects could be important for predicting the performance of a developer in a task [23], [46]. Thus, we hypothesize that such aspects might be useful to characterize a developer in terms of their work habits.

***WD*: Distribution of work by day of the week.** First, we want to understand what is the typical working week of d. We consider this aspect as important based on the assumption that work performed on some weekdays (*e.g.,* during the weekends, or even on Fridays [44]) may be correlated with higher defect rates due to factors such as developer fatigue or busier working routines. Thus, we compute features aimed at counting the percentage of near-past contributions made on each weekday. To do this, we define seven features, one for each weekday (*i.e.,* WD_{Mon}, ..., WD_{Sun}) Given the commit timestamps of all the near-past commits ($c \in C_r \forall r \in R$), we count the the number of times each weekday is represented. Finally, we normalize the values of all the *WD* features, so that they are between 0 and 1 and their sum equals 1.

***HD*: Hourly distribution of commits.** The time at which a developer makes changes to the code can affect the probability of introducing errors. Working on certain hours, particularly those associated with fatigue or prolonged work sessions (*e.g.,* night hours), may be associated with an increased risk of introducing errors [45], [46]. To compute those features, we follow a similar approach as the one described for *WD*. We group the working hours into four slots: from 8AM to 2PM (morning, HD_{mo}), from 2PM to 6PM (afternoon, HD_{an}), from 6PM to 11PM (evening, HD_{ev}), and from 11PM to 8AM (night, HD_{ni}). Again, given the commits in C_r, we divide them among the four slots we identified based on their time and, then, normalize the values of such features so that they are between 0 and 1 and their sum is 1.

***PT*: Time between push event.** An extended time interval between successive push events can suggest that either (i) the developers' working activity is lower, which may, in turn, be a symptom of reduced familiarity with the codebases, or (ii) the developer is accumulating changes to be pushed at a later time. In both the cases, we hypothesize that higher time intervals are associated with a higher risk of introducing bugs. To compute the PT_{avg} and PT_{sd} features, we trace each commit $c \in C_r \forall r \in R$ back to its push event $p(c)$. We then remove duplicate push events. Finally, we sort them based on the push event time and compute, for each couple of consecutive push events, the time difference. PT_{avg} is the mean of such intervals, while PT_{sd} is their standard deviation.

2) Change-related Features: Change-related features allow to characterize the developer in terms of how and how much they work. As for the former, we measure the characteristics of code changes and commit messages. As for the latter, we aim at measuring the workload of the developer.

***WL*: Developers' Workload.** A too high workload could be associated with stress and a higher risk of introducing faulty changes. Thus, we aim at measuring different facets of such an aspect. First, we measure the total number of commits made in the whole period considered, WL_{tot}. Besides that, we introduce two additional workload measures: The average workload per day, WL_{ad}, which is computed as the mean number of commits made on each day of the period considered, and the average workload per working day, WL_{wd}, which only considers commits done between Monday and Friday (it ignores commits done in the weekend). The rationale is that such commits might indicate actual work made by developers, while the others might indicate leisure coding activities which do not necessarily increase the developers' stress.

***WAI*: Recent workload increases.** An increased workload in the days leading up to a commit may reflect heightened developer commitment, which can correlate with increased stress. So, we aim at computing the ratio between the developers' workload in the very recent past and a baseline workload for the same developer. Here we measure workload in terms of number of commits. To compute *WAI*, we divide the commits $c \in C_r \forall r \in R$ in two sets: C_{recent}, containing the commits done in the last week of the period considered, and C_{old}, containing the ones dated before. We further divide the commits in C_{old} based on the week in which they have been made and compute the average weekly number of commits. Finally, we compute *WAI* as the number of commits in C_{recent} and the average weekly number of commits.

***Acc*: Commit Accumulation.** Some developers tend to synchronize their local repository with the central one as soon as possible, while others might prefer to accumulate several changes before doing that. While we conjecture such a piece of information allows to better characterize the developer, we are agnostic in terms of expectation of what strategy is more prone to introduce bugs. It could be that accumulating more commits indicates that the developer is more meticulous and wants to provide a good solution before synchronizing. Conversely, commit accumulation may indicate a developer who does not always pay attention to share their changes. To compute Acc, we trace each commit $c \in C_r \forall r \in R$ back to the push event in which it is contained ($p(c)$), remove duplicates, and define the set of push events P. Finally, we compute Acc as $\frac{|C_r|}{|P|}$.

***Mod*: Modifications in the Commit.** Extensive code modifications, such as adding or removing a large number of lines (LOC) or modifying multiple files, can increase the likelihood of introducing defects by adding complexity to a software component. These changes can also complicate debugging and code review, making it more difficult to detect problems. To capture these issues, we defined four metrics: Mod_F, Mod_{L+}, Mod_{L-}, and Mod_{LF}. Such metrics are inspired by state-of-the-art JIT-SPD metrics that aim at measuring the size of a given change. In this case, however, we perform the measure on the near-past commits. Specifically, Mod_F computes the average number of files modified per commit, Mod_{L+} and Mod_{L-} measure the average number of lines added and removed to a commit, respectively, while Mod_{LF}measures the average number of modified lines (either added or removed) for each file.

***MCML*: Average length of commit messages** Detailed commit messages should be preferred over short and shallow ones. Thus, writing more detailed commit messages might indicate that a developer is more meticulous. To measure to what extent the developer tends to write detailed commit messages, we compute the average commit message length for all the commits $c \in C_r \forall r \in R$. Specifically, we count the number of characters by excluding whitespaces to have a (even though slightly) more precise measure of the actual content of the messages.

3) Project-related Features: We conjecture that the repositories on which a developer works tell much about their working habits. If a developer works on several projects at the same time, it is more likely that they forget important details and thus introduce bugs.

***#Repos*: Number of Repositories.** First of all, we want to estimate the number of projects on which the developer worked in the near-past period. We conjecture that the higher this number, the more likely it is that they introduce bugs. To measure *#Repos*, we simply count the number of repositories R the developer is working on. Note that this does not necessarily matches the number of projects since a project might be contained in several repositories (*e.g.,* backend and frontend), but we believe it provides a sufficiently good estimate.

***LOF*: Developers' Focus on a Project** While a developer might work on several projects, it is possible that they put most of the effort on a few of them. To better estimate to what extent the developer is focusing on a project, we compute two focus metrics, *i.e.,* *LOF* (Lack of Focus) and LOF_N (Normalized Focus). Such metrics are based on the concept of information entropy, which has been previously adopted in defect prediction research [6]. We associate each repository $r \in R$ with the number of commits made by the developer in it, $|C_r|$. Then, we compute *LOF* using the following formula:

$$LOF = \sum_{r \in R} \frac{|C_r|}{T} \log_2(\frac{|C_r|}{T})$$

where T is the total number of commits, computed as $T = \sum_{r \in R} |C_r|$. A high *LOF* suggests that the developer distributed their time evenly across the projects, which could lead to a dispersion of focus and increase the likelihood of errors due to frequent context switches. In contrast, low entropy implies that the developer concentrates on a smaller number of projects, reducing the risk of errors by maintaining focus and continuity. We introduce LOF_N as a normalized version of *LOF*, computed as *LOF* multiplied by T (total number of commits). Let us consider two developers, A and B. A contributed to two repositories, with 100 commits each, while B contributed to two repositories as well, but with 2 commits each. *LOF* would be 1 for both, but the focus of A is probably lower than the focus of B because A made more changes (and, possibly, more switches between the projects). In the example, LOF_N would be 100 for A and 4 for B.

***#CS*: Number of Context Switches.** Let us consider two scenarios. In the first one, a developer worked on three projects in a sequential way (first they worked on P_1, then on P_2, and then on P_3). In the second scenario, instead, the developer alternates changes to a project with changes to another project. The *LOF* and LOF_N would be the same in both scenarios if the number of commits made for the three projects is the same. However, we conjecture that the second scenario is more problematic since it is more likely that the developer might confuse information from more projects and introduce errors. To capture this phenomenon, we introduce an additional metric specifically aimed at measuring the number of context switches, *#CS*. Given the ordered sequence of commits $c \in C_r \forall r \in R$, we count the number of consecutive couples of commits, (c_i, c_{i+1}) that belong to different repositories.

B. Mining Developer-Centric Features

Given a commit c for which we want to predict its possible faultiness, we first extract the commit author email, that we use as an author identifier. Then, we rely on GitHub Archive [47] for identifying *all* the contributions made by a developer throughout different projects. GitHub Archive allows to download all the events happened on GitHub at a given date/hour. Thus, given the commit date of c, ideally, it could be possible to mine all the events happened between the time of c and the X past days to define the near-past window used to compute the features (we explain shortly below how). Note that, however, this operation is computationally intensive: It would be necessary to re-compute the window after every commit to re-compute the features. To avoid this, we decided to use a fixed near-past window for each month. Given two commits c_1 and c_2 of the same developer in the same month but on different days, the near-past window of both will be the same.

	Feature	Description
🕒	*HD*	Hourly distribution of commits created
	WD	Distribution of work by weekday
	PT_{avg}	Average time between push events
	PT_{sd}	Standard deviation of time between push events
🔧	WL_{tot}	Total workload of the developer
	WL_{ad}	Average workload per day
	WL_{wd}	Average workload per working day
	WAI	Workload increase in the last week
	Acc	Average number of commits per push event
	Mod_F	Average number of files modified in the period
	Mod_L	Average number of LOCs added (+) or removed (-)
	Mod_{LF}	Average number of modified lines per file
	MCML	Average length of commit messages
📅	*#Repos*	Number of repos to which it contributed in a given period
	LOF	Percentage of time devoted to one project compared to others
	LOF_N	Weighted entropy on commits
	#CS	Number of times a developer switched projects

TABLE II: Developer-centric temporal (🕒), change-related (🔧), and project-related features (📅).

Out of all the events in the near-past window, we consider only the push events, which contain, among other data, the repository ID and the IDs and messages of the commits pushed in the event. Such information are not sufficient to compute some metrics (*e.g.,* temporal ones) because we do not have the commit times. To acquire the missing information, we use the GitHub APIs.

IV. Empirical Study Design

The *goal* of our study is to assess to what extent our developer-centric features improve the effectiveness of a JIT-SPD model based on ML techniques. We aim at answering the following research questions:

- $\textbf{RQ}_1$**:** *To what extent do developer-centric features improve the effectiveness of JIT-SPD in a* ***whitin-project*** *evaluation?*
- $\textbf{RQ}_2$**:** *To what extent do developer-centric features improve the effectiveness of JIT-SPD in a* ***cross-project*** *evaluation?*
- $\textbf{RQ}_3$**:** *What developer-centric features are important?*

A. Study Context

In this section, we report how we built the datasets used to answer all our RQs.

The datasets we used in our study are based on the datasets released by Tian *et al.* [41]. The authors built a dataset for each of the 18 projects they considered. Each dataset contains the commit of the projects, and each commit is characterized by several state-of-the-art JIT-SPD metrics and annotated with a *buggy* or *non-buggy* label.

Project Name	total commit	filtred commit	buggy commits
ActiveMQ	10,236	1,722	85
Ant*	14,539	1,124	0
Camel	38,909	18,381	830
Derby*	8,268	195	2
Geronimo*	13,137	0	0
HBase	16,726	5,493	854
JMeter	16,341	2,623	35
LOG4J2	10,695	6,086	242
LUCENE	32,230	8,056	184
OpenJPA*	4,893	173	2

TABLE III: Number of commits for each project in the original dataset. * indicates the project that we excluded for which we did not have a sufficient number of commits.

As a first step, we removed from each dataset all commits for which we could not compute developer-oriented features. Specifically, we excluded the commits (i) made before March 2015 (GitHub Archive provide data in a different format before such a date), and (ii) no longer available in the repository at the time we conducted our study. Table III reports the number of commits for each project in the original dataset and the one we considered in our dataset. We excluded projects for which we did not have a sufficient number of commits. As a result, we considered six projects. For each valid commit of such projects, we used the previously-defined approach to compute the developer-centric features for each project.

In the end, we considered a total of 42,361 commits, of which only 2,230 have a "buggy" label (∼5.6%).

B. *Experimental procedure*

To address the first two research questions, we compared two models. One containing *all* the features (both state-of-the-art JIT-SPD features and our new developer-centric features — the new model) and one containing only state-of-the-art features (the baseline). We trained and tested three different ML classification algorithms. Random Forest [48], (ii) ADABoost + LMT [49], [50], and (iii) Naive Bayes [51]. We used the implementations and default configurations available in Weka [52]. We chose them because Zhao *et al.* [2] identified these models as top performers in Just-In-Time Software Defect Prediction tasks.

For both the models, we run a preprocessing step before starting the training. First, to reduce the number of features, we adopt attribute selection using the Info Gain algorithm [53]. Specifically, we filter out features that provide 0 gain. The datasets we considered are highly unbalanced: As it is reasonable to expect, the vast majority of changes (94.4% throughout all the datasets) do not introduce bugs. Thus, before training each model, we use SMOTE (Synthetic Minority Over-sampling Technique) [54], an oversampling method which creates synthetic samples from the minority class. We only generate synthetic instances for the training set while leaving the test set intact.

We are mainly interested in assessing the ability of the model to find bugs. Thus, we compute and report recall, precision, and F1-score (the harmonic mean of precision and recall) for the buggy label. We also report the AUC (Area Under the ROC curve [55]), which is independent from the class and globally evaluates the worth of the classifier: An AUC of 0.5 indicates a model that has no capability of distinguishing between the two classes. A perfect model, which has no false positives and no false negatives, has, instead, an AUC = 1.0.

To answer RQ$_1$, we used a 10-fold cross validation on each dataset independently (within-project evaluation). Such a validation consists of randomly dividing the dataset into ten equally sized folds and using, in turn, nine folds for training and one for testing. To answer RQ$_2$, instead, we considered, in turn, the whole dataset referred to the project under test as test set, and the union of all the datasets referred to the other projects as training set (cross-project evaluation).

Finally, to answer *RQ3*, we analyze the developer-centric features selected by the Info Gain algorithm [53] for each project in both the evaluation scenarios. Specifically, we count how many times each features is selected and report them. When several features are available for capturing the same aspect, we count them only once (if at least one of them is selected). An example is "WD", which computes the distribution of work by day of the week. There are seven features for measuring such an aspect (one for each day). If at least one of them is selected for a given project, we consider "WD" important for that project.

V. Empirical Study Results

This section reports the results of the three research questions formulated in Section IV.

A. *RQ$_1$: Within-Project Evaluation*

Table IV shows the comparison between the two approaches (combined, which includes developer-centric features, and the state-of-the-art baseline) for the three ML algorithms.

First, we observed slightly mixed results for the Naive Bayes algorithm in terms of the comparison between the two approaches (*i.e.,* combined and baseline). Even though the combined model achieves slightly better results in terms of recall (+5.6%) and AUC (+0.5%), it does that at the cost of a slightly lower precision (-3.6%). In general, the models based on Naive Bayes exhibit a clearly lower precision than the other models (based on AdaBoostM1+LMT and RandomForest). Still, the AUC values show a relatively high discriminative ability, with an average of 0.736 for the combined approach.

On the other hand, we observed that for the other ML algorithms (AdaBoostM1+LMT and RandomForest) the model trained on the combined set of features reported sensible improvements in terms of all the metrics for all the projects, except for one. More specifically, for the AdaBoostM1+LMT model we observed an average improvement for all the metrics (+20.8% precision, +19.5% recall, +20.4% F1-score, and +2.5% AUC). We obtained the best overall results with the Random Forest algorithm. Also for this algorithm, we observed that the combined model achieves better results than the baseline for all the project, with the same exception we had for the other two algorithm. Also, the baseline model achieves higher recall only for one of the projects (Lucene).

Naive Bayes								
	Combined				Baseline			
Project	**Precision**	**Recall**	**F1**	**AUC**	**Precision**	**Recall**	**F1**	**AUC**
ActiveMQ	**0.067**	0.918	**0.125**	**0.780**	0.062	0.918	0.115	0.755
Camel	0.130	**0.584**	0.212	0.771	**0.148**	0.434	**0.221**	**0.776**
HBase	0.461	**0.278**	**0.346**	0.762	**0.464**	0.262	0.335	**0.779**
Jmeter	0.014	0.886	0.028	0.616	0.014	0.886	0.028	0.616
Log4j2	**0.049**	0.806	**0.093**	**0.693**	0.047	**0.847**	0.089	0.662
Lucene	0.079	**0.587**	0.139	0.794	**0.092**	0.500	0.155	**0.801**
Mean	0.133	**0.677**	0.157	**0.736**	**0.138**	0.641	0.157	0.732
AdaBoostM1+LMT								
	Combined				Baseline			
Project	**Precision**	**Recall**	**F1**	**AUC**	**Precision**	**Recall**	**F1**	**AUC**
ActiveMQ	**0.355**	**0.447**	**0.396**	**0.847**	0.301	0.329	0.315	0.827
Camel	**0.422**	**0.457**	**0.439**	**0.873**	0.364	0.447	0.401	0.863
HBase	**0.413**	**0.467**	**0.438**	**0.766**	0.396	0.429	0.412	0.753
Jmeter	0.143	0.229	0.176	0.700	0.143	0.229	0.176	0.700
Log4j2	**0.475**	**0.579**	**0.521**	**0.900**	0.288	0.372	0.324	0.839
Lucene	**0.177**	**0.283**	**0.218**	**0.832**	0.151	0.250	0.189	0.817
Mean	**0.331**	**0.410**	**0.365**	**0.820**	0.274	0.343	0.303	0.800
Random Forest								
	Combined				Baseline			
Project	**Precision**	**Recall**	**F1**	**AUC**	**Precision**	**Recall**	**F1**	**AUC**
ActiveMQ	**0.352**	**0.435**	**0.389**	**0.884**	0.351	0.400	0.374	0.863
Camel	**0.416**	**0.496**	**0.453**	**0.909**	0.402	0.495	0.444	0.895
HBase	**0.493**	**0.468**	**0.480**	**0.813**	0.464	0.424	0.443	0.800
Jmeter	0.127	0.200	0.156	0.755	0.127	0.200	0.156	0.755
Log4j2	**0.562**	**0.545**	**0.553**	**0.910**	0.350	0.355	0.352	0.870
Lucene	**0.196**	0.261	**0.224**	**0.867**	0.168	**0.304**	0.216	0.848
Mean	**0.358**	**0.401**	**0.376**	**0.856**	0.310	0.363	0.331	0.839

TABLE IV: RQ_1: Comparison in a within-project evaluation.

Naive Bayes								
	Combined				Baseline			
Project	**Precision**	**Recall**	**F1**	**AUC**	**Precision**	**Recall**	**F1**	**AUC**
ActiveMQ	0.169	**0.388**	**0.236**	0.763	**0.173**	0.282	0.214	**0.783**
Camel	0.056	**0.818**	0.105	0.653	**0.144**	0.337	**0.202**	**0.783**
HBase	0.163	**0.856**	0.274	0.671	**0.469**	0.379	**0.420**	**0.789**
Jmeter	0.047	0.143	0.070	0.653	**0.054**	**0.167**	**0.081**	**0.744**
Log4J2	0.063	**0.277**	**0.102**	0.525	**0.065**	0.196	0.098	**0.621**
Lucene	0.026	**0.614**	0.050	0.602	**0.066**	0.200	**0.100**	**0.624**
Mean	0.087	**0.516**	0.140	0.645	**0.162**	0.260	**0.186**	**0.724**
AdaBoostM1 + LMT								
	Combined				Baseline			
Project	**Precision**	**Recall**	**F1**	**AUC**	**Precision**	**Recall**	**F1**	**AUC**
ActiveMQ	**0.240**	**0.294**	**0.265**	**0.796**	0.157	0.224	0.184	0.745
Camel	**0.142**	**0.208**	**0.169**	**0.730**	0.132	0.128	0.130	0.691
HBase	0.397	**0.199**	**0.265**	0.706	**0.427**	0.170	0.243	0.706
Jmeter	0.024	0.086	0.038	**0.658**	**0.056**	**0.144**	**0.080**	0.640
Log4J2	**0.103**	**0.202**	**0.136**	0.632	0.096	0.140	0.114	**0.655**
Lucene	0.070	**0.522**	**0.123**	**0.798**	**0.087**	0.132	0.105	0.615
Mean	**0.163**	**0.252**	**0.166**	**0.720**	0.159	0.156	0.143	0.675
Random Forest								
	Combined				Baseline			
Project	**Precision**	**Recall**	**F1**	**AUC**	**Precision**	**Recall**	**F1**	**AUC**
ActiveMQ	**0.301**	**0.294**	**0.298**	**0.849**	0.228	0.212	0.220	0.795
Camel	**0.191**	**0.153**	**0.170**	**0.764**	0.186	0.124	0.149	0.727
HBase	**0.530**	**0.189**	**0.278**	**0.774**	0.500	0.144	0.224	0.768
Jmeter	**0.051**	0.057	0.054	0.652	0.043	**0.078**	**0.055**	**0.722**
Log4J2	**0.098**	**0.182**	**0.128**	**0.659**	0.072	0.100	0.084	0.658
Lucene	**0.099**	**0.543**	**0.167**	**0.846**	0.080	0.104	0.091	0.662
Mean	**0.212**	**0.236**	**0.183**	**0.757**	0.185	0.127	0.137	0.722

TABLE V: RQ_2: Comparison in a cross-project evaluation.

On average, again, we observed a consistent improvement in terms of all the metrics we considered (+15.5% precision, +10.5% recall, +13.6% F1-score, and +2.0% AUC).

As previously mentioned, there is a project (Jmeter) for which both the approaches achieve exactly the same results. This happens because the feature selection algorithm selected none of the developer-centric metrics for such a project (*i.e.,* the same sets of metrics are adopted for the combined and baseline models). We report some possible motivations in the analysis performed in RQ_3.

Answer to RQ_1. The integration of developer-centered features consistently enhances the performance of the ML models we studied. For the best-performing algorithm (Random Forest) we obtained substantial improvements (+15.5% precision, +10.5% recall, +13.6% F1-score, and +2.0% AUC).

B. *RQ_2: Cross-Project Evaluation*

Table V shows the comparison between the combined approach (which includes developer-centric features) and the state-of-the-art baseline for the three ML algorithms in a cross-project evaluation.

As for Naive Bayes, we observed, again, mixed results. Integrating developer-centric metrics generally has a negative effect on such an ML algorithm in terms of precision (-46.3%), F1 (-24.3%), and AUC (-10.1%). On the other hand, we observed that the combined model achieves a drastically higher recall for all the projects except for one (+98%).

Also for the cross-project evaluation, the results obtained with AdaBoostM1+LMT and Random Forest are positive for the combined model. The developer-centric features, on average, allow to substantially improve precision (+2.5% and +14.6%, respectively), recall (+61.5% and +85.8%), F1 (+16.1% and +33.6%), and AUC (+6.7% and +4.8%). This results are consistent throughout the projects, with the only exception of one of them (again, Jmeter).

The best overall results for the combined features can be achieved with Random Forest, while, this time, the best results for the baseline features is obtained with the Naive Bayes method. When comparing such models, the combined model still allows to achieve better results in terms of precision (+30.9%) and AUC (+5.6%), at the cost of a slightly reduced recall and F1 score (-9.2% and -1.6%, respectively).

As for the Jmeter project, which constitutes an exception in this evaluation as well. Again, we report some possible motivations in the analysis performed in RQ_3.

Answer to RQ_2. Developer-centric features allow to obtain the JIT-SPD model with the highest average discriminative power (AUC = 0.757 with Random Forest), which also achieves higher precision than the best baseline model (+30.9% with Naive Bayes). However, such a model suffers from slightly lower recall and F1 score.

C. *RQ_3: Feature Importance*

Table VI shows the prevalence of developer-centric features in both evaluations. It can be observed that a higher number of developers involved in a project appears to be associated with a higher percentage of selected developer-centric features (τ = 0.33).

Project	% Developer-Centric Features	# Authors
ActiveMQ	33.0%	102
Camel	68.0%	738
Hbase	20.0%	443
Jmeter	0.0%	24
Log4j2	70.0%	103
Lucene	61.2%	204

TABLE VI: Prevalence of developer-centric features selected for each project in the within project evaluation, alongside the number of unique contributors.

For example, in the model for the Camel project (738 contributors), 68% of the features are developer-centric, while even 70% of developer-centric metrics constitute the features for the model built for Log4j2 (103 contributors). It is worth noting that the results of RQ$_1$ show that the combined models for such projects achieve the highest AUC scores (> 0.9). This suggests that developer-centric features play an important role in JIT-SDP.

On the other hand, as previously noted in RQ$_1$ and RQ$_2$, we can observe that out of the six projects analyzed, the only one that did not use any of our features was Jmeter. This is not only the smallest dataset in our sample, but also the one that contains the lowest number of contributors (only 24). This result indicates that developer-centric metrics might mostly benefit projects with a higher number of contributors. This result is quite expected: If a few developers work on a project, studying their peculiarities does help the model characterize them. When several developers are working on a project, instead, the model could find out interesting relationships between the data and the probability of introducing bugs (like the ones we discuss below).

Table VII shows that all added developer-centric features were selected at least once, indicating that each macro-category contains informative features. However, some features appear to be more important than others. We discuss below the further analyses we conducted based on the categories of features.

Temporal Features. Metrics concerning the hourly and daily distribution of a developer's work appear to be very important. As for the former, we report in Fig. 1 and Fig. 2 the distribution of the percentage of time the developer worked in the morning (between 8AM and 2PM) and in the afternoon (between 2PM and 6PM), respectively, divided by the two classes. It is clear that having worked mainly in the morning in the near past is associated with a lower risk of introducing bugs, while having worked in the afternoon is associated with a higher risk of making buggy changes. This aligns with literature in other field, *e.g.,* medicine [56], [57].

Change-related Features. The most informative feature is MCML, which computes the Mean Commit Message Length. This finding aligns with what Zhao *et al.* [2] observed. The authors highlighted that features related to change messages are among the most commonly used and informative in training a JIT-SDP model. However, it is important to note a key difference here. Zhao *et al.* [2] referred to the specific commit message in the code changes that aims at being predicted. On the other hand, MCML captures the commit message writing *habits* of the developers, even if in a shallow way (*i.e.,* it does not take into account the actual content of the message). Among the features related to commit content, another one frequently selected is WL_{ad}, which measures the number of commits of the author in the near past. Fig. 3 reports a comparison of the distribution of WL_{ad} for the two classes (buggy and non-buggy). While the median value remains nearly unchanged, we observe that buggy commits tend to come from authors with lower variance in the number of commits. This might be due to two aspects. First, developers wrote longer commit messages in the near past are used to do this, which indicates that they are more meticulous and, thus, have lower risks of introducing bugs. Second, developers that in the near-past wrote less accurate (thus shorter) commit messages could have done that because they did not have enough time (*e.g.,* increased working pressure) and, thus, this causes a higher risk of introducing errors. Conversely, it is worth noting that the total number of commits made by a developer in the near-past is selected for only one project, while it is marked as irrelevant for most of the others, even in the cross-project evaluation, for which almost all developer-centric are usually kept.

Project-related Features. Both *LOF* (*i.e.,* how focused the developer has been on a single project) and *#Repos* (*i.e.,* number of repositores to which the developer contributed) appear to be relevant. However, when studying the distributions, we did not find as clear difference between the buggy and non-buggy classes as the ones observed for the other metrics. Thus, we do not report the boxplot for such variables. This likely means that, while such metrics matter, they are probably only relevant if combined with other metrics.

Answer to RQ$_3$. Developer-centric metrics are more useful for projects with more developers. The working hour/day distribution, the average commit message length, the focus, and the number of repositories to which a developer contributed appear to be the most crucial developer-centric petrics.

VI. Discussion and Implications

Our empirical evidence provides a clear message: **Developer-related information are important for predicting the introduction of defects**. While we tested this for feature-based model, such a theory can easily be extended to other more advanced models as well. Indeed, recent work has started to explore the use of Deep-Learning (DL) [15] and Large Language Models (LLMs) [18] for JIT-SDP. Still, considering developer-related information in such models is trickier since they are mostly able to treat text (and the more complex features that can be extracted from it) rather than simpler numeric features like the ones we introduced. It would be interesting to explore the integration of such information in DL-based models and LLMs. For example, as for the latter, introducing examples of previous commits made by the same author could give the model having more context.

	Feature	Description	#Projects (WP)	#Projects (CP)
🕑	***HD***	**Hourly distribution of commits created**	**4/6**	**6/6**
	WD	**Distribution of work by weekday**	**4/6**	**6/6**
	PT_{sd}	Standard deviation of time between push events	3/6	6/6
	PT_{avg}	Average time between push events	3/6	5/6
🔧	***MCML***	**Average length of commit messages**	**5/6**	**6/6**
	WL_{ad}	Average workload per day	3/6	6/6
	Acc	Average number of commits per push event	3/6	6/6
	Mod_F	Average number of files modified in the period	3/6	6/6
	Mod_{L+}	Average number of added LOCs	3/6	6/6
	Mod_{L-}	Average number of removed LOCs	3/6	6/6
	WL_{wd}	Average workload per working day	2/6	5/6
	WAI	Workload increase in the last week	2/6	5/6
	Mod_{LF}	Average number of modified lines per file	2/6	5/6
	WL_{tot}	Total workload of the developer	1/6	3/6
📅	***#Repos***	**Number of repositories to which it contributed in a given period**	**4/6**	**6/6**
	LOF	**Percentage of time devoted to one project compared to others**	**4/6**	**6/6**
	#CS	Project transitions, how many times project changes	3/6	6/6
	LOF_N	Weighted entropy on commits	3/6	5/6

TABLE VII: Number of times each temporal (🕑), change-related (🔧), and project-related (📅) feature is selected in both evaluations. We report in boldface the metrics that are selected in more than half the considered projects in both evaluations.

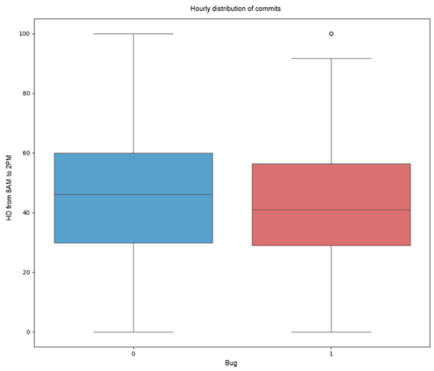

Fig. 1: Distribution of HD_{mo} (morning).

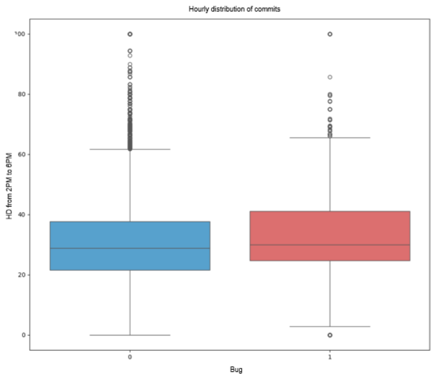

Fig. 2: Distribution of HD_{an} (afternoon).

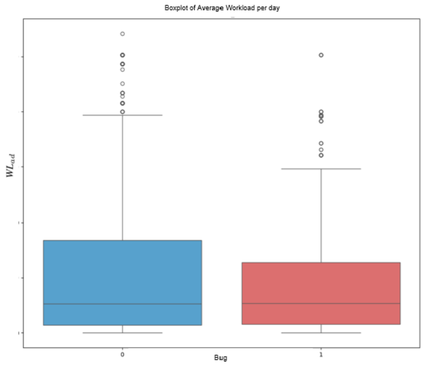

Fig. 3: Distribution of WL_{ad}.

The developer-centric metrics we considered aim at characterizing the behavior of a given developer in the near-past. A clear avenue for improvement is constituted by the introduction of features aimed at capturing the long-term behavior of developers and spotting any changes in the behavior. For example, if developers started writing shorter commit messages in the near-past, but they are used to write more detailed messages, it might mean that their workload increased and, thus, there might be a higher risk of introducing bugs. The same is true for temporal features: If a developer changed their circadian rhythm and started working in the night, it is possible that the risk of introducing bug increases. Similarly, information regarding the specific commit at hand can be related to the near-past or long-term behavior of the developer. For example, if a developer never works on Sundays and a specific commit is made on such a weekday, it might more likely introduce bugs.

Our results have implications for practitioners as well. First, developers should pay attention to the some "bad" habits we spotlighted (*i.e.,* writing shorter commit messages or working late), which might be associated to a higher risk of introducing defects. Since our features do not rely on the specific commit, but only on the past activity of the developer, a model containing only developer-centric features could be experimented and used *before* developers start working on a task (Before Time Defect Prediction). The practical suggestion that such a model could give is, for example, to rest before writing code or before making a commit, like Advanced Driver Assistance Systems (ADAS) alert the car driver when they detect fatigue.

VII. Threats to Validity

Threats to construct validity mainly pertain the selection of developer-centric features and their impact on Just-In-Time Software Defect Prediction (JIT-SDP) performance. We introduce specific metrics related to commit frequency and message length to represent developers' behavior. However, these metrics may not fully capture all relevant aspects of developers' activities, potentially leading to the incomplete interpretations of a developer's impact on defect introduction.

Another limitation of our approach is that it does not take into consideration the different accounts that may belong to the same developer. This is a well-known problem in the literature [**?**]. As a result, this limitation could lead to incomplete representations of developers' behavior and work patterns.

Threats to internal validity mainly concern the procedure used to collect and analyze data in our experiment. First, we only focused on a subset of commits available for the projects we studied (from March 2015) since the format for GitHub Archive changed on that date. A broader analysis on the whole set of commits could have led to different results. However, we believe that this thread does not affect the main result of our analysis, *i.e.,* that developer-centric metrics matter for predicting the bug-proneness of a commit. Besides, we relied on Info Gain [53] for selecting attributes in our experiment. This method, like any attribute selection method, might not select the best set of features, but a sub-optimal one. We used the default parameters of the classifiers as provided by the Weka [52]. As a result, it could be that none of the developer-centric metrics we considered is actually relevant. Still, we obtained consistent results among within different projects and evaluations (within-project and cross-project). This increases our confidence in our main conclusion — developer-centric features matter.

Threats to external validity concern the generalizability of the results to other contexts. This dataset we adopted is limited to 10 projects, eventually narrowed down to 6 because of the low number of recent contributions in four of them. The analyzed projects might be representative of the broad range of software projects available. It is worth noting, however, that our study considered the commits from a total of 1,614 developers.

VIII. Conclusion and Future work

In the last decade, Just-In-Time Software Defect Prediction (JIT-SDP) models have proven to be valuable tools for identifying potential defects in real-time when integrated within software development pipelines. Existing models, however, primarily rely metrics that can be computed taking into account the immediate context of the project at hand, neglecting developer-related aspects. In this paper, our studied to what extent integrating developer-centric metrics into JIT-SDP models allows to improve the accuracy of existing models. Our results clearly show that, from many perspectives, developer-centric features are important for predicting bugs. Indeed, a model that combines developer-centric features with state-of-the-art features generally achieves higher precision, recall, F1, and AUC. This effect is even more clear in projects with many developers. Among the introduced features, those related to the distribution of the developers' working hours and daily activity had a particularly strong impact. Our results support the idea that developer-centric features provide meaningful insights into defect-introducing behaviors, improving the accuracy of the JIT-SDP model. Our future research agenda includes the definition of more developer-centric metrics that consider both the long-term past of the developers' contributions and the relationship between the characteristics of the specific commit and the general behavior of the developer.

IX. Data Availability

We release the scripts we used to compute the developer-centric metrics and to run the experiment, the datasets, and the data analysis scripts in our replication package [58].

Acknowledgments

This work has been supported by the European Union - NextGenerationEU through the Italian Ministry of University and Research, Projects PRIN 2022 "DevProDev: Profiling Software Developers for Developer-Centered Recommender Systems", grant n. 2022S49T4W, CUP: H53D23003610001.

References

[1] Y. Tian, J. Tian, and N. Li, "Reliability assessment and prediction with testing efficiency growth for open source software," in *International Conference on Software Engineering and Data Engineering, SEDE 2017*, vol. 2017. ACM, 2017, pp. 72–83.

[2] Y. Zhao, K. Damevski, and H. Chen, "A systematic survey of just-in-time software defect prediction," *ACM Comput. Surv.*, vol. 55, no. 10, 2023.

[3] G. Canfora and A. Cimitile, "Software maintenance," in *Handbook of Software Engineering and Knowledge Engineering: Volume I: Fundamentals*. World Scientific, 2001, pp. 91–120.

[4] E. Alnagi and M. Azzeh, "Just-in-time software defect prediction techniques: A survey," in *2024 15th International Conference on Information and Communication Systems (ICICS)*, 2024, pp. 1–6.

[5] V. R. Basili, L. C. Briand, and W. L. Melo, "A validation of object-oriented design metrics as quality indicators," *IEEE Transactions on software engineering*, vol. 22, no. 10, pp. 751–761, 1996.

[6] A. E. Hassan, "Predicting faults using the complexity of code changes," in *2009 IEEE 31st International Conference on Software Engineering*, 2009, pp. 78–88.

[7] D. Di Nucci, F. Palomba, G. De Rosa, G. Bavota, R. Oliveto, and A. De Lucia, "A developer centered bug prediction model," *IEEE Transactions on Software Engineering*, vol. 44, no. 1, pp. 5–24, Jan 2018.

[8] 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," *Software Engineering, IEEE Transactions on*, vol. 39, pp. 757–773, 06 2013.

[9] ——, "A large-scale empirical study of just-in-time quality assurance," *IEEE Transactions on Software Engineering*, vol. 39, no. 6, pp. 757–773, 2012.

[10] 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*, 2016, pp. 157–168.

[11] W. Fu and T. Menzies, "Revisiting unsupervised learning for defect prediction," in *Proceedings of the 2017 11th joint meeting on foundations of software engineering*, 2017, pp. 72–83.

[12] Q. Huang, X. Xia, and D. Lo, "Revisiting supervised and unsupervised models for effort-aware just-in-time defect prediction," *Empirical Software Engineering*, vol. 24, pp. 2823–2862, 2019.

[13] R. Duan, H. Xu, Y. Fan, and M. Yan, "The impact of duplicate changes on just-in-time defect prediction," *IEEE Transactions on Reliability*, vol. 71, no. 3, pp. 1294–1308, Sep. 2022.

[14] Y. Fan, X. Xia, D. A. da Costa, D. Lo, A. E. Hassan, and S. Li, "The impact of mislabeled changes by szz on just-in-time defect prediction," *IEEE Transactions on Software Engineering*, vol. 47, no. 8, pp. 1559–1586, Aug 2021.

[15] 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*, 2021, pp. 427–438.

[16] J. Gesi, J. Li, and I. Ahmed, "An empirical examination of the impact of bias on just-in-time defect prediction," in *Proceedings of the 15th ACM/IEEE international symposium on empirical software engineering and measurement (ESEM)*, 2021, pp. 1–12.

[17] 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*, 2020, pp. 518–529.

[18] H. Wang, Z. Gao, X. Hu, D. Lo, J. Grundy, and X. Wang, "Just-in-time todo-missed commits detection," *IEEE Transactions on Software Engineering*, 2024.

[19] 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 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 11 2016, pp. 157–168.

[20] 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.

[21] L. Pascarella, F. Palomba, and A. Bacchelli, "Fine-grained just-in-time defect prediction," *Journal of Systems and Software*, vol. 150, pp. 22–36, 2019.

[22] C. Bird, N. Nagappan, B. Murphy, H. Gall, and P. Devanbu, "Don't touch my code! examining the effects of ownership on software quality," in *Proceedings of the 19th ACM SIGSOFT Symposium on Foundations of Software Engineering*, 09 2011, pp. 4–14.

[23] G. Laudato, S. Scalabrino, N. Novielli, F. Lanubile, and R. Oliveto, "Predicting bugs by monitoring developers during task execution," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 1–13.

[24] D. Posnett, R. D'Souza, P. Devanbu, and V. Filkov, "Dual ecological measures of focus in software development," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. IEEE Press, 2013, p. 452–461.

[25] A. Mockus and D. Weiss, "Predicting risk of software changes," *Bell Labs Technical Journal*, vol. 5, pp. 169–180, 06 2002.

[26] T. Gyimóthy, R. Ferenc, and I. Siket, "Empirical validation of object-oriented metrics on open source software for fault prediction," *Software Engineering, IEEE Transactions on*, vol. 31, pp. 897– 910, 11 2005.

[27] P. L. Li, J. Herbsleb, M. Shaw, and B. Robinson, "Experiences and results from initiating field defect prediction and product test prioritization efforts at abb inc." in *Proceedings of the 28th International Conference on Software Engineering*. ACM, 2006, p. 413–422.

[28] J. C. Munson and T. M. Khoshgoftaar, "The detection of fault-prone programs," *IEEE Trans. Softw. Eng.*, vol. 18, no. 5, p. 423–433, May 1992.

[29] G. dos Santos, E. Figueiredo, A. Veloso, M. Viggiato, and N. Ziviani, "Understanding machine learning software defect predictions," *Automated Software Engineering*, vol. 27, pp. 369–392, 12 2020.

[30] J. Li, P. He, J. Zhu, and M. R. Lyu, "Software defect prediction via convolutional neural network," in *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, July 2017, pp. 318–328.

[31] S. Wang, T. Liu, J. Nam, and L. Tan, "Deep semantic feature learning for software defect prediction," *IEEE Transactions on Software Engineering*, vol. 46, no. 12, pp. 1267–1293, Dec 2020.

[32] 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*. ACM, 2014, p. 172–181.

[33] 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.

[34] P. Tourani and B. Adams, "The impact of human discussions on just-in-time quality assurance: An empirical study on openstack and eclipse," in *Proceedings of the 23rd International Conference on Software Analysis, Evolution, and Reengineering*, vol. 1, 2016, pp. 189–200.

[35] 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, 2021, p. 427–438.

[36] 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 *Proceedings of the 11th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. IEEE Press, 2017, p. 11–19.

[37] M. Tan, L. Tan, S. Dara, and C. Mayeux, "Online defect prediction for imbalanced data," in *Proceedings of the 37th International Conference on Software Engineering - Volume 2*, ser. ICSE '15. IEEE Press, 2015, p. 99–108.

[38] J. G. Barnett, C. K. Gathuru, L. S. Soldano, and S. McIntosh, "The relationship between commit message detail and defect proneness in java projects on github," in *Proceedings of the 13th International Conference on Mining Software Repositories*, ser. MSR '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 496–499.

[39] H. Tessema and S. Abebe, "Enhancing just-in-time defect prediction using change request-based metrics," in *Proceedings of the 28th IEEE International Conference on Software Analysis, Evolution and Reengineering*, 03 2021, pp. 511–515.

[40] Y. Tian, N. Li, J. Tian, and W. Zheng, "How well just-in-time defect prediction techniques enhance software reliability?" in *2020 IEEE 20th International Conference on Software Quality, Reliability and Security (QRS)*, 2020, pp. 212–221.

[41] J. Eyolfson, L. Tan, and P. Lam, "Do time of day and developer experience affect commit bugginess?" in *Proceedings of the 8th Working Conference on Mining Software Repositories*. New York, NY, USA: ACM, 2011, p. 153–162.

[42] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" *ACM sigsoft software engineering notes*, vol. 30, no. 4, pp. 1–5, 2005.

[43] M. Claes, M. V. Mäntylä, M. Kuutila, and B. Adams, "Do programmers work at night or during the weekend?" in *Proceedings of the 40th International Conference on Software Engineering*. ACM, 2018, p. 705–715.

[44] GitHub, "GitHub Archive Program," https://archiveprogram.github.com/, 2024.

[45] T. K. Ho, "Random decision forests," in *Proceedings of 3rd international conference on document analysis and recognition*, vol. 1. IEEE, 1995, pp. 278–282.

[46] C. Ying, M. Qi-Guang, L. Jia-Chen, and G. Lin, "Advance and prospects of adaboost algorithm," *Acta Automatica Sinica*, vol. 39, no. 6, pp. 745–758, 2013.

[47] N. Fazakis, S. Karlos, S. Kotsiantis, and K. Sgarbas, "Self-trained lmt for semisupervised learning," *Computational intelligence and neuroscience*, vol. 2016, no. 1, p. 3057481, 2016.

[48] T. Bayes, "Naive bayes classifier," *Article Sources and Contributors*, pp. 1–9, 1968.

[49] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The weka data mining software: an update," *ACM SIGKDD explorations newsletter*, vol. 11, no. 1, pp. 10–18, 2009.

[50] S. Gnanambal, M. Thangaraj, V. T. Meenatchi, and V. Gayathri, "Classification algorithms with attribute selection: An evaluation study using weka," *International Journal of Advanced Networking and Applications*, vol. 9, pp. 3640–3644, 2018.

[51] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "Smote: synthetic minority over-sampling technique," *Journal of artificial intelligence research*, vol. 16, pp. 321–357, 2002.

[52] A. P. Bradley, "The use of the area under the roc curve in the evaluation of machine learning algorithms," *Pattern recognition*, vol. 30, no. 7, pp. 1145–1159, 1997.

[53] D. Montaigne, X. Marechal, T. Modine, A. Coisne, S. Mouton, G. Fayad, S. Ninni, C. Klein, S. Ortmans, C. Seunes *et al.*, "Daytime variation of perioperative myocardial injury in cardiac surgery and its prevention by rev-erbα antagonism: a single-centre propensity-matched cohort study and a randomised study," *The Lancet*, vol. 391, no. 10115, pp. 59–69, 2018.

[54] S.-S. Ren, L.-L. Xu, P. Wang, L. Li, Y.-T. Hu, M.-Q. Xu, M. Zhang, L.-N. Yan, T.-F. Wen, B. Li *et al.*, "Circadian rhythms have effects on surgical outcomes of liver transplantation for patients with hepatocellular carcinoma: a retrospective analysis of 147 cases in a single center," in *Transplantation Proceedings*, vol. 51, no. 6. Elsevier, 2019, pp. 1913–1919.

[55] T. Fry, T. Dey, A. Karnauch, and A. Mockus, "A dataset and an approach for identity resolution of 38 million author ids extracted from 2b git commits," in *Proceedings of the 17th international conference on mining software repositories*, 2020, pp. 518–522.

[56] E. Guglielmi, A. D'Aguanno, R. Oliveto, and S. Scalabrino, "Replication package of "enhancing just-in-time defect prediction models with developer-centric features"," https://doi.org/10.6084/m9.figshare.27633297, 2024.