

Continuous Integration and Continuous Delivery-Ready Defect Prediction for Risk-Aware Test Case Prioritisation

Jijo Joseph Cheeramban George*, D. Peter Augustine
Department of Computer Science, Christ (Deemed To Be University), India

Abstract—Continuous Integration and Continuous Delivery pipelines are highly dependent on automated regression tests to enable quick and effective feedback to the developers. Test Case Prioritisation provides a useful approach by arranging the tests in a sequence such that the ones that are likely to yield most faults are run first. Most of the current models are based on heavyweight learners that require significant training time. This article will tackle this problem by introducing a Continuous Integration and Continuous Delivery-ready defect prediction pipeline using LightGBM, which will enable predicting fault-proneness estimates in limited time. The analysis shows that LightGBM can perform equally well or better in predictive accuracy and can significantly decrease training and scoring time. These results make LightGBM-based defect prediction a practical foundation for time-sensitive, risk-aware test prioritisation in Continuous Integration and Continuous Delivery pipelines.

Keywords—Continuous integration and continuous delivery pipelines; defect prediction; test case prioritisation; risk-aware testing; regression testing

I. INTRODUCTION

Continuous Integration and Continuous Delivery (CI/CD) have been adopted, radically transforming the current state of software development. Rather than lengthy, infrequent release periods, modern software systems develop in an endless sequence of small, minor changes, which are incorporated, constructed, and assessed constantly. This change has helped organizations to be more responsive to the needs of the users as well as market pressures; however, it has also placed unprecedented demands on the speed, reliability, and automation of quality assurance endeavours. Automated regression testing in this case is an important protection measure as it guarantees that the new functionality will not be impacted by the changes in the implementation process.

The regression test suites tend to increase with the increase in the size of the software system, its complexity, and its lifespan. Long-established industrial systems often have thousands of test cases, many of which are stored to maintain an illusion of trust in old behaviour. Although a complete test is preferred in terms of quality assurance, it is not always feasible to run a complete regression package with every change. Even in CI/CD pipelines, where a build can be triggered many times a day, the slightest delay in the execution of a test can interrupt developer processes, augment the number of context switches, and delay delivery. Consequently, development teams

experience an unending conflict between extensive testing and fast feedback.

This pressure has encouraged widespread investigations into cost-efficient testing plans. In general, these strategies encompass test suite reduction, test case selection, as well as test case prioritisation. Test suite reduction is the permanent elimination of unnecessary tests, whereas test case selection is selecting a set of appropriate tests in relation to a particular change. While the two methods can lower execution costs, they can also drop tests that may be useful elsewhere or over longer periods. TCP presents an alternative, and perhaps more realistic, approach since it maintains the entire test suite but alters the sequence of test execution to ensure that tests with the greatest potential utility are run first. This enables pipelines to be aborted when time or resource requirements occur early without impacting the long-run test coverage.

The way tests are sorted out is the key to the effectiveness of TCP. In conventional prioritisation methods, heuristics are usually based on the coverage of the program, e.g. Total Coverage and Additional Coverage, which give preference to tests that cover a greater proportion of the program (or new coverage of a program) compared to tests previously chosen. These heuristics are appealing in that they are not complex, relatively easy to apply, and based on readily accessible tooling. Nevertheless, empirical data reveals that coverage-based prioritisation may result in redundant early execution, particularly in large test suites that have numerous tests that test the same code areas. In these situations, the first test executions have a diminishing marginal value, which restricts the pace at which different failures can be revealed.

Risk-aware prioritisation methods have been suggested in order to overcome this shortcoming. These techniques do not give equal weight to all code, but rather they emphasize tests that involve components that are more likely to fail. At the heart of these strategies lies defect prediction, which approximates the fault-proneness of software modules with the aid of static code metrics, historical defect data, and change characteristics. Predicting defects allows testing activities to concentrate on the areas that are more prone to failure by using the patterns of past failures, which could enhance prompt fault detection.

The literature has shown that there is significant potential for integrating defect prediction in the context of regression testing and TCP. However, only limited adoption of defect prediction is observed, though the empirical outcomes of the method are encouraging in CI/CD-based TCP pipelines. The major

*Corresponding author

challenge is not prediction accuracy, but practical feasibility. A large number of the state-of-the-art defect predictor models are based on heavyweight learners, most prominently the ensemble boosting methods, which include XGBoost, and need huge training periods and querying resources. The cost of predicting defects in itself may become a bottleneck in a setting where a fast turnaround is vital. In cases where the prediction latency is similar to or even greater than the time of running the test, it may happen that prediction is disabled, which cancels the benefits of risk-aware prioritisation.

The given challenges present one of the important gaps between research-level defect prediction and its application in the CI/CD environment. Although predictive performance has been the focus of most of the literature, little research addresses training and scoring latency as issues of first-class evaluation. Though predictive performance is key, equal attention should be given to the fact that even minor delays at the stage of a series of pipeline executions may add up. Consequently, the necessity to develop CI/CD -ready defect prediction pipelines that are consistent in predictive effectiveness and strict operational constraints is increasing.

The current study is aimed at filling this practical gap by creating and testing a time-sensitive defect prediction pipeline specifically on CI/CD-driven risk-sensitive TCP. The proposed approach leverages LightGBM, which is a tree-based ensemble learning framework, but it is efficient due to its use of histogram-based learning and leaf-wise tree learning. LightGBM has been an increasingly popular machine learning software in large-scale machine learning, which can provide strong predictive accuracy with much lower training time and memory usage. However, a systematic study of its applicability as an alternative to heavier models like XGBoost for defect prediction as a regression testing method has not been examined.

To assess the suitability, we conduct an empirical study based on real-world defects derived from the Defects4J and Defects4J +M benchmarks. These datasets constitute realistic development histories and include ground-truth failing tests across five popular studied Java projects: Chart, Closure, Lang, Math, and Time. The training algorithm for defect prediction models uses a rolling online learning strategy, reflecting how defect prediction models are updated in actual CI/CD pipelines. Standard classification measures are used to determine predictive effectiveness, and direct measures of training and scoring latency are used to establish computational feasibility.

The current research contributions include, first, a CI/CD-ready defect prediction pipeline that directly balances latency and repeatability without compromising predictive quality. Second, systematically compare LightGBM and XGBoost under the same experimental settings and point out the trade-offs between accuracy and computational cost. Third, we show that LightGBM-based defect prediction provides an effective basis for risk-aware test case prioritisation, enabling frequent model updates in CI/CD cycles.

II. BACKGROUND AND MOTIVATION

Defect prediction is primarily aimed at identifying software components with a higher likelihood of faults in them so that testing and quality assurance can be directed where they are

most required. Empirical evidence gathered over the last twenty years or so has always indicated that the distribution of faults is not homogeneous in software systems. Rather, the defects will be clustered around a relatively small amount of code influenced by structural and process-based factors, including the code complexity, module size, frequency of changes, developer activity, and historical fault density. Such a phenomenon is commonly referred to as the 80-20 rule of software bugs, where a minority of components are responsible for a majority of failures observed.

With these non-uniform patterns, defect prediction models provide a mechanism for risk-based decision-making. Instead of testing equally across the whole system, teams are able to focus on those components that have historically shown behaviour that is associated with a greater likelihood of a fault. This strategy can be used to enhance the effectiveness and efficiency of regression testing when used efficiently. This is especially true in large and changing systems where exhaustive testing is not feasible.

The prediction of defects is of particular significance in the case of regression testing, where the main aim is to identify the unwanted behavioural shifts brought about by the new modifications. Not all changes are equally risky. Refactorings in the stable modules can have comparatively low risk, and refactorings in the complex, highly connected, or historically faulty components are more likely to cause regressions. Risk-aware methods take advantage of this asymmetry by giving various levels of priority to various sections of the system so that the testing effort can be better aligned.

In CI/CD environments, though, the defect prediction utility is time-sensitive by nature. CI/CD pipelines are created with the purpose of having quick feedback within minutes to enable quick development cycles. To be useful in such a setting, defect prediction models have to be trained, updated, and run within a short duration of time so that they can be used to influence the decisions made in test execution before regression testing commences. Late predictions or those that make pipeline latency considerably larger are of limited practical use, regardless of their accuracy.

This necessity creates a basic conflict between predictive complexity and operational feasibility. Traditional defect prediction pipelines typically assume offline training and batch testing, in which models are trained regularly on large historical data and then used freely without time restrictions. These assumptions can be sensible in research contexts or in pre-release quality testing, although they are not well fitted to the flow of CI/CD, where the code can change a lot and so must the model.

A number of reasons can be linked to the fact that the traditional defect prediction pipelines cannot be compatible with the CI/CD practices. First, feature extraction processes may be computationally expensive themselves, especially those based on deep static analysis, fine-grained change history, or detailed dependency graphs. Second, a great number of high-performing learning algorithms cannot be trained without a lot of hyperparameter tuning in order to gain the best performance. This tuning procedure is effective under controlled experiment conditions, but it can greatly add to training latency. Third, the

models themselves (in particular, those using large numbers of trees) can be computationally intensive to train and run.

Consequently, heavyweight-learner-based defect prediction pipelines may eventually be a limitation in time-bounded contexts. Users might be tempted to turn off or avoid such models when they experience a prediction latency that is either close to or beyond acceptable CI/CD execution windows. Practically, this usually leads to a retreat to the more basic methods of prioritisation, like coverage-based heuristics, which are more convenient to implement even with their well-known limitations. This operational feasibility versus theoretical effectiveness difference is one of the biggest obstacles to the widespread use of defect prediction in CI/CD-oriented test case prioritisation.

The more recent research has started to recognize these difficulties, and the importance of the need to predict defects with latency factors in mind. Nonetheless, latency is often still regarded as a secondary aspect of concern, reported post hoc, and not as a core design objective. In most experiments, predictive accuracy metrics are prevalent in the evaluation, and training and scoring time are briefly discussed or not mentioned at all. In terms of the adoption of CI/CD, such an imbalance devalues otherwise potentially useful strategies.

This study is motivated by the fact that defect prediction of CI/CD should be good enough and fast, rather than theoretically optimal yet slow. A model that is a few percent less accurate, can be easily retrained, and can be run with reasonable predictability within CI/CD time budgets can be more useful than a model that is more accurate but has a high cost of computation that would not be used frequently.

Light Gradient Boosting Machine (LightGBM) is a strong competitor in this regard. LightGBM was created with efficiency as a key objective and uses histogram-based learning as one of the methods to minimise memory consumption and speed up split finding, and the leaf-wise tree growth method that scales up quickly. The attributes allow LightGBM to train much faster than most gradient-boosting implementations, as well as to scale more effectively, especially on tabular data with mixed feature scales, which is characteristic of software metrics datasets.

In spite of these benefits, LightGBM has a relative lack of interest in defect prediction studies, with other learners, including Random Forests and XGBoost, still prevailing. Although XGBoost has a high predictive performance, it is also known to have large training and tuning overhead, particularly when there is a lot of repetition in training models. This brings up a crucial question in CI/CD situations: can LightGBM offer a viable alternative to XGBoost and offer the same predictive guarantees with significantly lower computational latencies?

To answer this question, it is necessary to evaluate it in realistic conditions, with data sets that are representative of real development histories and not artificially balanced data. In addition, the training protocols should not be biased toward lookahead, but should simulate how models would be maintained in the real world, with exposure to new versions of the system gradually.

It is only under such consideration that the suitability of LightGBM as a defect predictor ready to be used in CI/CD could be properly evaluated.

This gap between research studies on defect prediction and practice in CI/CD is the motivation of the present research. Close comparisons between LightGBM and XGBoost under the same experimental conditions, i.e. rolling training, actual defect data and explicit latency measurement, we will attempt to give empirical evidence on whether fast, lightweight defect prediction can be a valid risk signal used in regression testing. By doing so, this work is aimed at repositioning defect prediction as the preferred operational component of the CI/CD-driven test case prioritisation rather than an offline tool in the analytical category.

III. RELATED WORK

This section is a review of the existing research with respect to the proposed study in two aspects. First, we overview the literature on software defect prediction models, and in particular, the development from previous statistical methods to the recent ensemble and boosting methods. Second, we analyse works incorporating defect prediction into regression testing and test case prioritisation, highlighting the constraints regarding computational latency and applicability of CI/CD.

A. Defect Prediction Models

The initial work on software defect prediction was mainly based on statistical and regression models of the predictor models constructed by using the metric of static code. Initial research has shown that size, complexity, coupling, and cohesion metrics, including those suggested by Chidamber and Kemerer [9], have statistically significant relationships with fault-proneness [1], [2]. Later empirical evidence was able to substantiate the fact that the prediction models of defects created using such metrics would be effective as early indicators of software quality [3], [4].

With the emergence of machine learning methods, scholars started to discard the simplistic regression models in favor of the classification-based approach. Algorithms like Logistic Regression, Naive Bayes, Decision Trees, and Support Vector Machines have all been popularly tested on benchmark datasets with better predictive performance compared to pure statistical methods [10], [15]. These methods, however, struggled with class imbalance and feature interactions, as is the case in real-world defect data.

In order to overcome these shortcomings, ensemble learning methods gained popularity. Gradient Boosted Decision Trees and Random Forests, especially, were effective because they could model non-linear relationships and deal with noisy features [6]. Extreme Gradient Boosting (XGBoost) was the most popular boosting technique in the defect prediction literature because it is able to offer consistent results when utilized on a wide range of datasets and is also able to cope with imbalanced classes [7], [6]. Several researchers have indicated that when used to predict defects using both statistical and process measures, XGBoost performed better than the traditional learners on the F1-score, AUC, and MCC metrics.

More recent studies have considered deep learning-based defect prediction, which uses neural representations of source and execution traces, to extract semantic information that goes beyond hand-crafted features [8], [4]. Although deep learning models have been demonstrated to have high predictive power, systematic reviews have pointed to their high computational cost, reliance on hyperparameter fine-tuning, and low interpretability as limiting their use in time-sensitive development settings [8].

In spite of the improvement in predictive effectiveness, multiple studies highlight a significant drawback of modern defect prediction studies: the accuracy does not guarantee any practical usefulness. According to the works by Menzies et al. and later authors, the time it takes for feature extraction, model training, and inference is one of the most important factors that are used to determine if a model can be put into operation or not [5], [11]. This is especially a central issue in the continuous development environment, where models need to be retrained regularly to capture an evolving codebase.

As a result, there is increased focus on lightweight learning methods and incremental training strategies that can trade marginal predictive gain for high computational overhead reduction. Nevertheless, the majority of the previous literature is qualitative in assessing these factors or assumes offline training conditions, leaving a gap in the systematic latency-level analysis of defect prediction models in CI/CD pipelines.

The use of defect prediction in regression testing has been used mainly in relation to risk-aware test case prioritisation (TCP). The main concept is to use the estimates of fault proneness to assign tests that test components with higher predicted risk higher priority, which will enhance early fault detection. Empirical studies have revealed that risk-related tests can be prioritized to outperform in evaluations like First-Fail and Percentage of Faults Detected (APFD) over mere coverage-based methods [12] [13].

A number of methods integrate both the defect prediction and code coverage data to prioritize the tests in terms of their risk contribution in the program elements they test [14] (Mostafa et al., 2022). These techniques tend to perform better than baseline prioritisation strategies, especially in projects that have had rich defect histories. Researchers also, however, observe that naive risk-weighted ordering may also create additional sources of redundancy, in which two or more early tests are over the same high-risk components, and thus constrain behavioural diversity and diminish marginal returns.

In addition to fixed risk-aware prioritisation, there is recent research on adaptive and learning-based TCP approaches to CI environments. Prioritisation schemes based on machine-learning-driven prioritisation, reinforcement learning, and history-based schemes have demonstrated potential to dynamically change the prioritisation strategies based on the response of the previous test executions [2], [16]. Extensive industrial case studies show that the information on past failures is a powerful predictor of future failures, especially in the context of CI [17].

The cost of predicting defects is a factor that is often ignored regardless of these advances. Numerous TCP works assume that

the estimates of fault-proneness can be obtained cheaply or easily. Practically, such estimates may need to be obtained by repeatedly training and inferring models, which may significantly increase the pipeline latency. Very few studies specifically consider training and scoring overhead in their experimental design process [17], [18].

According to recent surveys and systematic literature reviews, it is crucial to assess defect prediction and TCP techniques in the context of realistic CI/CD constraints, including limited time budget, changing datasets, and the need to meet reproducibility requirements [19], [20]. These reviews point to an increasing mismatch between academic potentially useful methods and those that can be operationally implemented in high-speed development workflows.

In brief, it can be said that current studies present defect prediction as a useful element of risk-sensitive regression testing, though they do not answer all the questions related to computational feasibility in a CI/CD context. The majority of previous literature gives preference to quality of predictions over timeliness or latency is an implementation detail instead of a core evaluation criterion.

IV. RESEARCH GAP

The reviewed literature indicates that there is a blatant gap at the crossroads of the accuracy of defect prediction and computational efficiency. Although more sophisticated methods, including XGBoost and deep neural networks, are taking over research efforts, their training and inference are expensive, not allowing them to be practically used in continuous testing pipelines. On the other hand, lightweight solutions are either not well-explored or not sufficiently tested on realistic defect data using CI/CD-compatible training protocols.

To close this gap, the study will evaluate LightGBM as a CI/CD-ready defect prediction model in a systematic manner where both the predictive effectiveness and computational latency are clearly taken into account. Contrary to previous research, we assume that training and scoring time are primary evaluation measures and use rolling, version-sensitive training to simulate CI/CD realistic usage. With this, the current study will supplement the available literature on risk-aware TCP by considering defect prediction as a deployable unit in modern regression testing pipelines.

Even though software defect prediction has been shown to have a high potential in enhancing risk-sensitive test case prioritisation, the practical application of software defect prediction in CI/CD pipelines is rare. The main cause of this disparity is not the inability to predict this defect, but rather the computational cost of training and deploying defect prediction models under the rigid time constraints of the current development processes. CI/CD pipelines are created to deliver results in an extremely short amount of time, and any other processing that slows down tests can dramatically decrease their performance.

Most of the current defect prediction strategies are based on heavyweight machine learning models and a considerable feature engineering pipeline. Although these methods can achieve high predictive accuracy in offline or batch-processing

settings, they are insensitive to the features of CI/CDs where code changes happen continuously, and the model needs to be retrained on a regular basis. Defect prediction itself can become a bottleneck in such environments due to long training times, complicated hyperparameter tuning processes, and expensive inference processes. Consequently, practitioners can decide to either turn off risk-aware mechanisms or revert to other less complex heuristics, including coverage-based prioritisation, even though they are known to have limitations.

The other difficulty is the lack of relationship between the research evaluation and the deployment situation. Most studies analyse defect predictive models with the assumption of fixed datasets and low-frequency retraining, but do not overtly consider training and scoring latency as being first-class metrics of evaluation. Nonetheless, in practical CI/CD pipelines, even small delays may be accumulated in each iteration of the run, which ends up breaking the continuous feedback loop the CI/CD is intended to provide.

Thus, it is evident that defect prediction pipelines with a balanced prediction-computation ratio are in demand. These pipelines should be able to be retrained within seconds, be combined with the current CI/CD processes, and be run over and over again without destabilizing the pipeline performance. To overcome this difficulty, it is necessary to reconsider the popular learning algorithms and determine whether less weighty versions can provide acceptable predictive power and meet the operational requirements.

Research Objective: The research aims to create and test a CI/CD compatible defect prediction pipeline that can effectively estimate fault proneness with high accuracy and under the strict time constraints that the new regression testing pipelines require.

The research questions of the study are the following:

RQ1: Does LightGBM have comparable predictive performance to XGBoost on real-world datasets with respect to defect prediction?

RQ2: Does LightGBM reduce the training and scoring latency of a CI/CD like setup compared to XGBoost?

RQ3: Does LightGBM defect prediction offset XGBoost as an appropriate substitute to be incorporated in test case prioritisation?

V. PROPOSED METHODOLOGY

The defect prediction pipeline proposed focuses on the clear aim of aiding time-sensitive risk estimation in CI/CD settings. In contrast to the old-fashioned offline workflow of defect prediction that focuses on predictive accuracy only, this workflow focuses on lightweight computation, repeatability, and practical deployability, although it still has enough predictive power to inform risk-aware prioritisation of test cases. The workflow comprises three components, including feature engineering, model training strategy, and learner selection. Every element is consciously limited to reduce the runtime cost and integration friction in real-world CI/CD setups.

A. Feature Engineering

In defect prediction, feature engineering is a very significant factor because feature quality and availability are directly

proportional to predictive performance as well as the cost of computation. The choice of features in CI/CD settings needs to be a trade-off between expressiveness and efficiency; that is, features need to be informative enough to reflect fault-proneness, but at the same time, features should also be inexpensive to compute and update.

The feature set of the proposed pipeline is class-level metrics of static code. They are metrics that encapsulate structural aspects of software constituents that have been reported extensively to predict defect occurrence. In particular, the chosen metrics are indicators of six major dimensions, which include size, complexity, coupling, cohesion, inheritance, and documentation quality. All these dimensions combine to define the size and complexity of a component, the level of interaction it has with other components, the localization of responsibilities, and its comprehensibility as well as maintainability.

In order to supplement the measures which are static, the set of features is extended by historical defect measures and inter-version change information. Such historical indicators indicate that a class has had faults in the earlier versions, and these are useful in providing context regarding trends of defects. Inter-version change characteristics, e.g., numbers or size of changes between one version and the next, indicate churn and instability in the code, which are commonly linked to an increased likelihood of faults in evolving systems.

Notably, every feature applied in the pipeline is selected with two principles. To begin with, they have to be inexpensive to calculate, not based on deep static analysis, dynamic instrumentation, or costly repository mining. Second, they should be easily available within the regular construction environment and can be extracted automatically as part of CI/CD processes. The limitation of the feature space to cheap, commonly available measurements also helps the pipeline to make sure that feature extraction is not a bottleneck when retraining the model or influencing it. This feature of isolated design, as well, facilitates consistency and reproducibility, since one and the same metrics can be retrieved across versions and projects with minimal configuration.

B. Model Training Strategy

The training plan followed in the workflow is aimed at being as close as possible to the practice of the use of defect prediction models in CI/CD settings. The pipeline uses a rolling online split in order to train and evaluate the model, as opposed to assuming access to a fixed, completely labeled dataset.

According to this approach, the model is trained on the basis of information available when a project reaches its current version. Given a software version v_i , the defect prediction model is trained on all the previous versions, i.e., $v_1, v_2, \dots, v_{i-1}$, and then tested on version v_i . This is repeated on the next successive version, yielding a series of successive model updates that resemble incremental learning in actual CI/CD pipelines.

There are several uses of this design. To begin with, it removes look-ahead bias, i.e., predictions of each version are executed without having information about the future. It is essential in order to have validity and realism in experiments. Second, it represents the practical nature of CI/CD, where models have to be constantly updated as new versions and defect

statistics are available. Third, it has a natural ability to support version-aware adaptation, meaning that the model can react to the changing characteristics of the system over time.

A combination of negative class sampling and class weighting is used to overcome the problem of an imbalance between the classes. The distribution of software defect data is usually skewed, with a lot of defective classes as compared to clean ones. Instead of using heavy re-sampling classification methods, which add to the dataset size complexity and the training cost, the pipeline would prefer simple adjustment methods that do not add to the cost of learning but, instead, enhance learning stability.

Training frequency is also another crucial factor of consideration. Ideally, every version of the pipeline run can be retrained with the help of the rolling strategy. Practically, the retraining rate can be set according to the requirements of the organization, i.e., retraining can be performed daily or after a certain number of commits. The efficiency of the pipeline means that even the rather frequent retraining can be accomplished within the limits of CI/CD time budgets. The pipeline provides a sound tradeoff between realism, predictive quality, and computational feasibility by incorporating rolling training with effective imbalance management.

C. Choice of Learner

The decision made on the learning algorithm is what makes the pipeline goals successful. Although many machine learning models have performed well in predicting defects, a lot of them have enormous computational overheads either during training or inference. This is especially in the case of ensemble learners whose trees are deep, have a large number of boosting rounds, or whose optimisation schedules are intricate.

In order to overcome these shortcomings, Light Gradient Boosting Machine (LightGBM) is proposed as the central learner. LightGBM is an ensemble model that is a tree-based model, which is specially optimized to be efficient with histogram-based split finding and leaf-wise tree expansion. These design decisions enable less memory to be used and also less training time to be consumed; thus, LightGBM is best suited to be used frequently in time-constrained settings.

LightGBM has a variety of regularization tools, where one can control model complexity and mitigate the risk of overfitting, a useful property in scenarios where the data being trained is changing, and the distribution of defects may change. Moreover, LightGBM supports heterogeneous and tabular feature sets, which are applicable to the static and historical metrics that the current pipeline uses.

Randomized search takes place for hyperparameter tuning within restricted parameter spaces. Optimisation of hyperparameters was performed using random search within a limited parameter space. Randomized search is also computationally cheaper than exhaustive grid search and offers broader exploration, making it appropriate for CI/CD environments. The ranges in which parameters are allowed are limited intentionally to avoid too much complexity in the model and to make the models repeatable.

Trained models are also stored in cache and used again when necessary to minimise overhead. Model artifacts reflecting the particular version or training window are saved and can be reloaded into the next scoring task, thus eliminating the necessity of retraining where there is no significant change in underlying data between training and scoring. This caching is beneficial in CI/CD scenarios, where repetitive, similar builds can be triggered.

The selection of LightGBM can be seen as a wider design philosophy of current work: it does not conceive it as excessively complex, but as operationally viable. Whereas heavy learners can potentially provide some marginal improvement to predictive metrics, they have the potential to be more expensive than beneficial in fast-paced testing processes. The pipeline allows defect prediction to be used as a practical, always-available risk estimation element rather than an occasional offline analysis tool.

VI. EXPERIMENTAL SETUP

A. Dataset

Current research is analysed based on Defects4J and Defects4J+M, containing 357 bug versions in five Java projects: Chart, Closure, Lang, Math, and Time. Details of these projects are mentioned in Table I and Fig. 1. These datasets contain real defects and ground-truth failing tests, and it is reasonable to evaluate them in a realistic manner.

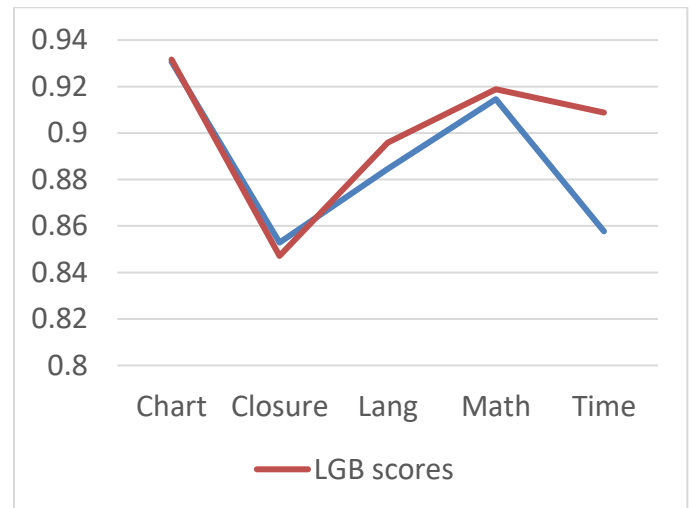


Fig. 1. Matthews Correlation Coefficient (MCC) scores with existing results.

TABLE I. DATASET PROJECT VERSIONS, NUMBER OF BUGGY VERSIONS AND TEST CASES

Identifier	Project name	Buggy Versions	Test Cases
Chart	jfreechart	26	2205
Closure	closure-compiler	133	7927
Lang	commons-lang	65	2245
Math	commons-math	106	3602
Time	joda-time	27	4130

XGBoost is considered the primary base since it is extensively used in the study of defect prediction. The equality of learning is ensured through the use of equal features and training regimes for the learners. In predictive performance, MCC, F1, AUC, Precision, and Recall are applied. Measures of computational efficiency are the time taken for training and scoring.

VII. RESULTS AND ANALYSIS

A. Predictive Performance

LightGBM's predictive performance was similar to XGBoost in all projects. On the Math project, XGBoost slightly outperformed LightGBM. However, in some projects, such as Time, LightGBM had a higher MCC score. The total difference observed was small compared to the regular significant decrease in the computational latency. The presence of such stability in a variety of software systems means that the predictive ability of the suggested pipeline will not decline due to the decrease in computational complexity. The differences in the MCC values between LightGBM and XGBoost are generally small. This means replacing XGBoost with a lighter learner does not affect the capacity of the model to predict defective and non-defective components. Fig. 2 gives an overall picture of the results discussed. The results are especially significant given CI/CD, where predictive reliability is to be preserved across successive versions. From the results, it is clear that LightGBM can learn similar fault-proneness patterns as the more computationally demanding models, and hence, it is relevant as a defect prediction engine under evolving project conditions.

B. Training and Scoring Latency

Unlike the close predictive parity, the difference between the two learners in computational latency is very significant. In any project, LightGBM cuts these end-to-end training and scoring times down to about 5 minutes compared to the 30-35 minutes in any project. This decrease is a major enhancement and is a direct response to the main obstacle to implementing defect prediction within CI/CD pipelines. An overall picture of this can be found in Fig. 2. This reduction in latency means that defect prediction can be done on a regular basis without spending impractical time budgets on the pipeline. Notably, this decrease is not limited to model training but also to scoring, so that the predictions can be produced well in advance before the commencement of the regression tests. Table II shows the test case prioritization effectiveness results obtained using fault-proneness scores generated by the proposed defect prediction

models. The First-Fail metric is reported to evaluate how quickly failing tests are exposed during CI/CD execution.

TABLE II. PRIORITISATION EFFECTIVENESS: FIRST-FAIL VS. BASELINES

Project	Base Paper First-Fail	Proposed First-Fail	Δ (absolute)
Chart	1.31	0.8301	-0.4799
Closure	9.19	8.5807	-0.6093
Lang	8.79	5.7046	-3.0854
Math	8.98	6.9635	-2.0165
Time	4.31	4.2665	-0.0435

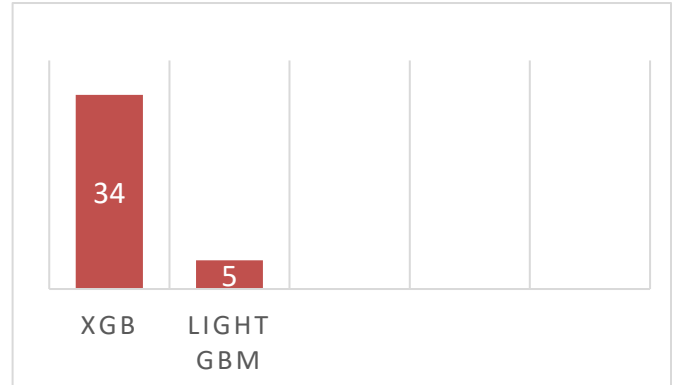


Fig. 2. XGB vs LGB comparison.

C. Practical Implications

Practically, these findings indicate that LightGBM-based defect prediction can be run at the same frequency as CI/CD-compatible frequencies (e.g., per build or daily retraining), without interfering with developer processes. This allows risk-aware prioritisation of test cases to be more practical in practice, since, in a real-world setting, risk estimation is frequently not adopted because its latency prevents risk prioritisation.

The proposed pipeline offers a practical basis for incorporating defect prediction into ongoing testing plans as it offers nearly identical predictive accuracy at a far lower computational cost. Table III shows the predictive performance of the LightGBM model across the five Defects4J projects using Accuracy, Precision, Recall, F1-score, and MCC. Table IV shows the corresponding predictive performance results obtained using XGBoost under the same rolling-version evaluation protocol.

TABLE III. DEFECT PREDICTION XGB (ONLINE)

Project	Type	MCC	F1	Area Under the Curve	Precision	Recall	Accuracy	Avg Time
Lang	lgb	0.932	0.919	0.995	1	0.85	0.9993045	0.9103
Math	lgb	0.847	0.817	0.972	0.8941	0.753	0.9995527	1.4845
Chart	lgb	0.896	0.895	1	0.85	0.944	0.9997832	0.3641
Closure	lgb	0.919	0.919	0.994	0.9084	0.93	0.9997879	1.8628
Time	lgb	0.909	0.909	0.998	0.9091	0.909	0.9994186	0.3781

TABLE IV. DEFECT PREDICTION LGB (ONLINE)

Project	Type	MCC	F1	Area Under the Curve	Precision	Recall	Accuracy	Avg Time
Lang	xgb	0.931	0.929	0.994	1	0.867	0.9993818	6.1906
Math	xgb	0.853	0.853	0.98	0.875	0.832	0.9996185	10.095
Chart	xgb	0.885	0.878	1	26	1	0.999729	2.4762
Closure	xgb	0.915	0.915	0.993	0.9077	0.922	0.9997778	12.667
Time	xgb	0.858	0.857	0.999	0.9	0.818	0.9991279	2.5715

VIII. DISCUSSION

The findings from the study indicate that one of the core and common trade-offs observed in software defect prediction is the complexity versus the feasibility of a model. Although heavyweight machine learning models, including XGBoost, are currently popular because of excellent predictive capability, they can be computationally intensive, making them impractical in time-constrained development settings. Comparatively, the findings also indicate that LightGBM is more favorable in terms of balance, whereby its predictive performance is equivalent to that of XGBoost and its training and inference latency is significantly reduced.

The trade-off is especially applicable in CI/CD pipelines where generating timely feedback is more likely to be of importance than marginal improvement in predictive accuracy. Practically, too slow defect prediction models that cannot be retrained or deployed may not even be included in the pipeline. The noticed predictive performance similarity, along with a significant decrease in computing cost, indicates that LightGBM is more consistent with the practicality of continuous testing environments.

The next point to consider is that predictive effectiveness should take into consideration the deployment frequency. A model that can be retrained only periodically, even though slightly more accurate, can be of less overall value than one that can be retrained more often, and that which is more in phase with the changing codebase. In this sense, LightGBM's latency can be leveraged more effectively, making risk estimates more relevant and up-to-date and allowing them to be updated more frequently.

The results also support the bigger picture according to which timeliness must be considered as a first-class quality dimension of CI/CD-oriented defect prediction. The traditional tests usually maximize accuracy, assuming that things are static, whereas the present study has shown that the same level of accuracy can be attained with much better responsiveness. This helps make defect prediction a useful building block for integrating with more complex regression testing strategies.

In general, the discussion supports the main argument of this work: CI/CD defect prediction models should be developed not only to be accurate, but also to be used in a long-term repeatable manner that fits within a limited execution time frame.

IX. THREATS TO VALIDITY

Similar to any empirical study, the results of this work can be affected by a number of possible threats to validity. These

threats are divided into construct validity, internal validity, and external validity.

Construct validity issues are related to whether the measures and metrics and experimental measures are an appropriate reflection of the concepts under assessment. The performance of defect prediction in this work is evaluated based on the common classification measures, including MCC, F1, and AUC, and computational efficiency is determined by the time of training and scoring. Although these metrics are popular in studies of defect prediction, they fail to address everything regarding practical utility. As an example, the severity of a fault and downstream effects are not directly modeled, which can have implications for the interpretation of predictions in practice at the actual testing stage.

Internal validity is concerned with the reliability of the results that are measured through the suggested methodology and not the instrumental factors. Look-ahead bias is mitigated by a rolling online training strategy that closely resembles actual CI/CD deployments. The same sets of features, training regimes, and evaluation criteria are used among learners to have a fair comparison. However, stochastic factors like randomized hyperparameter optimization and natural fluctuations in project development can still have an effect.

External validity deals with the generalizability of the findings. The empirical analysis is applied to Java-based projects of Defects4J and Defects4J+M, which, though popular and representative, are not exhaustive of all programming languages and development systems. Nevertheless, the proposed methodology is mostly language-independent, as it is based on the general structural measures and learning strategies that can be applied to other ecosystems in case of proper support with feature extraction.

Defect-history features might introduce a risk that the history will be more predictive than the actual defects. Using defect-history features may introduce a risk that the historical fault information will be more predictive than actual faults, which should be investigated further. While only information available before the target version is used, future studies should look into feature-ablation experiments to determine the contribution of the defect-history metrics. Moreover, results are obtained based on single experimental runs, and variability from random initialisation and hyperparameter search is not represented.

X. CONCLUSION

In this study, we have described a CI/CD-ready defect prediction pipeline using LightGBM and its applicability in supporting risk-sensitive regression testing.

The study proved that LGBM has the same predictive performance as XGBoost, but with significantly reduced training and scoring latency on actual defect data.

This work has made the most important contribution, which is to overcome one of the main obstacles to practical adoption: computational overhead. This study will allow accurate prediction of defects to be introduced as a consistent and predictable part of continuous testing processes by demonstrating that high-quality prediction is possible with even restricted CI/CD time limits. Notably, the findings indicate that prioritising timeliness does not necessitate sacrificing predictive reliability.

The findings formed the basis of additional application of fast defect prediction to more elaborate regression testing methods, such as blended and diversity-aware test case prioritisation methods. On a larger scale, the article emphasizes the importance of judging software analytics methods not just based on the theoretical precision of the method, but also based on their capacity to work in a sustainable manner in actual development pipelines. In doing so, it contributes toward making risk-aware quality assurance a practical reality in modern CI/CD environments.

DECLARATIONS

All authors declare that they have no conflicts of interest.

REFERENCES

- [1] J. J. C. George and D. P. Augustine, "Automation of test case prioritisation: a systematic literature review and current trends," *Journal of Theoretical and Applied Information Technology*, vol. 101, no. 3, pp. 1381–1387, 2023.
- [2] B. Busjaeger and T. Xie, "Learning for test prioritisation: an industrial case study," in *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, 2016, pp. 1–10, doi: 10.1145/2950290.2983954.
- [3] G. Chaurasia, S. Agarwal, and S. S. Gautam, "Clustering-based novel test case prioritisation technique," in *Proceedings of the IEEE Students Conference on Engineering and Systems (SCES)*, 2015, pp. 45–50.
- [4] R. K. Saha, L. Zhang, K. Khurshid, and D. E. Perry, "An information retrieval approach for regression test prioritisation based on program changes," in *Proceedings of the IEEE/ACM 37th International Conference on Software Engineering (ICSE)*, 2015, pp. 268–279, doi: 10.1109/ICSE.2015.47.
- [5] J.-H. Kwon, I. Y. Ko, G. Rothermel, and M. Staats, "Test case prioritisation based on information retrieval concepts," in *Proceedings of the 21st Asia-Pacific Software Engineering Conference (APSEC)*, 2014, pp. 19–26, doi: 10.1109/APSEC.2014.12.
- [6] S. Abele and P. Göhner, "Improving proceeding test case prioritisation with learning software agents," in *Proceedings of ICAART*, vol. 2, 2014, pp. 293–298.
- [7] X. Wang and H. Zeng, "Dynamic test case prioritisation based on multi-objective optimisation," in *Proceedings of the 15th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*, 2014.
- [8] M. N. Minhas et al., "Lessons learnt from replicating a study on information-retrieval-based test case prioritisation," *Software Quality Journal*, vol. 31, no. 4, pp. 1527–1559, 2023.
- [9] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object-oriented design," *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476–493, Jun. 1994, doi: 10.1109/32.295895.
- [10] M. Hall, D. Bowes, and D. Gray, "A systematic literature review on fault prediction performance in software engineering," *IEEE Trans. Softw. Eng.*, vol. 38, no. 6, pp. 1276–1304, Nov.-Dec. 2012.
- [11] M. J. Arafeen and H. Do, "Test case prioritisation using requirements-based clustering," in *Proceedings of the IEEE Sixth International Conference on Software Testing, Verification and Validation*, 2013.
- [12] R. Kalyani et al., "Test case prioritisation using requirements clustering," *International Journal of Applied Engineering Research*, vol. 13, no. 15, pp. 11776–11780, 2018.
- [13] J. Badwal and H. Raperia, "Test case prioritisation using clustering," in *Proceedings of the IEEE Sixth International Conference*, 2013.
- [14] M. Mostafa, S.-H. Mirian-Hosseinabadi, and M. Mahdiah, "Test case prioritisation using test case diversification and fault-proneness estimations," *Automated Software Engineering*, vol. 29, no. 2, p. 50, 2022.
- [15] R. Aggarwal and K. Kaur, "Machine learning approaches in software fault prediction: A review," *Cureus Journal of Computer Science*, vol. 3, Jan. 2026, doi: 10.7759/s44389-026-00021-7.
- [16] R. Lachmann et al., "System-level test case prioritisation using machine learning," in *Proceedings of the IEEE 15th International Conference on Machine Learning and Applications (ICMLA)*, 2016.
- [17] R. Lachmann, "Machine learning-driven test case prioritisation approaches for black-box software testing," in *Proceedings of the European Test and Telemetry Conference*, 2018.
- [18] R. Pan et al., "Test case selection and prioritisation using machine learning: a systematic literature review," *Empirical Software Engineering*, vol. 27, no. 2, p. 29, 2022.
- [19] H. Spieker et al., "Reinforcement learning for automatic test case prioritisation and selection in continuous integration," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2017.
- [20] S. Omri and C. Sinz, "Learning to rank for test case prioritisation," in *Proceedings of the 15th Workshop on Search-Based Software Testing*, 2022.