The *Why*, *When*, *What*, and *How* About Predictive Continuous Integration: A Simulation-Based Investigation

Bohan Liu, He Zhang, Weigang Ma, Gongyuan Li, Shanshan Li, and Haifeng Shen, *Senior Member*

***Abstract*—Continuous Integration (CI) enables developers to detect defects early and thus reduce lead time. However, the high frequency and long duration of executing CI have a detrimental effect on this practice. Existing studies have focused on using CI outcome predictors to reduce frequency. Since there is no reported project using predictive CI, it is difficult to evaluate its economic impact. This research aims to investigate predictive CI from a process perspective, including why and when to adopt predictors, what predictors to be used, and how to practice predictive CI in real projects. We innovatively employ Software Process Simulation to simulate a predictive CI process with a Discrete-Event Simulation (DES) model and conduct simulation-based experiments. We develop the Rollback-based Identification of Defective Commits (RIDEC) method to account for the negative effects of false predictions in simulations. Experimental results show that: 1) using predictive CI generally improves the effectiveness of CI, reducing time costs by up to 36.8% and the average waiting time before executing CI by 90.5%; 2) the time-saving varies across projects, with higher commit frequency projects benefiting more; and 3) predictor performance does not strongly correlate with time savings, but the precision of both failed and passed predictions should be paid more attention. Simulation-based evaluation helps identify overlooked aspects in existing research. Predictive CI saves time and resources, but improved prediction performance has limited cost-saving benefits. The primary value of predictive CI lies in providing accurate and quick feedback to developers, aligning with the goal of CI.**

***Index Terms*—Continuous integration, machine learning, software process simulation, discrete-event simulation.**

I. Introduction

CONTINUOUS Integration (CI) is a software development practice that automates the compilation, building, and testing of source code on dedicated servers, which makes it possible to accelerate development whilst securing quality. The benefits of CI, such as reducing merge conflicts, have been widely recognized [1], [2], [3], [4]. The annual GitLab Global Developer Survey[1] shows that only 10% respondents did not use CI tools. The Annual State of DevOps Survey[2] also asserts that "*CI is a must-do in the DevOps space, right after version control becomes ubiquitous*".

Some studies have identified the challenges practitioners are facing. A systematic literature review [5] suggests that the most critical testing problem in CI is the excessive consumption of time. Since developers need to wait for the CI process to complete before engaging in other development activities, a long duration of CI means not only the consumption of substantial computing resources, but also human resources [6], [7], [8]. Fowler indicated that every minute of reducing CI execution time is a minute saved for developers every time they commit code [9]. Among the 104,442 CI executions over 67 projects in TravisTorrent, 40% took more than 30 minutes [10], implying the consumption of tens of thousands of man-hours. Furthermore, the long duration of CI makes it impractical to conduct frequent CI executions, and developers are reported to often bypass some CI executions to save time [11].

Therefore, there are clear benefits of predicting the CI outcomes based on which the frequency of executing CI scripts can be reduced. Hassan and Zhang [12] pioneered the application of machine learning techniques to predict a CI outcome, which can be either *passed* or *failed*. A *passed* result means no defect found by all the scripts executed, whilst a *failed* means that defect(s) exist and need to be located and fixed. Predictive CI is to forecast the result of the current CI using a predictor trained with a set of features pertaining to the context of CI. By skipping the CIs predicted as *passed*, the code will be merged directly into shared mainline code repository. This method can potentially improve efficiency by reducing the number of CI executions while maintaining the number of test cases in the CI

Manuscript received 11 January 2023; revised 20 October 2023; accepted 25 October 2023. Date of publication 10 November 2023; date of current version 12 December 2023. This work was supported in part by the National Natural Science Foundation of China under Grants 62072227, 62202219, 62302210, and 72372070, in part by the Jiangsu Provincial Key Research and Development Program under Grant BE2021002-2, in part by the National Key Research and Development Program of China under Grant 2019YFE0105500, and in part by the Innovation Project and Overseas Open Projects of State Key Laboratory for Novel Software Technology (Nanjing University) under Grants ZZKT2022A25, KFKT2022A09, and KFKT2023A09. Recommended for acceptance by R. Kazman. *(Corresponding author: He Zhang.)*

Bohan Liu, He Zhang, Weigang Ma, Gongyuan Li, and Shanshan Li are with the State Key Laboratory of Novel Software Technology, Software Institute, Nanjing University, Jiangsu 210093, China (e-mail: dg1732003@smail.nju.edu.cn; hezhang@nju.edu.cn; mf21320110@smail.nju.edu.cn; gyli@smail.nju.edu.cn; lss@nju.edu.cn).

Haifeng Shen is with Peter Faber Business School, Australian Catholic University, Sydney, NSW 2060, Australia (e-mail: Haifeng.Shen@acu.edu.au).

Digital Object Identifier 10.1109/TSE.2023.3330510

[1]https://about.gitlab.com/developer-survey

[2]https://puppet.com/resources/report/2018-state-devops-report

script. In other words, the quality standard of each CI execution is retained at the expense of reduced number of CI executions.

However, since its inception over a decade ago, no attempt has been reported to adopt predictive CI in real projects. One possible reason is due to the lack of commonly accepted predictors or available prediction systems that work in different projects in practice. Existing research [12], [13], [14], [15], [16], [17], [18], [19] has focused only on identifying suitable predictors for different projects in terms of prediction accuracy using common machine learning performance evaluation metrics such as recall, precision, and F measures.

Another possible reason, which is probably more critical, is that there is no evidence available to support that the prediction is able to actually improve the efficiency of the CI process in terms of cost saving. The effectiveness of predictive CI is subject to a variety of factors and performance of a predictor is only part of the concern. For example, if a defect-free CI execution is predicted as *passed*, the time cost of that CI execution is saved. However, if a defective CI execution is predicted *passed*, it would be mixed with the other future commits for testing at a later stage, which would likely incur higher costs to locate defects. Skipping will cause multiple commits to be tested together, which will undoubtedly cause an extra time cost for locating defects. A crucial question is how to determine the performance threshold of a predictor to be economically effective in predictive CI.

What makes the evaluation even more challenging is that the proportions of *failed* and *passed* in existing projects are imbalanced [18], [20] and evaluation of imbalanced learning is a thorny issue. Metrics such as Recall or F-measure tend to only evaluate the predictive performance of one category whilst comprehensive indicators such as Accuracy, and Area Under Curve may also fail [21]. In reality, which metrics should be included in the evaluation, and how to understand the levels of the metrics? To the best of our knowledge, no empirical endeavor has been reported to evaluate the impact of predictors on the CI process. Due to the lack of evidence, developers are reluctant to take the risk of adopting predictive CI in their projects and the unavailability of empirical data makes it impossible to perform meaningful evaluation.

This paper reports the first effort that attempts to break this paradox by proposing a novel approach that harnesses Software Process Simulation to investigate predictive CI from a process perspective, including *why* and *when* to adopt predictors, *what* predictors to be used, and *how* to practice predictive CI in real projects, which would provide valuable insights to assisting developers in making informed decisions on the adoption of predictive CI. In software process research, simulation is one of the most effective methods when it is difficult or expensive to achieve the research purpose through real cases. It can help researchers in an approximate way to study a software process without investing considerable effort on experimentation with actual software projects. Simulation has been widely used since its introduction into the software process area in the 1980s [22]. In this work, we conduct simulation experiments to evaluate the impact of predictors on the CI process using a Discrete-Event Simulation (DES) model calibrated by real CI process data from projects in GitHub. The DES model is called CISimulator [23], which specifically simulates the continuous integration process supported by Travis CI[3] with the exception of integrated predictors. Projects using Travis CI are the usual research subjects in existing studies [14], [17], [18], [20], [24], [25], [26].

Furthermore, we propose a Rollback-based Identification of Defective Commits (RIDEC) method to locate one or more commits that cause the failure from the test results of multiple commits. This method is a key complement to predictive CI because it shifts the burden of locating defects caused by false CI prediction back to servers. Without this method, developers are likely to be deterred from adopting predictive CI as it saves server resources at the expense of increasing their workload. In addition, the simulation of RIDEC is important to evaluate the effectiveness of predictive CI from a process perspective.

Through a series of simulation experiments, we have investigated the *why, when, what*, and *how* (*3W1H*) aspects of predictive CI. For *why*, our results reveal that predictive CI is able to save time in most cases, including the time for executing CI and the waiting time before executing CI. For *when*, the time saving resulted from using predictive CI varies across projects, however, projects with higher commit frequency but not so high CI failure rate tend to save more time. For *what*, it is safe to use predictive CI if predictors can achieve a relatively high level of comprehensive predictive performance. Specific prediction performance metrics (e.g., accuracy) related to time savings vary across different projects. In general, while ensuring a similar level of time cost savings, it is advisable to pursue higher accuracy in order to maximize the ability of the predictor to provide fast feedback. For *how*, the CISimulator proposed in this study can provide support for the research and practice of predictive CI by offering a simulation environment for developers to make informed decisions, which is publicly available on GitHub [23].

The main contributions of this study are as follows:

- We propose a novel simulation-based approach to systematically investigating predictive CI from a process perspective, including *why* and *when* to adopt predictors, *what* predictors to be used, and *how* to practice predictive CI in real projects.
- We propose the RIDEC method for locating the defective commits from multiple commits that are tested together to account for the negative effect of false predictions.
- We build a DES model called CISimulator that can simulate CI processes with and without predictive CI. The CISimulator can be used to evaluate the time saving in the CI process by applying the predictive CI method. We publicly share the simulation environment together with relevant instructions to assist developers in making informed decisions on the adoption of predictive CI.
- We conduct simulation-based experiments using the data collected from real projects to investigate *why* and *when* to use *what* predictors.

The remainder of this paper is structured as follows. Sec. II explains the predictive CI method, the problems of its evaluation in related work, as well as the motivation of using simulation. Sec. III discusses the overview of the methodology and depicts CISimulator. The verification and validation of

[3]https://www.travis-ci.org/

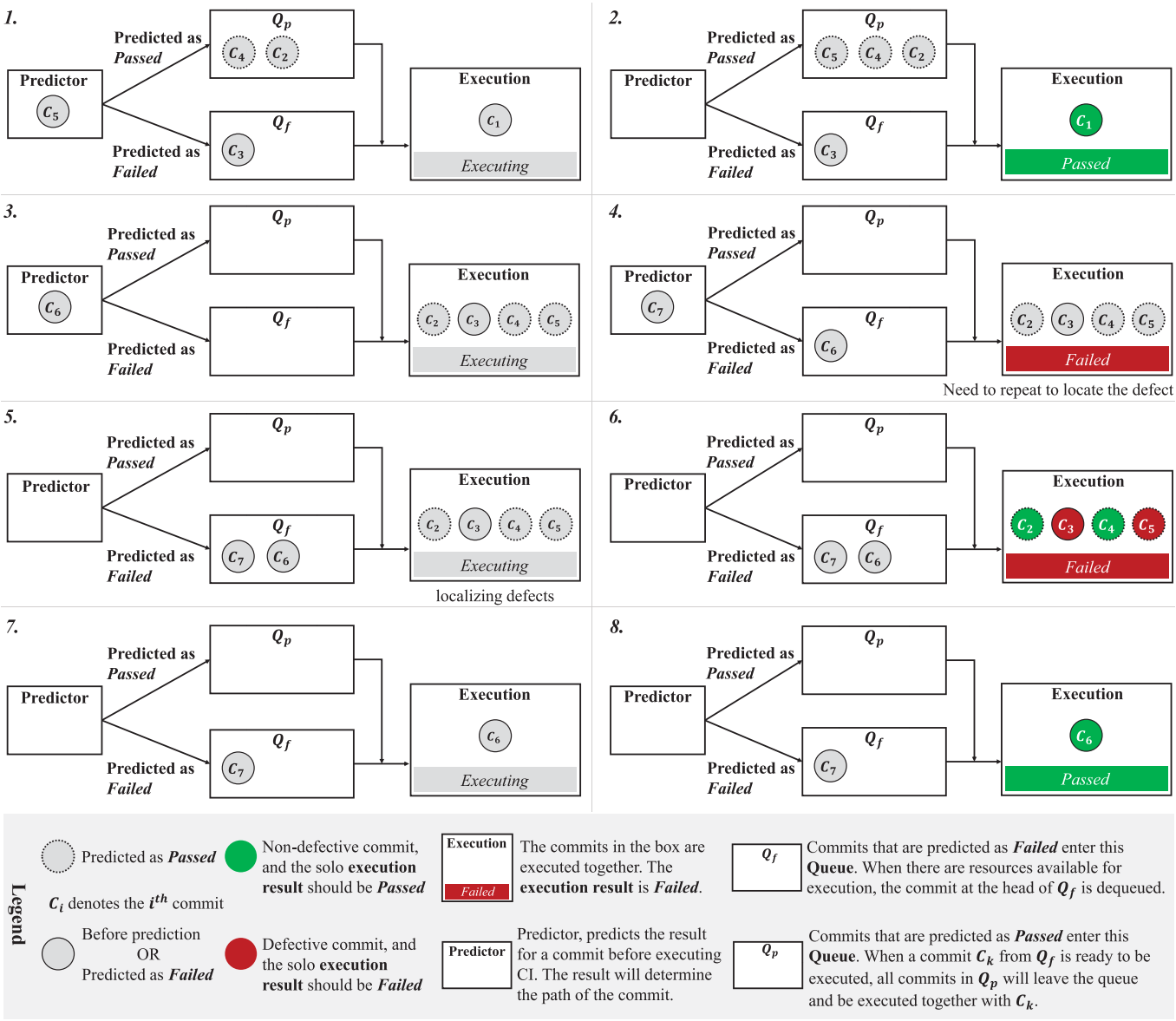


Fig. 1. An example of the CI process with predictive CI.

CISimulator are discussed in Sec. IV. Sec. V delineates the details of our experimental study, and Sec. VI analyzes results. Sec. VII discusses the practical implications of predictive CI and the benefits of simulation. Sec. VIII discusses the threats to validity, followed by the conclusions in Sec. IX.

II. Background and Related Work

This study aims to use a simulation-based approach to investigating the effectiveness of predictive CI from a process perspective. This section introduces what predictive CI is, the existing problems in evaluating predictive CI in related work, and the related work on software process simulation.

A. What Is Predictive CI?

Predictive CI is a binary classification problem as only *passed* and *failed* results are considered in existing research [12], [13], [14], [15], [16], [17], [18], [19]. The *passed* means exit codes for all steps are zero, while the *failed* means the script returns a non-zero exit code. Other results (*canceled* and *errored*) are caused by human operation errors or configuration environment problems and occur before the execution of the CI script. Hence they are not considered.

Without predictive CI: the commit will be allocated to execute the CI, that is to carry out activities such as build, code scanning and testing. If the CI result is *passed*, the code change is merged into the trunk. Otherwise, the developer needs to fix the defects and re-submit the code change.

With predictive CI: the process varies according to different application scenarios. For example, to coordinate limited server resources, CI can be executed preferentially for the commits that are predicted to be *passed*. This study discusses a scenario that is more recommended in existing research, that is, skipping the execution of CI for commits that are predicted as *passed*. Fig. 1 presents an example of eight continuous scenes to illustrate the process with predictive CI.

The 1^{st} scene: before the new commit C_5 triggers the CI, the predictor predicts the result of executing CI for this commit. C_3 and C_2 are the commits that were predicted as *passed* before, which are waiting in the queue Q_p for the next execution of CI. C_3 is the commit that was predicted as *failed* before, which is waiting in the queue Q_f for an available server to execute the CI. C_1 is executing the CI.

The 2^{nd} scene: C_5 was predicted as *passed* and hence entered Q_p. The execution of C_1 is finished and the result is *passed*, which means a wrong prediction before.

The 3^{rd} scene: the new commit C_6 appeared and its result of CI execution is being predicted. As C_1 leaves, C_3 which is waiting in Q_f gets an available server. C_2, C_4, and C_5 have been waiting for this moment, and now they can execute CI together with the C_3 as a batch.

The 4^{th} scene: the new commit C_7 appeared and C_6 entered Q_f. The execution result of the batch is *failed*. Since it is not known which commits caused the failure, it needs to locate the defect. Repeating CI by rolling back is an effective method, such as RIDEC proposed in this paper. The specific process of RIDEC will be introduced in Sec. III-C.

The 5^{th} scene: C_7 entered Q_p. The batch is executing the CI for localizing defects.

The 6^{th} scene: The defective and non-defective commits are identified.

The 7^{th} scene: Since the server was released and the Q_p was empty, C_6 is executing the CI alone.

The 8^{th} scene: The execution of C_6 is finished and the result is *passed*.

Table I summarizes existing research on machine learning-based predictive CI. TravisTorrent [27] is a build log data set that has been mainly adopted in recent years. All the latest studies [18], [20], [24], [25], [26] considered both class imbalance and time-series problems. Class imbalance refers to the problem that the number of *failed* is far less than that of *passed* in common build log data sets. The class imbalance would significantly affect the prediction performance. Time series refers to the problem that the chronological order of build logs must be maintained. If the data set is shuffled, the validation results would deviate from the real situation.

Hassan et al. [12] pioneered the use of machine learning methods to predict CI results. By analyzing the logs, they identified *social factors*, *technical factors*, *coordination factors*, *previous authentication factors*, and other variables that affect the CI results. They compared the decision tree built by different combinations of these features (factors) using large-scale IBM projects. The shuffle split validation was applied to the evaluation. The results showed that training with all features achieved the best performance. However, the Recall of predicting *failed* ($R_f = 0.69$) is significantly worse than the Recall of *passed* ($R_p = 0.95$).

Wolf et al. [13] were the first to use communication structure measures to predict CI results. They built a Naive Bayes classifier using the social network information of collaboration and interaction between developers and evaluated the predictor with leave-one-out validation using the data from IBM Jazz^{TM} project. The R_f of the predictor ranged from 0.55 to 0.75. Schröter [28] extended Wolf et al.'s method with technical relationships. A support vector machine is trained using the social networks that depict the coordination behavior of developers involved in a CI execution as input to predict the CI execution outcome. Both R_f and Precision of predicting *failed* (P_f) have been significantly improved as the minimum R_f is increased to 0.65. However, these predictors are difficult to generalize to other projects since not every project can provide traceability between communication artifacts and development artifacts.

Finlay et al. [29] applied the Hoeffding Tree classification method, which uses a set of code metrics as input to predict CI results. They used the IBM Jazz^{TM} project to compare Hoeffding Tree and C4.5. The results show that the former performs better, with an overall accuracy rate at 72%, but it is difficult to accurately predict the *failed*. They indicated that the projects they used did not store a complete history record in the software repository, which may affect the prediction performance. In addition, they also pointed out that only a few features may have a significant impact on prediction performance, but the study did not perform feature selection before training.

TravisTorrent data set has been widely used in CI-related research in recent years. Two studies [14], [17] used similar features but evaluated different classifiers. Their results showed that prediction performance varied greatly from project to project and was not good enough. Hassan and Wang [15] built a predictor using a Random Forest algorithm with the metrics available in TravisTorrent and Java code metrics as input. The predictor can predict the *failed* with an average $F\text{-}measure$ over 0.87. However, the results have not been generalized to the projects other than large-scale Java projects and only three projects were used for the evaluation of predictors. The code metrics they used only apply to Java. They used an Eclipse plug-in JGit[4] to identify code changes, and then obtain code metrics based on GumTreeDiff [30]. For projects that use multiple development languages, these code metrics are not applicable.

Chen et al. [20] and our previous work [18] almost simultaneously identified the limitations of existing research on the training and validation of predictors. We conducted an empirical study [18] to provide a comprehensive evaluation for predictive CI in terms of the two data characteristics, i.e. time series and class imbalance. The experiments indicate that the cross-validation may not be applicable to the predictive CI due to the time-series features of the CI process. Furthermore, considering the higher value of predicting the *failed* in practice, F_2-Measure was used instead of F_1-Measure. These results indicate that imbalanced learning can significantly improve the prediction performance, especially for predicting *failed*. However, the performance of existing predictors on different projects was not good enough to provide practitioners with sufficient confidence in practicing them. Chen et al. [20] used more features and applied feature selection for training an imbalanced learning classifier, XGBoost [31]. Instead of using cross-validation, they split the CI executions into the training and test sets with a ratio of 3:1 without disturbing the order

[4]https://eclipse.org/jgit/

TABLE I
SUMMARY OF EXISTING RESEARCH ON PREDICTIVE CONTINUOUS INTEGRATION

Study	Data sets	Imbal.[1]	Validation	Time-series[2]	Results[3]
[12]	IBM Toronto Labs		Shuffle split		$R_f = 0.69$, $R_p = 0.95$
[13]	IBM Jazz^{TM}		Leave-one-out		$R_f \in [0.55, 0.75]$, $P_f \in [0.50, 0.76]$
[28]	IBM Jazz^{TM}		Shuffle split		$R_f \in [0.65, 1.00]$, $P_f \in [0.90, 0.96]$
[29]	IBM Jazz^{TM}		K-fold cross-validation		$R_f = 0.65$, $A = 0.72$
[15]	Ant, Maven, Gradle		K-fold cross-validation		$R_f = 0.82$, $P = 0.85$, $F_1 = 0.83$, $R_p = 0.93$, $P_p = 0.92$, $F_{1p} = 0.92$
[17]	532 OSS projects available on TravisTorrent		Shuffle split		$R_f = 0.69$, $A = 0.74$
[14]	126 OSS projects available on TravisTorrent		K-fold cross-validation, On-line prediction	✓	K-fold cross-validation: $F_{1f} > 0.6$; On-line prediction: More than 50% projects achieved $F_{1f} < 0.3$.
[18]	14 OSS projects available on TravisTorrent, and 4 OSS projects from Gitlab	✓	Time-series-validation, K-fold cross-validation	✓	Time-series-validation: $F_{2f} = 0.50$; K-fold cross-validation: $F_{2f} = 0.53$
[20]	20 OSS projects available on TravisTorrent	✓	Time-series-validation	✓	$F_{1f} = 0.47$, $F_{1p} = 0.91$
[24], [25]	359 OSS projects available on TravisTorrent	✓	Quasi-simulation	✓	Save 61% of cost, 73% of *failed* commits were detected without delay, 27% of *failed* commits were delayed detected within two CI executions on average.
[26]	10 OSS projects available on TravisTorrent	✓	Time-series-validation	✓	$AUC = 0.69$, $balance = 0.66$
This work	6 OSS projects on GitHub that using TravisCI	✓	Simulation-based validation	✓	Ref. VI

[1] Whether the study evaluated predictors with imbalanced learning techniques. The symbol ✓denotes imbalanced learning was used.
[2] Whether the study considered the time-series issue in validation. The symbol ✓denotes the chronological order of the data is maintained in validation.
[3] We only present the best performance achieved in each study.

considering the time-series concern. They compared their approach with existing methods [15], [17], [32] and indicated that their approach can significantly improve the best of the state-of-the-art approaches by 47.5% in terms of F_1-Measure of *failed* (F_{1f}). However, their approach only achieved an F_{1f} of 0.47 on average. Converted to F_{2f}, it only reaches 0.448. Of the 67 predictors evaluated in [18], 13 predictors were able to achieve an F_2-Measure of *failed* (F_{2f}) over 0.448. In addition, the best one can achieve 0.498.

Jin and Servant [24], [25] proposed a novel approach Smart-BuildSkip, which only predicts when the result of the last CI execution is *passed*. They adopted a quasi-simulation to evaluate their approach. For the evaluation, they counted the percentage of skipped CIs as a measurement of the time cost savings; they also measured the delay length of the *failed* CI, which is the number of CIs that were skipped until the predictor decided to execute CI again for a skipped *failed* CI.

Saidani et al. [26] took a different approach and proposed a search-based method, i.e. Multi-Objective Genetic Programming. This method can mitigate the impact of class imbalance and their experiments show that it is superior to machine learning predictors that generally do not consider imbalance. The study used time-series validation.

There is currently no real-world cases of predictive CI being reported yet, although manual skipping without using prediction is a very common phenomenon. The readiness of predictive CI remains doubtful based on the experimental results from existing research. Currently, there are significant variations in the performance of predictive CI across different projects, and its performance is also unstable within the same project. The potential benefits and the amount of benefit that can be obtained from predictive CI are economically unknown at present. In other words, from a practical point of view, in a dynamic development process, what kind of predictor should be selected (*what*)? when the prediction could be used (*when*)? and what effect would be obtained (*why*)? have to be examined. The root of research on these issues lies in effective evaluation.

B. Problems With Evaluating Predictive CI

All the related work published by 2020 [12], [13], [14], [15], [17], [28], [29] and [26] used the common metrics in machine learning for performance evaluation. Our previous experiments [18] demonstrated that the timing problem cannot be ignored in predictor evaluation, and recommended the time-series validation method. However, our previous work also used those common classifier metrics and did not evaluate the predictor from a process perspective. In [20] a simple statistical metric, namely the ratios of CIs would be skipped, was used to evaluate the time cost savings that can be saved by using predictions. However, the metric does not reflect the possible negative effects of wrong predictions. None of the above studies has solved a key issue of evaluation, a question that practitioners really care about, that is, what is the impact of CI prediction on a dynamic software development process, and what benefits can be obtained.

Jin et al. [24], [25] took a step forward. They used a single time-line quasi-simulation method (the real simulation should be a computer-executable model, which can be repeatedly configured and repeatedly experimented [33], [34]) for evaluation. The quasi-simulation simulates real scenarios based on real data and a time-series validation method. They keep the actual order of the real data, making predictions one by one for each build. The quasi-simulation method ignores the complexity of the CI process and did not consider several issues such as the time interval between commits, the queuing time, etc. The using of prediction would change the order and results of CI. The quasi-simulation cannot effectively simulate these scenarios. Fig. 2 shows an example of a scenario where quasi-simulation can be problematic. For ease of understanding, we use the same

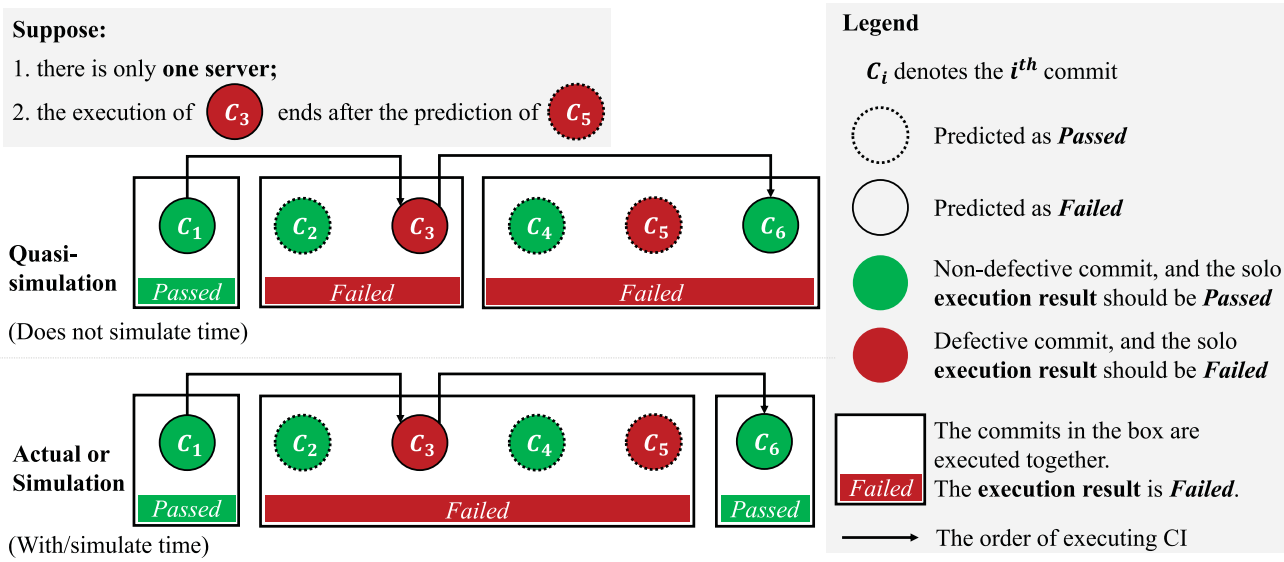


Fig. 2. The comparison between the actual situation (can be simulated using process simulation) and the quasi-simulation.

example as in Fig. 1. Assuming that there is only one server, the completion time of the execution for C_3 is later than the completion time of the prediction for C_5. C_3 is predicted as *failed* and it will trigger the CI execution. The quasi-simulation does not simulate execution time. Therefore, after the execution for C_1 ends, CI will be executed on a batch that contains C_3 and all the commits skipped before C_3, namely C_2. In fact, when the execution for C_3 starts, C_4 and C_5 have already been skipped. That is, C_2 is waiting in the queue along with C_4 and C_5 for the execution. In quasi-simulation, the next execution after C_3 contains C_4, C_5 and C_6. Actually, C_6 should execute CI alone as C_4 and C_5 should appear in the previous execution (together with C_3). The introduction of predictive CI would change the consumption of server resources as well as the order of executions. Since quasi-simulation does not simulate time, it cannot accurately simulate the execution sequence and the commits involved in an execution. Furthermore, it may generate different execution results. The widely used common simulation paradigms, such as those based on the DES paradigm, can simulate time to avoid this problem.

Neither the simple statistics [20] nor the quasi-simulation [24], [25] can accurately simulate the real scenarios and comprehensively evaluate the effects of applying predictive CI in development processes. The use of predictive CI would have impacts on the dynamic process, and such impacts are not only caused by the prediction itself, but also by other factors in the dynamic process, such as the duration of execution and the availability of server resources, etc.

In general, there are two problems with existing evaluation methods (including quasi-simulation). First, the dynamic nature of the process was not considered. The order and results of CI triggers are no longer the same as they appear in historical data. As depicted in Fig. 2, the execution time of CI also affects the order of CI execution and commits involved in a CI, whereas quasi-simulations consider only one of the most noticeable factors, namely the commits involved in a CI. Second, the potential negative impact of the predictive CI has not been effectively evaluated, and the final effect after combining the gain and loss has not been effectively measured. If we focus only on CI, it is easy to conclude that skipping some CI executions would save time and server resources. However, if the use of predictive CI increases the workload of developers in locating defects, there is clearly a possibility that the gains outweigh the losses. Also, developers are likely reluctant to adopt an approach that would increase their workload.

The existing problems hinder the wider adoption of predictive CI in practice. They also make it difficult for researchers to truly understand the directions in which predictive CI can be optimized. These motivate us to propose a simulation-based approach to systematically investigating predictive CI from a process perspective.

C. The Software Process Simulation-Based Evaluation

The Software Process Simulation modeling is to build a computer-executable model to simulate the real dynamic software development process. The most common simulation paradigms include System Dynamics (SD), Discrete Event Simulation (DES), and Agent-Based Simulation (ABS). The most critical shortcoming of the quasi-simulation used in [24] compared with these common simulation paradigms is the lack of quantification of time.

Since Abdel-Hamid and Madnick [22] introduced Software Process Simulation to Software Engineering (SE) in the 1980s, there have been a large number of studies published in the community. The value of software process simulation in understanding the real process and supporting the planning, decision-making, and optimization of software development has been widely recognized [35], [36]. With the improvement of automation and the ability to observe and analyze data, the importance of simulation as an empirical method in software engineering is increasing [37].

Simulation-based evaluation has demonstrated its powerful capabilities for evaluation in various problems that are difficult to be directly validated in practice. Baum et al. [38] simulated the agile development process for comparing pre-commit reviews and post-commit reviews. Their model can help researchers analyze the factors influencing the choice of the two review practices. Garousi and Pfahl [39] evaluated the performance of different testing processes with different degrees of automated testing activities using a simulation model. The model helped decision-makers determine whether and to what degree the company should automate its test processes to achieve a balance of quality and efficiency. Liu et al. [40] used simulation to compare the proposed Incremental V-model process and traditional V-model process for automotive development since it is almost impossible to experiment a new development process in real automotive development without the validation results of the simulation. However, there is no research on using simulation for the evaluation of machine learning-based predictors in SE and as such the approach proposed in this work is novel.

In other disciplines, simulation-based evaluation for predictors is also a novel direction. The simulation was suggested as a trending method for evaluating machine learning predictors in the research of Industry 4.0. However, real examples are still lacking [41]. In the context of production planning, simulation was used to generate forecast data and evaluate predictors for customer orders and demand forecasts [42]. However, the predictors evaluated in the study are based on the statistical method instead of machine learning. To our best knowledge, there is no research on simulating machine learning-based predictors (which involves how to simulate predictive performance) let alone simulating a process that is integrated with predictors (which involves relationships between the predictor and the activities around that either have impacts on it or are influenced by it).

III. Methodology

Although common machine learning metrics can objectively evaluate prediction performance based on a confusion matrix, it is unknown what they mean to the CI process. Besides, it is costly and potentially disruptive to evaluate predictors using real projects, and even more so, it would be impossible to evaluate and compare multiple predictors using real projects in a single study as it is too difficult to control variables. If we use statistical methods to analyze the cost savings of using predictive CI as in the study [20], the evaluation may not accurately reflect the impact of dynamic processes. Their work ignored an important issue, that is, skipping executions of CI causes multiple commits to be tested together, which incurs additional costs of locating bugs, some of which were not recently introduced. Apart from that, since CI is a continuous process and predictions have an impact on subsequent commits, it is also not reasonable to consider each CI execution in isolation when evaluating the time cost saved by using predictors. In the CI process with prediction, various dynamic factors are involved, such as the varying time interval of commit generation and server resource availability. Process simulation has been recognized as an affordable means to more accurately analyze this dynamic process like this. In addition to the above benefits, simulation also has the advantage of being easy to replicate and extend. It can be applied to different projects/processes, and it is easy to implement the Monte Carlo method [43].

A. Methodology Overview

The goal of our research is to build a software process simulation model to investigate the effect of predictors in a CI process. An overview of our methodology is shown in Fig. 3, which mainly contains five phases as follows.

First, data acquisition and processing: we selected representative GitHub projects. For each project, we obtained logs related to the continuous integration process, including code commit logs and build logs recorded in Travis CI, and established associations between logs based on commit IDs. These logs are used to train/test CI predictors and build the process simulation model of the CI process.

Second, training and testing CI predictors: we replicated our previous work [18] using the newly acquired data. The results of the prediction performance of each predictor are used as calibration variables in the simulation model.

Third, construction and verification of CISimulator: we adopted the DES paradigm for modeling the CI process (cf. Sec. III.B) for the reasons. The specific construction process and introduction of the model are detailed in Sec. III.E. Predictive CI would cause some builds to be skipped, and wrong predictions may increase the burden of defect location. We proposed the RIDEC method to identify all the defective commits in a batch through non-full repeated CIs. That is, if RIDEC is used, the cost of localizing defects in commits remains unchanged regardless of whether predictive CI is used or not. We simulated RIDEC in CISimulator and the details are elaborated in Sec. III.C. The details of model verification are introduced in Sec. IV.

Fourth, calibration and validation of the CISimulator: strictly speaking, the construction, calibration, verification and validation is an iterative process rather than three independent steps. The construction and verification are intertwined, whilst validation relies on calibration. So we group these four steps into the third and fourth phases. We perform the statistical analysis for the log data to obtain calibration values for the model variables, where the calibration variables of CI predictors are derived from the second phase. The details of calibration and validation are given in Sec. III-F and Sec. IV respectively.

Last, simulation experiments: we performed simulation experiments to answer our research questions on the *why*, *when*, *what*, and *how* aspects of using predictive CI. To obtain enough samples under a certain degree of variable control, we randomly sliced the data of each project and used the data of each slice to calibrate CISimulator to conduct each simulation experiment. The details are given in Sec. V.

The CISimulator is built to simulate the CI process to validate the effectiveness of the predictor. The dynamic (simulation) process model is the computerized implementation of

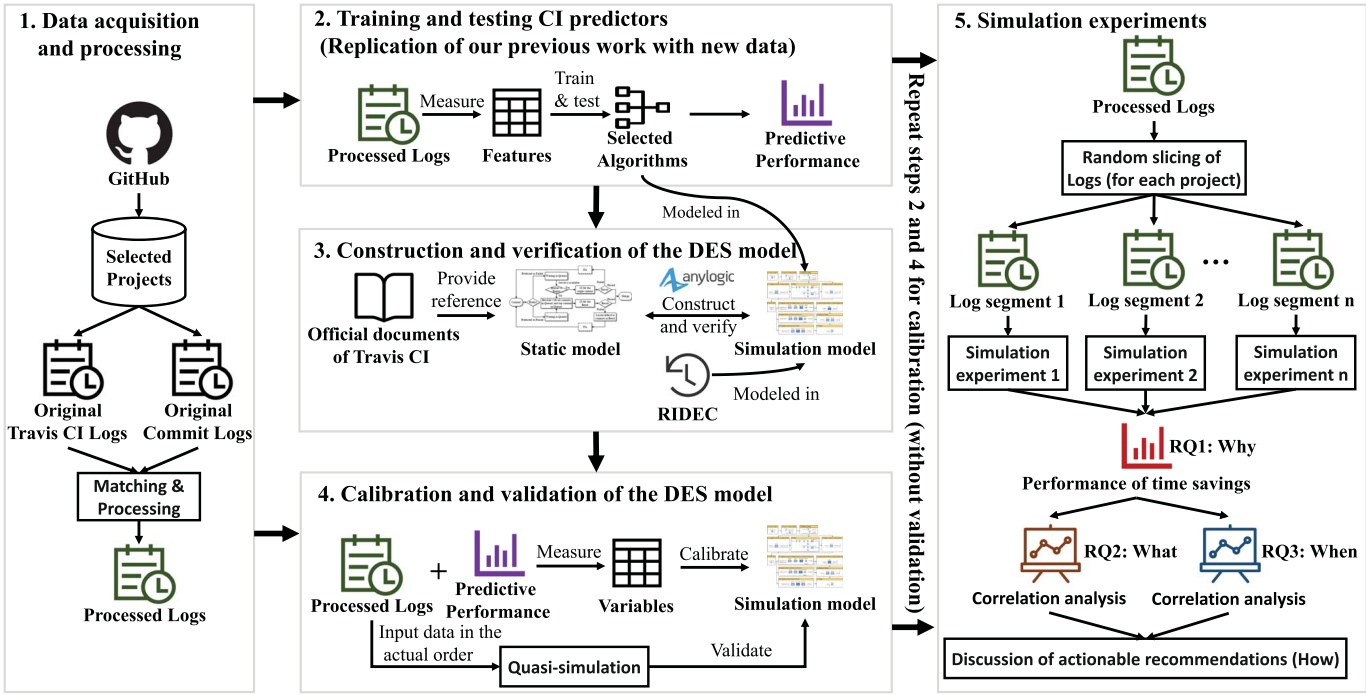


Fig. 3. Overview of the research methodology.

the descriptive (process) model. The descriptive model is an abstraction of the real world we learned and analyzed from the official documents of Travis CI. We further analyzed TravisTorrent data to revise and confirm the descriptive model. We build the model using *AnyLogic*[5] *(version 8.7.6)* and the components it provides. Other tools such as ExtendSim[6], can also be used for this purpose. Quantification of the simulation model is through calibration of variables obtained from real projects or empirical studies using statistical analysis. We verify and validate CISimulator from four quality aspects following existing frameworks [44], [45], [46]. The four quality aspects include syntactic quality, semantic quality, pragmatic quality, and performance. The quantitative comparisons with real data show that the model has a fidelity at 95% in simulating the number of created commits and a predictor's prediction results.

B. Selection of Simulation Paradigm

The most widely used paradigms for software process simulation are System Dynamics (SD), Discrete-Event Simulation (DES), and Agent-Based Simulation (ABS) [47]. These three paradigms are adept at different levels of abstraction [48], [49]. SD is mainly used to simulate the process of high abstraction with complex feedback loops and it simulates the dynamic process in a continuous-time manner. ABS can be applied to the process at the finest granularity level, and it is excellent at simulating the characteristics of individual agents (developers) and their interactions with each other. DES is applied at low to middle abstraction levels. DES simulates changes in the state triggered by discrete events, providing a better simulation of the step-by-step flow of entities through a system/process. The main advantage of a DES model is its ability to capture exact information at the actual process level and its ability to uniquely represent each work product of the development process by using attributes (e.g., the commit is a *push* or *pull request*) that are attached to each work product.

The CI process to simulate is one with a moderate level of abstraction and where state changes are mainly triggered by discrete events. In the CI process, after the developer commits the code, it is mainly handled by the Travis CI tool according to a certain workflow, and only the subsequent defect-fixing activity would involve developers. This process is more likely a business process and requires less portrayal of individual developers and their interactions. Besides, the model needs to perform a fine-grained simulation of the commit, including whether it is a *push* or *pull request*, and whether it is defective. Activities such as code committing, CI triggering, and execution of the CI are discrete events. Therefore, DES is more suitable for this study than both SD and ABS.

C. Rollback-Based Identification of Defective Commits (RIDEC)

The purpose of using predictive CI is to save time and resources by skipping some CI executions, which would cause skipped commits (predicted as *passed*) to be packaged with a triggering commit (predicted as *failed*) as a batch to execute CI. Then a failure means that one or more commit in the batch are defective. This makes predictive CI an anti-CI practice. The proposed RIDEC method can identify all the defective commits in a batch through non-full repeated CIs. Fig. 4 depicts the RIDEC with an example. For ease of understanding, it depicts the process of localizing defects of the 5^{th} scene in Fig. 1.

[5]https://www.anylogic.com/
[6]https://extendsim.com/

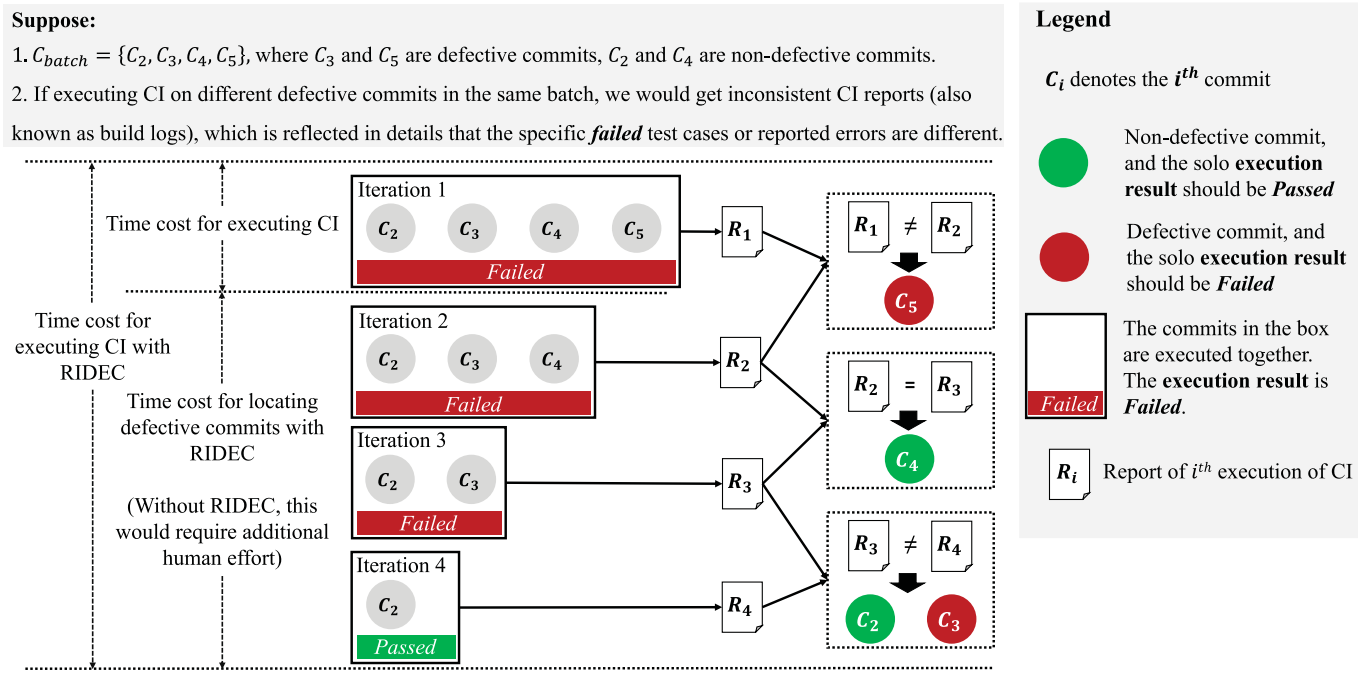


Fig. 4. The server automatically locates defective commits based on RIDEC.

Suppose $C_{batch} = \{C_1, C_2, ..., C_n\}$, where C_{batch} is the batch consists of commits from C_1 to C_n. The commits are created in chronological order, and C_1 is created earliest. If the execution result of C_{batch} is *failed*, the code will roll back to its precedent version (i.e. $C_{batch} = \{C_1, C_2, ..., C_{n-1}\}$) and execute CI again but with fewer commits. This process will be repeated until the result is *passed* or C_{batch} only has one commit. Each execution (iteration) would return a CI Report (also known as build log, denoted as R_i), which includes the test results of each test case. When all iterations are over, we can identify all defective commits by comparing the reports between the adjacent iterations.

As shown in Fig. 4, there are a total of four iterations, that is, the CI is executed 4 times. The results of iteration 1 and iteration 2 are both *failed*, which means that there is at least one defective commit in C_2, C_3, and C_4. By comparing the test results of each specific test case in R_1 and R_2, it can be found that the two are inconsistent, which can deduce that C_5 is a defective commit. By comparing the test results of each specific test case in R_2 and R_3, it can be found that the two are consistent, which can deduce that C_4 is a non-defective commit. The result of iteration 4 is *passed*, which can deduce that C_2 is a non-defective commit and C_3 should be a defective commit.

Fig. 4 shows an extreme case, that is, there are 4 commits in the initial batch, and a total of 3 additional executions (the first execution is bound to trigger) are required to identify all the defective commits. At the other extreme, all commits are non-defective, that is, the result of the first execution is *passed*. All batches would be tested only once.

Algorithm 1 shows how to calculate the total time spent by the server to localize defective commits in C_{batch} using RIDEC. C_{batch} is a queue. CI would be repeatedly executed until the result is *passed*. After each execution, the commit at the head of the queue would be dequeued. Since the execution duration of CI for different commits in C_{batch} may be different, for simplicity, we take the maximum duration as the execution duration of C_{batch}. The final execution time is the sum of all individual execution times.

Algorithm 1 Calculation of the time cost for the server to localize defective commits in C_{batch}.

[htb]

Input: C_{batch}: an array contains N commits awaiting execution of CI;

Output: T_{fl}: time cost for the server to localize the defective commits in C_{batch};

1: $T_{fl} \leftarrow 0$
2: $i \leftarrow N$
3: **repeat**
4: $t \leftarrow$ Max CI execution time of commit in $C_{batch}[1, i]$
5: $T_{fl} \leftarrow T_{fl} + t$
6: $i = i - 1$
7: **until** $i < 1$ **or** $C_{batch}[1, i]$ does not contain defective commits
8: **return** T_{fl};

D. Underlying Assumptions

The simulation model is an abstraction of the real world with assumptions.

The basic assumption here is that the effect on cost can be tolerated by changing the order of the real and predicted outcomes in the sequence of commits, that is, the research conclusion will not be affected by the change. Specifically, it is based on the following assumptions:

- When the probability distributions of numerical calibration variables (such as the creation time interval of commits,

TABLE II
DESCRIPTIONS OF DES MODEL COMPONENTS IN ANYLOGIC

Components	Icons	Descriptions
Source		To generate agents (indicate the starting point of a process model)
Sink		To disposes of agents (indicate the endpoint in a process model)
Delay		To delay agents for a given amount of time
Queue		To define a queue (a buffer) of agents waiting to be accepted by the next object(s) in the process flow
Select output		To route the incoming agents to one of the two output ports depending on (probabilistic or deterministic) condition
Seize		To seize a given number of resource units from a given ResourcePool
Release		To release a given number of resource units previously seized by Seize block
Batch		To convert a number of agents into one agent (batch)
Unbatch		To extract all agents contained in the incoming agent (batch) and output them through the output port
Pick up		To remove agents from a given Queue and add them to the contents of the incoming agent (″container″)
Drop off		To remove the agents contained in the incoming ″container″ agent and output them via outDropoffport
Enter		To insert the (already existing) agents into a particular point of the process model
Move to		To move the agent to a new location

the time taken for a single execution of continuous integration, etc.) are determined, the randomness will only have a tolerable impact on research results;

- When the probability of different result types of CI results (*failed* and *passed*) is determined, the change of the result type sequence will only have a tolerable impact on research results;
- When the probability of different commit types of commits (*push* and *pull request*) is determined, the change of the commit type sequence will only have a tolerable impact on research results;
- When the performance of the predictor in the project is determined, that is, when the confusion matrix is determined, the change of the prediction result sequence will only have a tolerable impact on the research results.

In this paper, the defective commit only refers to the commit that can cause the execution result of CI to be *failed*. The RIDEC method proposed in this paper is based on the following assumption: if executing CI on different defective commits in the same batch, we would get inconsistent CI reports (also known as build logs), which is reflected in details that the specific *failed* test cases or reported errors are different.

E. Construction of CISimulator

We followed three steps to build the simulation model, including analysis of the CI process, construction of the descriptive model, and construction of the simulation model. We have explained the CI process and predictive CI in Sec. II-A. Based on the analysis of the official documents of Travis CI and the data in TravisTorrent, we first built a descriptive model. We then used the DES paradigm to realize the dynamics of the descriptive model, that is, to construct a simulation model. The evolution from the descriptive model to the dynamic (simulation) model is shown in Fig. 5. The static process, which is a generalized flow-chart of CI process with predictive CI integrated, contains thirteen Steps (i.e. from **S1** to **S13**). Fig. 1 depicts an example of this process. The order from Step 1 (**S1**) to Step 13 (**S13**) is not strictly followed because there are conditional branches in the process.

We used the simulation tool *Anylogic (version 8.7.6)* to implement the computerized (simulation) model. The *process modeling library* were used and the detailed description of them can be referred to the official support document[7]. The main computerized components used in the model are shown in Table II. The model source file is available on GitHub [23].

In short, Fig. 5 shows the descriptive model with 13 Steps (labeled as **S1** to **S13** in figures). For each step, a simulation model snippet discloses its inside implementation in CISimulator, namely the computerized components corresponding to the descriptive components. All these computerized components are connected to form the simulation model. Therefore, Fig. 5 depicts the descriptive model, the simulation model, and the mapping between them. The defect fixing stage was not included in our simulation model for the following two reasons: 1) With RIDEC, defect fixing is no longer affected by using CI predictions; 2) Defect fixing is an offline human intellectual activity, lacking relevant logging and difficult to measure accurately, therefore the inclusion of defect fixing may introduce bias in evaluating predictive CI. The simulation model simulates the event flow of the commit from being submitted (created) to being merged into the master branch (ended). The detailed descriptions of CISimulator are as follows:

[S1: Create Commits] As the model starts running, commits will be created by the NewCommit at random intervals.

[7]https://anylogic.help/library-reference-guides/process-modeling-library/index.html

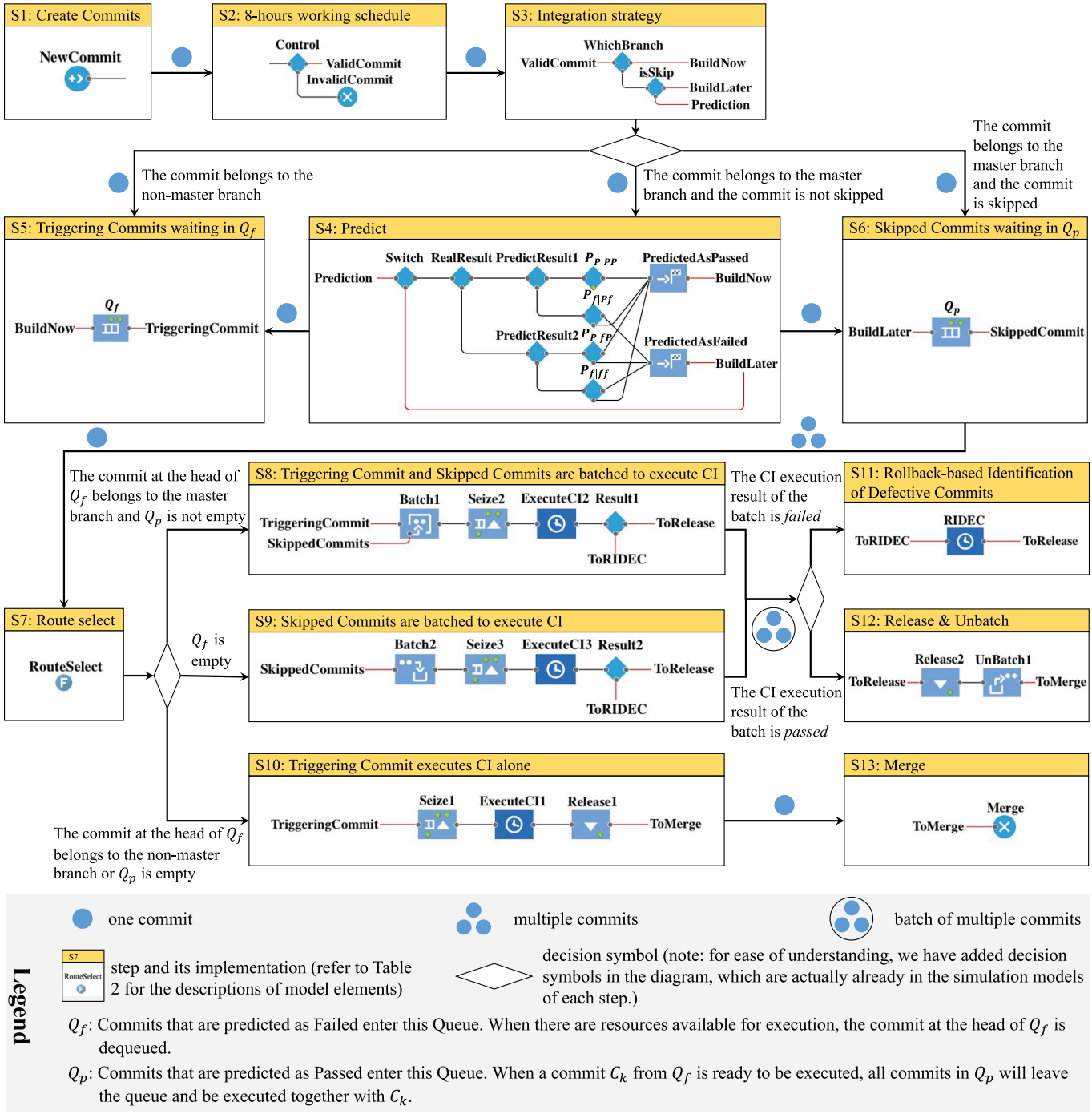


Fig. 5. Mapping of the descriptive model and the simulation model.

When creating a commit, the commit will be set with various properties, including the commit type (pull request or push), a branch of commit (master or non-master), whether to skip CI, the execution time and result of CI. Among these properties, the commit type and branch of the commit are determined when the developer submits the commit, so these two attributes can be set when the commit is created. In addition, a commit can skip the execution of CI by adding some keyword in the commit message, such as "skip ci", "skip travis", etc. Therefore, whether to skip the execution of CI can be determined when the commit is created. If the small probability event (e.g., flaky test) is not considered, and the execution result of CI is only related to whether the current commit contains defects, the execution result of CI triggered by the commit is also determined when it is created. In short, it is practical to set the above properties for the commit when it is created. In addition, we found that when the execution result of the current CI is determined, the execution result of the next CI has different probabilities. In order to fit the real situation, we use two conditional probabilities to assign a result type (*passed* or *failed*) to the commit. The details are explained in Table IV.

TABLE III
DESCRIPTIONS OF MEASUREMENT VARIABLES OF PREDICTORS USED IN THE SIMULATION

Var.	description
$P_{f\|ff}$	$P(A=f\|B=f,C=f)$, which means the probability that the prediction result of current commit (A) is *failed* on the premise that the real result of current commit (B) is *failed* and the prediction result of previous commit (C) is *failed*. $P_{p\|ff}=1-P_{f\|ff}$.
$P_{f\|pf}$	$P(A=f\|B=p,C=f)$, which means the probability that the prediction result of the current commit is *failed* on the premise that the real result of current commit is *passed* and the prediction result of previous commit is *failed*. $P_{p\|pf}=1-P_{f\|pf}$.
$P_{p\|fp}$	$P(A=p\|B=f,C=p)$, which means the probability that the prediction result of the current commit is *passed* on the premise that the real result of the current commit is *failed* and the prediction result of previous commit is *passed*. $P_{f\|fp}=1-P_{p\|fp}$.
$P_{p\|pp}$	$P(A=p\|B=p,C=p)$, which means the probability that the prediction result of current commit is *passed* on the premise that the real result of current commit is *passed* and the prediction result of previous commit is *passed*. $P_{f\|pp}=1-P_{p\|pp}$.

[S2: 8-hours working schedule] In most enterprises, developers typically work only eight hours a day. Therefore, we simulated the "8-hours working schedule" in the simulation. We assume that new code commits are created only during developers' working hours. To implement this, we add a round-time event, which will be triggered every 8 hours. When the event is triggered, it will determine whether the current time belongs to the working time. If it does, the current commit is valid and will enter **S3**; Otherwise, the current commit is invalid and will be discarded.

[S3: Integration strategy] We specify different integration strategies for the commits on master or non-master branches. For the commits on the master branch, there are two integration strategies: predictive CI and random skipping. We use IsSkip to configure which strategy to apply. It should be noted that random skipping is used as a baseline to study the effectiveness of predictive CI. For the commits on the non-master branches, since the proportion of the commits on the non-master branches is small (less than 10%) and the commit interval is long, these commits will directly trigger the CI instead of using predictive CI. Commits from the master branch with predictive CI will enter **S4**; commits from the master branch with random skipping will enter **S6**; commits from non-master branches will enter **S5**.

[S4: Predict] It is reported in [50] that the prediction result of the current CI has a high correlation with the execution result of the previous CI. Therefore, we also use conditional probability to model the predictor. We use four variables ($P_{f|ff}$, $P_{p|fp}$, $P_{f|pf}$, $P_{p|pp}$) to describe a predictor. The description of each variable is shown in Table III. These variables are in the form of $P(A|B,C)$, where A denotes the prediction result of the current commit, B denotes the real result of the current commit, and C denotes the prediction result of the precedent commit. $P(A|B,C)$ is essentially a conditional probability, which means the probability of the occurrence of A on the premise that B and C are satisfied. In our research, A, B, C have only two results: *passed* and *failed*, which are represented by *p* and *f* respectively. Algorithm 2 shows the execution process of the predictor based on these variables. It assigns a prediction result based on the real result of the current commit (i.e. its result type) and the prediction result of the previous commit, with a certain probability. For example, if the real result type of the current commit is *passed* and the prediction result of the previous commit is *failed*, the probability that the prediction result of the current commit is *failed* equals to the calibration variable $P_{f|pf}$. Commits predicted as *failed* will enter **S5**, and commits predicted as passed will enter **S6**.

Algorithm 2 Calculation of the prediction results of the current commit.

Input: $P_{f|ff}$, $P_{f|pf}$, $P_{p|fp}$, $P_{p|pp}$; $R_{current}$: real result of current commit; $P_{previous}$: prediction result of previous commit;
Output: $P_{current}$: prediction result of current commit;

```
Generate a random float number N ∈ [0, 1)
if R_current == passed then
    if P_previous == passed then
        if N < P_p|pp then
            P_current ← passed
        else
            P_current ← failed
        end if
    else
        if N < P_f|pf then
            P_current ← failed
        else
            P_current ← passed
        end if
    end if
else
    if P_previous == passed then
        if N < P_p|fp then
            P_current ← passed
        else
            P_current ← failed
        end if
    else
        if N < P_f|ff then
            P_current ← failed
        else
            P_current ← passed
        end if
    end if
end if
return P_current;
```

[S5: Triggering Commits waiting in Q_f] After using predictive CI, commits can be divided into two categories: triggering commit and skipped commit. Triggering commit, which is predicted as *failed*, will trigger CI immediately after obtaining server resources. If server resources are not available, the triggering commit will enter Q_f and wait for server resources. Skipped commit, which is predicted as passed, needs to wait in Q_p until a triggering commit appears. Then, all the skipped commits in the Q_p will be packaged with the triggering commit (at the head of the Q_f) as a batch to execute CI together.

[S6: Skipped Commits waiting in Q_p] Skipped commit will enter Q_p and wait for triggering commit.

[S7: Route select] When the server resources are available, there are three situations: 1) If the triggering commit at the head

TABLE IV
THE CALIBRATION AND INPUT VARIABLES OF EACH PROJECT

#	Variables	Type	Descriptions
01	InputCommit	Cal.	The mean of the time interval for generating a new commit. The variable follows the Gamma distribution, and the unit is minute.
02	CommitType	Cal.	The probability that the commit type is *pull request*.
03	ResultType	Cal.	The CI execution result (*failed* or *passed*) of the current commit. It consists of two conditional probabilities, denoted as ($P(A = p\|B = p)$ and $P(A = f\|B = f)$). Where $P(A = p\|B = p)$ means the probability that the CI execution result current commit (A) is *passed* on the premise that the CI execution result of previous commit (B) is *passed*, $P(A = f\|B = f)$ means the probability that the CI execution result of current commit (A) is *failed* on the premise that the CI execution result of previous commit (B) is *failed*.
04	PRPassedDelay	Cal.	The time cost of a *pull request* commit to executing CI and the execution result of CI is *passed*. The variable follows the Gaussian distribution, and the unit is minute.
05	PRFailedDelay	Cal.	The time cost of a *pull request* commit to executing CI and the execution result of CI is *failed*. The variable follows the Gaussian distribution, and the unit is minute.
06	PushPassedDelay	Cal.	The time cost of a *push* commit to executing CI and the execution result of CI is *passed*. The variable follows the Gaussian distribution, and the unit is minute.
07	PushFailedDelay	Cal.	The time cost of a *push* commit to executing CI and the execution result of CI is *failed*. The variable follows the Gaussian distribution, and the unit is minute.
08	Master	Cal.	The probability that a commit belongs to the master branch.
09	Skip	Inp.	The probability that a commit skips the CI execution.
10	Predictor	Inp.	Quadruple is used to describe the predictor, e.g., ($P_{f\|ff}$, $P_{p\|fp}$, $P_{f\|pf}$, $P_{p\|pp}$), refer to Table III.
11	ServerResources	Inp.	The total number of server resources.

of Q_f belongs to the master branch and Q_p is not empty, all the skipped commits in Q_p will be packaged with the head triggering commit as a batch to execute CI together, that is, enter *S8*. 2) If the server is released and there is no triggering commit (Q_f is empty), in order to make full use of the server resources, the skipped commits in Q_p will be packaged as a batch to execute CI together, that is, enter *S9*. 3) If the head triggering commit belongs to the non-master branches or Q_p is empty, the head triggering commit will execute CI alone, that is, enter *S10*.

[S8: Triggering Commit and Skipped Commits are batched to execute CI] After execution, the batch will enter different processes according to the CI execution result. If the execution result of CI is *passed*, the batch will enter **S12** to release server resources and be unpacked. Otherwise, the batch will proceed to **S11** to locate the defective commits in the batch using RIDEC, and the server resources will not be released until RIDEC is complete.

[S9: Skipped Commits are batched to execute CI] Similar to **S8**, the batch will enter different processes according to the CI execution result.

[S10: Triggering Commit executes CI alone] Different from **S8** and **S9**, the server does not need to locate defective commits with RIDEC, because the CI execution result indicates whether the current commit is defective. After execution, the server resources will be released and the commit will enter **S13** to merge into the master branch.

[S11: Rollback-based Identification of Defective Commits] The RIDEC method automatically locates defective commits through the server. The details were explained in Sec. III-C. After finding the defective commits, C_{batch} will enter **S12** to release server resources and be unpacked.

[S12: Release & Unbatch] After CI execution, the server resources will be released and the current batch will be unpacked.

[S13: Merge] After CI execution, all commits will be merged into the master branch and exit the process.

F. Calibration of CISimulator

The simulation uses probabilistic randomness to simulate the real situation through the distribution presented by the historical data. Each variable in the simulation model needs to be calibrated according to the actual project data, so as to reflect the real situation. We calibrated CISimulator respectively with the data from each project. The calibration was based on data obtained from build logs on TravisTorrent and commit logs on GitHub, which can be associated by "commit_id". The calibration variables and input variables of the model are shown in Table IV. We conducted Kolmogorov-Smirnov test [51] for testing whether the data conforms to one of the twelve types of distributions supported by *Anylogic*, including Gaussian, Uniform, and Exponential, etc. Unfortunately, some of the real data set does not follow any of these distributions. Therefore, we selected the closest standard distribution based on the Kolmogorov-Smirnov Distance for each variable. Table V presents the results of the Kolmogorov-Smirnov test. A smaller distance means that the data are more similar to the hypothesized distribution, e.g., the distribution of InputCommit is closer to a Gamma distribution than others.

For the variable InputCommit, we sorted all the commits in ascending order according to the "started_at" in the build log. Then, we calculated the creation time interval of all adjacent commits. We found that the creation time interval of adjacent commits is close to the Gamma distribution. Therefore, we use exponential distribution to set the creation time interval of adjacent commits.

For the variable CommitType, we identify the commit type of each commit according to the "event_type" field in the build log. Travis CI has four trigger types of CI, including *push*, *pull request*, *cron job*, and *api*. Since the sum proportions of *cron job* and *api* are at very small levels in our selected projects (shown in Table VI), we excluded these two CommitTypes. Hence, we use the proportion of *pull request* as the probability that a newly generated commit is set as *pull request* in CISimulator, otherwise it is a *push*.

For the variable ResultType, we identify the execution result of each CI according to the "tr_status" field in the build log. Then, we set the ResultType for commit according to the conditional probability introduced in Sec. III-E.

For the variables PRPassedDelay, PRFailedDelay, PushPassedDelay, PushFailedDelay, we extracted the start time and end time of each CI through "started_at" and "finished_at"

TABLE V
THE MEAN OF KOLMOGOROV-SMIRNOV DISTANCE BETWEEN THE REAL DATA DISTRIBUTIONS AND THE HYPOTHESIZED DISTRIBUTIONS ON THE SELECTED PROJECTS

	InputCommit	**PRPassedDelay**	**PRFailedDelay**	**PushPassedDelay**	**PushFailedDelay**
Gaussian	0.22	0.24	0.23	0.18	0.29
Uniform	0.54	0.44	0.47	0.36	0.47
Triangular	0.46	0.40	0.41	0.34	0.48
Exponential	0.20	0.49	0.41	0.51	0.42
Gamma	0.10	0.81	0.63	0.95	0.81
Beta	0.96	1.00	1.00	1.00	1.00
Chi2	1.00	1.00	1.00	1.00	0.78
Erlang	0.99	1.00	1.00	0.83	0.83
Logistic	0.40	0.36	0.35	0.35	0.38
Pareto	0.45	0.85	0.74	0.92	0.78
Pert	0.63	0.59	0.56	0.59	0.58
Rayleigh	0.45	0.39	0.34	0.39	0.48

We highlight the selected distribution for each variable with a green background.

TABLE VI
DATA CHARACTERISTICS OF THE SIX PROJECTS

proj.	CI Result			Trigger Type			
	Total	Passed	Failed	Push	PR	Cron	Api
A	23402	19511	676	0	22911	489	2
B	8650	7234	1076	3765	4885	0	0
C	6431	5936	427	1215	5214	0	2
D	5361	3719	1565	783	4138	440	0
E	4225	3518	658	993	3232	0	0
F	4010	1775	1970	1331	2679	0	0

* A: python/cpython; B: pypa@warehouse; C: apache/hive; D: pypa/pip; E: akka/akka; F: opf/openproject.

fields in the build log. Then we use the difference between these two variables as the execution time of one CI. There are obvious differences in the execution time of one CI for different ResultTypes and CommitTypes of commits. Hence, we measured the distribution of CI execution time for four types of commit, including *passed push*, *failed push*, *passed pull request*, and *failed pull request*. The most similar distribution is the Gaussian distribution.

For the variable Master, existing studies [14], [15], [17] do not distinguish the branch of commit when training and validating the model. However, we observed that the performance of the predictor on different branches is different. Hence, it is reasonable to establish prediction models for different branches. The commit branch can be identified by the "branch" in the build log. Since the proportion of the commits on the non-master branches is small (less than 10%), we divided commits into two categories: commits on the master branch, and commits on non-master branches (non-master). Hence, we use the proportion of master as the probability that a newly generated commit belongs to the master branch in CISimulator.

IV. VERIFICATION & VALIDATION OF CISIMULATOR

The purpose of our establishment of the CISimulator is to provide a simulation environment for the evaluation of the application effects of predictors in continuous integration. The verification and validation of the CISimulator itself is crucial as it is the premise of implementing effective simulation experiments.

Gong et al. [52] summarized the Verification and Validation (VV) frameworks for software process simulation modeling [44], [45], [46] and conducted a mapping study to investigate the V&V methods used in existing studies. They identified five quality aspects including syntactic quality, semantic quality, pragmatic quality, performance, and value. The goal of value is to validate the practical utility of the model, which is beyond the scope of this research. Therefore, we verify and validate CISimulator from the first four quality aspects.

A. Syntactic Quality

The goal of syntactic quality is syntactic correctness. To achieve this, two researchers independently checked the parameters of each block and checked the links between blocks to ensure the model meet the syntax defined in *AnyLogic*. No errors or warnings were reported during the running of the simulation.

Qualitative evaluation results: The syntax of CISimulator conforms to the requirements of *AnyLogic 8.7.6*.

B. Semantic Quality

Semantic quality contains two goals, i.e. feasible validity and feasible completeness. To achieve the goals, we verified and validated the model from three aspects, including model structure, variables, and simulation results. We established the mapping shown in Fig. 5 to check the structural consistency between the simulation model and the descriptive model. To verify variables, we first performed a dimensional consistency test. In this model, the only unit of measurement is time. We manually checked the time dimension of all blocks in minutes. Then we manually checked the distribution functions of all input variables. After the verification of the structure and variables of the model, we ran the model with variables calibrated from projects shown in Table VI. We performed extreme condition tests as well as used the slow animation function built-in *AnyLogic* to observe the flow of commits. For each branch in the model, we set 0%, 50%, and 100% probabilities respectively. We ran 10 trials for each setting. In running, we validated that the flow of commits is in line with expectations. After executing the simulation, we validated that the outputs are in line with expectations. By the

extreme condition tests, we determined that the proportion of commits in the model entering different workflows conforms to the set probabilities, that is, the operation of the model conforms to the expected design.

Qualitative evaluation results: The structure of the CISimulator is consistent with the descriptive model. The units of each variable are consistent. The extreme condition tests show that CISimulator conforms to the context of the CI processes (including with and without the predictive CI method).

C. Pragmatic Quality

Pragmatic quality consists of two goals, i.e. understandability and comprehension. In other words, the model needs to be easy to understand and easy to use. The common methods for validating pragmatic quality are survey and face validity (present the model to users or experts). In another work in our lab [53], a third-year academic master student planned to build a CI process simulation model that integrates security vulnerability predictors based on CISimulator. This master student had never participated in this research and had not learned software process simulation before. He first spent 3 days learning how to use *AnyLogic* to build a DES model. After a half-hour discussion with him, he fully understood CISimulator and was able to use it for simulation experiments. Two weeks later, he completed the simulation model for another work (security-oriented commits priority in CI) alone based on CISimulator. This case illustrates the understandability and the usability of CISimulator.

Qualitative evaluation results: CISimulator shows good quality in terms of understandability and comprehension. In addition, it exhibits good scalability and multi-scenario applicability.

D. Performance

The goal of performance aspects is to validate the fidelity of the DES model. There are a variety of methods that can be used to validate performance, and one of the most effective is to conduct experimental case studies. Since there are no open source projects applying continuous integration result prediction in reality, we compared ours with the quasi-simulation method [24]. As we discussed in Sec. II-B, although quasi-simulation has obvious limitations, it can reflect reality under limited conditions, that is, it can be regarded as approximate reality. Therefore, we compare the results of our simulation and quasi-simulation under limited conditions. Although this is an incomplete validation, it is sufficient to validate the correctness and fidelity of our simulation model.

Projects: We used Travis CI API to crawl the CI history data of 1738 open-source projects on GitHub that use Travis CI for continuous integration. Considering the timeliness of the data, each project only retains the CI data of 2020 and 2021 for experiments. In order to ensure the amount of data and reflect the differences between experimental projects, we sorted all projects according to the number of CI executions, and selected 6 projects based on two rules: 1) projects with more than 4,000 CI executions; 2) among the retained projects, we selected 6 representative projects according to the number of CI executions, of which two projects involve the highest number of executions, two with the median number, and two with the least number. Table VI shows the details of them. There are clear differences among these projects, which can well validate the universality of the simulation model.

Predictors: Based on new data, we repeated the experiments in study [18]. We evaluated the performance of 67 predictors using a time series validation and selected five representative predictors. The performance of these predictors is generally in five positions in the distribution of performance (indicated by F_{2f}) of all predictors, corresponding to the upper limit (100%), 75%, 50%, 25%, and the lower limit (0%) respectively.

The five predictors are the combination of Instance Hardness Threshold [54] sampling algorithm and Random Forest [55] (IHT+RF), Balanced Bagging (BB) [56] without sampling, Cost Sensitive Pasting (CSP) [57] without sampling, the combination of Neighbourhood Cleaning Rule [58] sampling algorithm and Decision Tree [59] (NCR+DT), LocalOutlierFactor (LOF) [60] without sampling. It should be noted that in this step, we only sort and select the predictors, and do not directly use the obtained evaluation results as the input of the simulation.

Experimental settings: Due to a series of problems such as cross-time zone collaboration and the unknown number of servers, we limit the comparative experiments to scenarios where quasi-simulation can be performed. Therefore, we made the following three settings for the experiment: 1) The 8-hour work schedule is not considered, that is, **S1** will continuously create commits; 2) the server resources are unlimited, that is, there will be no waiting phenomenon in **S5** and **S6**; 3) the skipped commits can only execute CI together with the triggering commit, even if the server resources are available, that is, **S10** will not be selected.

Quasi-simulation: Quasi-simulation essentially performs and records the prediction process of each commit in the test set in a more fine-grained way, and finally reinterprets the prediction results. The quasi-simulation input history commits in the order in which they were actually built. The details were discussed in Sec. II-B, and please refer to the study [24] for more details. We divide the data set into a training set and test set according to 8:2.

Simulation: The simulation used the same training and test sets as the quasi-simulation. The difference is that instead of directly using them as simulation inputs, we use them as data sources for input variables and calibration variables. We computed calibration variables for the simulation model on the test set. The calibration variables and input variables for each project are shown in Table VII and Table VIII respectively. For each scenario, we performed 100 runs to mitigate accidental errors.

Validation metrics: We used eight metrics to validate the correctness and fidelity of the simulation model, including the number of True *Failed* commits produced by the predictor (# TF), the number of False *Failed* commits produced by the predictor (# FF), the number of True *Passed* commits produced by the predictor (# TP), the number of False *Passed* commits produced by the predictor (# FP), Time cost for executing CI without RIDEC on the Master branch (T_{ci_m}), Time cost

TABLE VII
CALIBRATION VARIABLES FOR EACH PROJECT

#	Variables	Projects					
		A	B	C	D	E	F
01	InputCommit	19.88	18.88	47.12	41.08	43.18	33.19
02	CommitType	1	0.9	0.87	0.85	0.77	0.65
03	ResultType	(0.97,0.30)	(0.93,0.54)	(0.92,0.24)	(0.82,0.49)	(0.91,0.40)	(0.67.0.61)
04	SkipPR	0	0	0	0	0	0
05	SkipPush	0	0.02	0.19	0	0	0.37
06	PRPassedDelay	(13.95, 3.46)	(8.90, 5.03)	(13.43, 1.40)	(36.69, 8.76)	(18.89, 3.53)	(38.45, 16.08)
07	PRFailedDelay	(5.03, 0.71)	(12.10, 7.95)	(8.18, 4.18)	(15.47, 15.74)	(12.11, 3.62)	(33.33, 12.57)
08	PushPassedDelay	(0,0)	(10.74, 7.12)	(12.66, 1.58)	(40.38, 4.45)	(23.84, 3.74)	(42.00, 19.45)
09	PushFailedDelay	(0,0)	(12.66, 10.30)	(9.65, 11.50)	(30.63, 1.79)	(16.28, 13.44)	(38.35, 11.96)
10	Master	0.88	0.98	0.95	0.97	0.93	0.72
11	ServerResources	1	1	1	1	1	1

TABLE VIII
MEAN PERFORMANCE OF PREDICTORS ON EACH PROJECT

Predictor	Projects					
	A	B	C	D	E	F
IHT+RF	(0.91, 0.72, 0.58, 0.96)	(0.96, 0.24, 0.76, 0.87)	(0.89, 0.65, 0.74, 0.71)	(0.87, 0.45, 0.62, 0.81)	(0.88, 0.34, 0.66, 0.64)	(0.85, 0.66, 0.42, 0.88)
BB	(0.75, 0.64, 0.37, 0.87)	(0.84, 0.43, 0.63, 0.88)	(0.63, 0.55, 0.55, 0.68)	(0.69, 0.51, 0.42, 0.70)	(0.72, 0.45, 0.53, 0.69)	(0.81, 0.42, 0.59, 0.48)
CSP	(0.62, 0.89, 0.14, 0.99)	(0.89, 0.50, 0.36, 0.92)	(0.00, 1.00, 0.00, 1.00)	(0.81, 0.58, 0.43, 0.82)	(0.58, 0.92, 0.17, 0.98)	(1.00, 0.00, 0.99, 0.00)
NCR+DT	(0.38, 0.83, 0.27, 0.95)	(0.68, 0.52, 0.40, 0.87)	(0.33, 0.89, 0.35, 0.89)	(0.65, 0.44, 0.39, 0.73)	(0.51, 0.66, 0.43, 0.81)	(0.62, 0.72, 0.20, 0.81)
LOF	(0.00, 0.94, 0.21, 0.98)	(0.09, 0.92, 0.56, 0.95)	(0.47, 0.91, 0.30, 0.95)	(0.64, 0.97, 0.42, 0.98)	(0.33, 0.92, 0.35, 0.93)	(0.20, 0.95, 0.22, 0.93)

* Quadruple are used to describe the predictor, e.g., ($P_{f|ff}$, $P_{p|fp}$, $P_{f|pf}$, $P_{p|pp}$).

for executing CI without RIDEC on the Other branches (non-master) (T_{ci_o}), Time cost for locating defective commits using RIDEC (T_{fl}), the Number of commits to Wait from creation to execution of CI (N_{w_ci}). We measure the relative Error (E_r) between simulation results (R_s) and quasi-simulation results (regarded as approximate actual values, R_a) for each variable as follows:

$$E_r = (R_s - R_a)/R_a \tag{1}$$

Quantitative evaluation results: The validation results are shown in Table IX. For each predictor, we present the mean value of E_r of all the simulation runs for each variable. The results show that the relative errors of # TF, # FF, # TP, and # FP are at a low level (less than 10% on average). Although the mean relative errors of T_{ci_m},T_{ci_o}, T_{fl}, and N_{w_ci} are more than 10%, the approximate actual values are within the distribution range of 100 runs of simulation results. Quasi-simulation is an approximation of actual values obtained directly using real data, which reflects what would have happened if the prediction were used in the past depicted by the historical data. However, simulation is based on the statistical distribution of real data, which reflects what the prediction would be like if the future data distribution is consistent with the historical data. The result of the simulation is represented by a probability distribution, whilst the result of the quasi-simulation is represented by a specific value (a definite outcome of all possible outcomes that follow the distribution). If the result of quasi-simulation is within the distribution obtained by the simulation, we can infer that the simulation can cover the real situation. Hence, the validation results indicate that CISimulator can accurately simulate the generation of commits and the prediction performance of predictors.

V. SIMULATION EXPERIMENTS

We conduct simulation experiments using six popular OSS projects in Table VI as empirical cases to study the effectiveness of predictive CI using simulation. At this stage, we consider effectiveness from two aspects: 1) the relative cost savings of the time cost for executing CI, that is, the sum of the time cost for executing CI triggered by the commit on the master branch; 2) the relative cost savings of the average waiting time before executing CI, that is, the ratio between the sum of the waiting time before executing CI and the total number of commit.

A. Research Questions

It is difficult to know the effect of using machine learning-based predictors without evaluating it from the perspective of the software process. The need to use simulation rather than common prediction evaluation metrics stems from the complexity of the software development process. The process may change due to the use of predictions, especially considering that the order of commits for executing CI can be optimized and that the gains of successful predictions and the losses of incorrect predictions are not equal. To evaluate machine learning-based CI predictors, we simulate a new CI process that includes predictive CI. To investigate why and when CI prediction should be used, and what type of predictors should be selected from the process perspective, we derive the following Research Questions (RQs).

TABLE IX
RELATIVE ERROR BETWEEN SIMULATION RESULTS AND ACTUAL VALUE

Proj.	Predictors	E_r for Each Variable							
		# TF	# FF	# TP	# FP	T_{ci_m}	T_{ci_o}	T_{fl}	N_{w_ci}
A	IHT+RF	1.26%	-28.57%	-9.23%	-3.58%	67.48%	3.46%	118.96%	-34.66%
	BB	3.66%	-7.59%	-14.82%	-12.19%	34.27%	3.35%	65.00%	-15.49%
	CSP	1.01%	-17.84%	-7.45%	-6.13%	15.87%	2.22%	15.52%	52.41%
	NCR+DT	2.44%	-13.80%	-18.03%	-20.00%	10.93%	2.84%	12.48%	0.27%
	LOF	0.39%	-17.15%	0.22%	14.25%	13.56%	3.51%	12.07%	-8.01%
B	IHT+RF	-9.56%	10.79%	-4.60%	4.31%	29.18%	-23.49%	108.56%	-52.87%
	BB	-6.96%	-10.71%	1.57%	2.09%	45.19%	-20.16%	105.98%	-19.28%
	CSP	-6.18%	9.05%	-9.40%	11.44%	75.81%	-21.61%	136.27%	-40.24%
	NCR+DT	-5.01%	-3.72%	-3.07%	-1.51%	76.70%	-25.61%	134.37%	-1.30%
	LOF	-6.87%	-12.17%	22.22%	16.67%	-29.70%	-19.43%	-35.50%	-57.45%
C	IHT+RF	-7.34%	9.23%	10.63%	5.24%	14.03%	17.45%	76.45%	-52.26%
	BB	-3.84%	4.89%	13.98%	3.79%	8.47%	18.60%	6.71%	-25.36%
	CSP	-4.73%	15.13%	0.00%	0.00%	0.00%	23.87%	0.00%	0.00%
	NCR+DT	-2.00%	21.56%	-0.20%	0.49%	18.87%	21.23%	22.57%	-26.57%
	LOF	-1.64%	3.91%	16.70%	4.36%	-1.11%	37.09%	-8.62%	-18.58%
D	IHT+RF	-3.54%	5.58%	-1.95%	1.40%	27.19%	9.97%	57.91%	-36.28%
	BB	-2.15%	3.38%	-1.43%	-1.63%	16.47%	10.40%	17.03%	-12.02%
	CSP	-4.98%	7.03%	-6.14%	9.57%	51.15%	10.83%	75.77%	-27.31%
	NCR+DT	-1.50%	3.26%	0.50%	-4.19%	23.13%	12.69%	32.78%	-4.28%
	LOF	-1.70%	-1.93%	10.87%	8.99%	8.18%	12.65%	6.63%	2.67%
E	IHT+RF	-3.30%	9.79%	9.04%	0.94%	12.36%	0.20%	23.98%	-24.46%
	BB	-0.25%	7.83%	-1.77%	0.09%	14.35%	1.99%	19.58%	-16.08%
	CSP	-2.14%	10.80%	12.50%	16.67%	-5.53%	6.73%	-6.33%	-25.59%
	NCR+DT	0.53%	4.83%	12.21%	-5.89%	28.60%	3.52%	44.13%	37.36%
	LOF	-1.81%	6.35%	21.06%	3.11%	-4.60%	5.37%	-8.50%	-35.91%
F	IHT+RF	-8.94%	4.79%	-24.37%	-19.65%	2.87%	-16.43%	28.53%	-35.30%
	BB	2.91%	-11.87%	-25.85%	-9.29%	1.76%	-18.42%	50.65%	1.45%
	CSP	0.00%	0.00%	-23.63%	-6.85%	-7.34%	-18.66%	0.00%	0.00%
	NCR+DT	-5.03%	-25.42%	-19.10%	2.62%	16.06%	-18.85%	18.40%	-9.12%
	LOF	-5.91%	-23.01%	6.61%	-4.60%	-13.68%	-17.01%	-16.50%	-20.97%

* A: python/cpython; B: pypa@warehouse; C: apache/hive; D: pypa/pip; E: akka/akka; F: opf/openproject.

RQ1: Why should predictive CI be used?

Saving time is the main purpose and motivation of using predictive CI. Therefore, RQ1 aims to evaluate the performance of existing predictors in terms of their ability to save time in executing CI and the average waiting time before executing CI. Hence, RQ1 specifically includes the following two sub-questions:

RQ1.1: How much time can the existing predictors save for executing CI?

RQ1.2: How much time can the existing predictors save for waiting before executing CI?

We do not distinguish information such as the number and configuration of servers in different case projects but regard the processing power provided by all servers currently in the case project as a unit of the server. Therefore, the time cost of executing CI is equivalent to the overhead of server resources.

RQ2: When should predictive CI be used?

Not all projects are suitable for adopting predictive CI, and the effectiveness of using predictive CI can also vary greatly at different stages of a project. RQ2 aims to investigate the relationship between project features and the amount of time that can be saved through predictive CI. Specifically, we analyze the relationship between three simulated project features in CISimulator and time-saving. These features include the average time interval between adjacent commits (referred to as commit interval), the failure rate of CI executions (referred to as failure rate), and average execution duration of CI (referred to as execution duration). Hence, RQ2 specifically includes the following two sub-questions:

RQ2.1: What are the impacts of different project features on the time cost for executing CI?

RQ2.2: What are the impacts of different project features on the time cost for waiting before executing CI?

RQ3: What type of predictors should be used?

In the case that predictive CI is suitable to a project, it is important to choose a good predictor for maximizing the returns. RQ3 aims to investigate the relationship between common machine learning performance metrics and time saving, that is, to analyze what kind of performance we should prioritize when choosing a predictor. Hence, RQ3 specifically includes the following two sub-questions:

RQ3.1: What are the impacts of different prediction performance on the time cost for executing CI?

RQ3.2: What are the impacts of different prediction performance on the time cost for waiting before executing CI?

B. Evaluation Metrics

We measure the time cost for the server to execute CI on the master branch with RIDEC (T_{ci_m}), and the average waiting time before executing CI (T_{wait}) respectively.

T_{ci_m} represents the total server resource consumption, that is, the sum of the time consumed by all commits on the master branch to execute CI, and the time consumed for the server to locate defective commits based on RIDEC. We denote the T_{ci_m} of the predictor M as $T_{ci\_m(M)}$.

T_{wait} represents the average waiting time before executing CI, that is, the ratio between the sum of the waiting time before executing CI and the total number of commits during the simulation. Due to limited server resources, if developers frequently commit to the code warehouse, these commits need to wait for server resources in the queue according to FIFO. That is, the following commit can only be integrated after the previous commit completes the CI and releases the server resources.

We evaluate the effectiveness of predictors from two aspects, namely the time cost of executing CI and the average waiting time before executing CI. We define $S_{ci\_m(M,N)}$ (Equation 2) and $S_{wait(M,N)}$ (Equation 3) to respectively measure the relative savings of the time cost for executing CI (with RIDEC) and relative savings of the average waiting time before executing CI between the predictor M and the predictor N.

$$S_{ci\_m(M,N)} = (T_{ci\_m(M)} - T_{ci\_m(N)})/T_{ci\_m(M)} \quad (2)$$

$$S_{wait(M,N)} = (T_{wait(M)} - T_{wait(N)})/T_{wait(M)} \quad (3)$$

C. Design of Experiments

Samples. We use the six projects presented in Table VI as the experimental cases. The projects were also used for the V&V of CISimulator as explained in Sec. IV-D. In the V&V experiments, we directly use the entire dataset of a project for training and testing of predictors, as well as the calibration of the CISimulator. While in simulation experiments, we randomly sample the data for each project to obtain a sufficient number of samples for statistical analysis. We do not collect more projects because there are significant contextual differences between them, and sampling multiple times from the same project helps control unmeasurable variables effectively. Specifically, for each project, while preserving the time sequence, we randomly slice the data 30 times. Each slice (i.e., one sample) contains 100 consecutive CI executions. We employ a sampling method with replacement, which means that there might be some overlapping data points between two samples (the probability of obtaining two identical samples is very low). We calibrated the model for each sample.

Control group. We take the scenario that does not use predictive CI and does not skip executions as the control group. We denote it as skip-0.

Treatment groups (predictors). We select five representative predictors as explained in Sec. IV-D. The five predictors are {IHT+RF, BB, CSP, NCR+DT, LOF}. For RQ1, we add an ideal predictor (accuracy equals 1) to explore the maximum effect that can be achieved using predictive CI.

Comparison groups (random skipping). For comparison, we introduce five control groups as baselines, namely random skipping 20%, 40%, 60%, 80%, and 100% of the commits respectively. We denote them as skip-20, skip-40, skip-60, skip-80, skip-100.

Number of servers: Changes in the number of servers will eventually be reflected in changes in server utilization and the wait time of commits before executing CI. To evaluate the effectiveness of prediction through a unified dimension for RQs, we set the number of servers as 1.

Simulation setup. We performed the simulation experiments for the 6 projects (30 samples for each) with 5 predictors, 5 random skipping strategies, and the control group. Hence, a total of 1980 $(30 \times 6 \times (5 + 5 + 1))$ cases were simulated. For the statistics of contingency, we executed 100 runs for each case, which is much more than the number of simulations in most related studies [61], [62], [63]. Each run will continue until the life-cycle of the 100th commit is completed, during which new commits will be continuously created, that is, typically more than 100 commits will be created, but the simulation will end before their lifetime ends. As we introduced in Sec. III-E, we have simulated the 8-hour work schedule. Therefore, commits will only be created continuously for 8 hours in a day, and no commits will be created at other times.

VI. Results and Analysis

This section presents and analyzes the simulation results to answer each research question.

A. Effectiveness of Existing Predictors (Why)

Figs. 6 and 7 respectively show the relative time savings of executing CI ($S_{ci\_m(M,O)}$), and the average waiting time before executing CI ($S_{wait(M,O)}$), where M indicates the predictor used, and O indicates that no predictor is used. In order to determine whether the five predictors are better than random skipping, we simulated five additional scenarios: skip-20, skip-40, skip-60, skip-80, skip-100, which denotes randomly skip 20%, 40%, 60%, 80%, 100% of the commits respectively. Furthermore, we simulated an ideal predictor to investigate the best effect that predictive CI can possibly achieve. For each project, we randomly sampled 30 times. Each box plot in the two figures represents the simulation results of a strategy (predictive or random skipping) on 30 samples of a project.

In our simulation, no matter what strategy is adopted, CI will be executed as long as there are idle server resources and execution demand. Fully utilizing resources in this way is more practical, and not doing so may exaggerate the effectiveness of CI predictions.

1) Effectiveness of Existing Predictors in Terms of Saving Time for Executing CI (RQ1.1): **All predictors can save time for executing CI in most cases.** As shown in Fig. 6, savings of over 30% can be achieved in some samples. On average, predictive CI can save time. However, except for project A, there are cases in every other project where using predictions instead leads to increased time costs.

The effectiveness of predictive CI varies greatly across different projects, and there is also substantial variation within different samples of the same project. Project F is an extreme example where there are cases in which

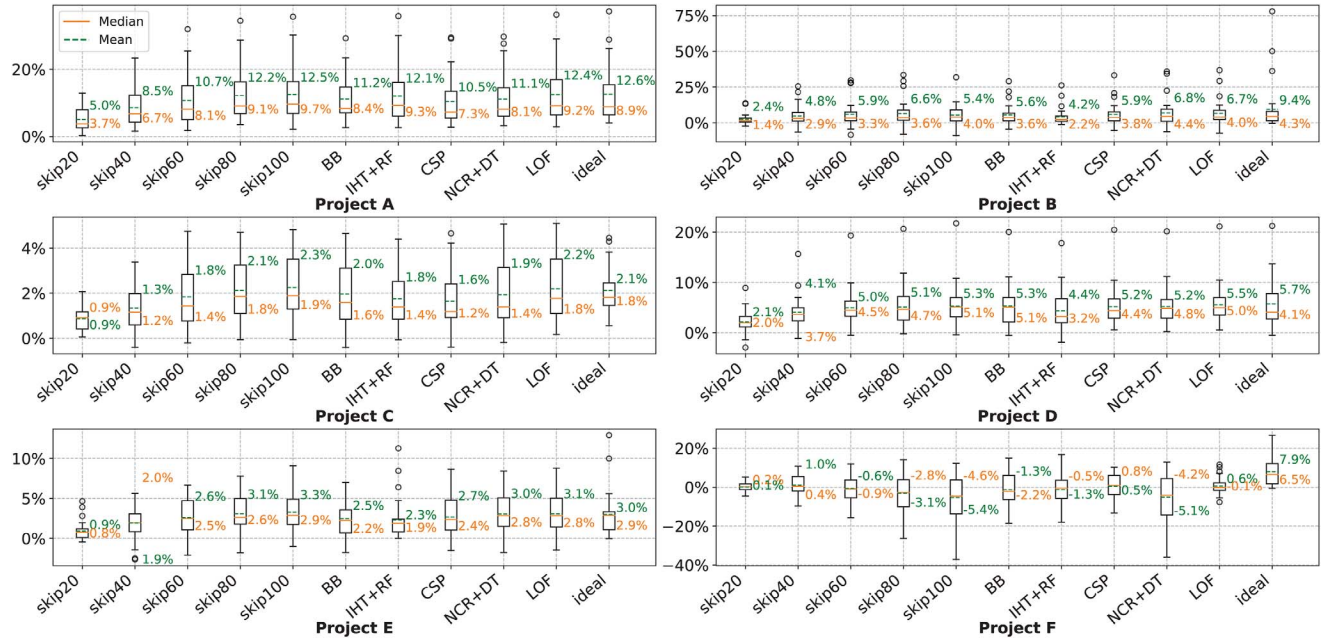


Fig. 6. Relative cost savings of the time cost for executing CI.

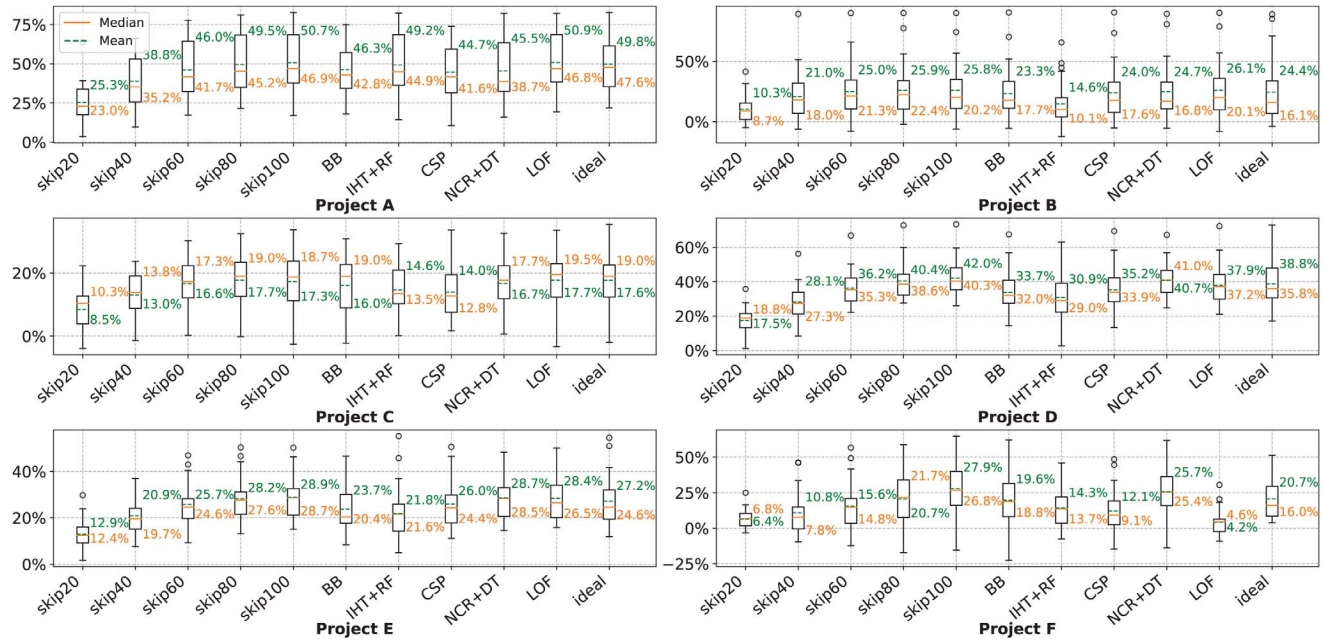


Fig. 7. Relative cost savings of the average waiting time before executing CI.

the predictive CI can save more than 10% time, as well as cases in which predictive CI causes more than 10% additional cost. The main reason for this phenomenon is that the overall failure rate of project F is high and the failure rate of different time segments (samples) has a large variation (ranging from 0.23 to 0.82).

The difference between different predictors and random skipping strategies is relatively small. The LOF predictor, which performs poorly in terms of prediction performance metrics, excels in saving time for executing CI. Skip-100 performs best on average across four projects among the random skipping strategies and it is a special case that all of the commits waiting in the queue will execute CI together once the server resources are available. LOF tends to predict most CIs as *passed*, in other words, it tends to skip the execution for the vast majority of commits. From this point of view, LOF is similar to skip-100, so it is understandable that they have similar results. However, on average, skip-100 has a negative impact on both Project B and Project F, whilst LOF can achieve savings. This is the advantage of predictors over random skipping strategies. In addition, the average results of all projects indicate that CSP and LOF perform better than all random strategies.

TABLE X
RESULTS OF PEARSON CORRELATION ANALYSIS BETWEEN THE THREE PROJECT FEATURES AND $S_{ci\_m(M,N)}$

Features	A		B		C		D		E		F	
	r	p	r	p	r	p	r	p	r	p	r	p
Commit inteval	-0.86	0.00*	-0.67	0.00*	-0.80	0.00*	-0.47	0.01	-0.47	0.01	0.25	0.19
Failure rate	-0.42	0.00*	-0.32	0.00*	0.36	0.00*	-0.16	0.01	-0.31	0.01	-0.66	0.19
Execution duration	-0.51	0.00*	0.10	0.60	-0.30	0.11	-0.42	0.02	-0.07	0.71	0.11	0.57

* Means close to but not equal to zero.

The improvement of prediction performance can indeed improve the confidence of using predictive CI. Although the difference in time-saving between existing predictors and the ideal predictor is very small, only the ideal predictor can obtain a positive effect in every sample (those values slightly less than 0 are caused by random errors). Existing predictors and random skipping strategies demonstrate insufficient effectiveness on project F and both present potential negative consequences. In contrast, an ideal predictor not only carries no risk but also offers considerable time savings.

2) Effectiveness of Existing Predictors in Terms of Saving Time for Waiting Before Executing CI (RQ1.2): **All predictors can save the average waiting time before executing CI.** As shown in Fig. 7, every predictor has achieved savings of over 30% in some samples in each of the six projects. In certain samples, savings even exceed 80%.

The effectiveness of predictive CI varies greatly across different projects, and there is also substantial variation within different samples of the same project. For the time savings for waiting before executing CI, we also see large variations in how well the predictors perform across samples. But even in project F, predictive CI can play a clearly positive role in most cases.

Random skipping strategies might be better than existing predictors. Overall, when considering all the projects, NCR+DT is the best-performing predictor with an average cost savings of 30.47%. It outperforms skip-80 (29.71%) but is weaker than skip-100 (32.44%).

An ideal predictor is not necessarily better than existing predictors in terms of $S_{wait(M,N)}$. The NCR+DT, which performs best in terms of the mean of $S_{wait(M,N)}$ over all projects, is slightly better than the ideal predictor. However, the ideal predictor is significantly better than NCR+DT in terms of $S_{ci\_m(M,O)}$. It is still meaningful to improve prediction performance with the ideal predictor as the ultimate goal.

Main findings for RQ1: existing predictors can save considerable time, but the effectiveness varies across projects. There is little difference between existing predictors and random strategies in terms of saving time in most cases. However, for projects with a high failure rate, the random strategy will have a negative impact, whilst a good predictor can still effectively save time.

B. The Impacts of Project Features (When)

An imperfect predictor works most of the time, but not always. Hence, we investigate the relationship between project features and the time savings a predictor is able to achieve. We respectively performed Pearson Correlation Analysis between the three project features (i.e. commit interval, failure rate, and execution duration) and the two metrics of relative savings (i.e. $S_{ci\_m(M,N)}$ and $S_{wait(M,N)}$). For each sample, we took the best result of the 5 predictors as the value of the $S_{ci\_m(M,N)}$ and $S_{wait(M,N)}$. The results are shown in Tables X and XI. There are noticeable differences in the degree and polarity of the correlations in different projects. The key reason for this phenomenon is that time savings are influenced by multiple variables, and there are also correlations among these variables.

1) The Impacts of Project Features on the Time Cost for Executing CI (RQ2.1): As shown in Table X, none of the three project features have a significant correlation with $S_{ci\_m(M,N)}$ in project F ($p < 0.1$). The major reason is that the performance of predictors on project F is heavily affected by the changes in project features and the predictor performs significantly worse in project F than in other projects.

The commit interval has a significant negative correlation with $S_{ci\_m(M,N)}$. A smaller commit interval means that there is a higher likelihood of queuing, and queuing is the premise for predictive CI to work because it will execute CI regardless of the predicted result when the server is idle. In project A, where the degree of correlation is highest, increasing the commit interval from about 15 minutes to 30 minutes results in a decrease in the $S_{ci\_m(M,N)}$ from about 30% to 10%.

The failure rate has a significant negative correlation with $S_{ci\_m(M,N)}$ in most cases. A lower failure rate means there is a greater likelihood of executing fewer times CIs during defect localization in RIDEC, that is, to obtain a larger $S_{ci\_m(M,N)}$. However, we observe the opposite correlation in project C, which is mainly due to two reasons. Firstly, the commit interval in project C is significantly longer compared to other projects, resulting in a less queuing phenomenon. Secondly, the execution duration of *failed* CIs is noticeably lower than that of *passed* CIs. Therefore, in project C, a higher failure rate implies more instances where individual commits are executed separately with a result of *failed*.

There is uncertainty regarding the correlation between execution duration and $S_{ci\_m(M,N)}$. Since the execution

TABLE XI
RESULTS OF PEARSON CORRELATION ANALYSIS BETWEEN THE THREE PROJECT FEATURES AND $S_{wait(M,N)}$

Features	A		B		C		D		E		F	
	r	p	r	p	r	p	r	p	r	p	r	p
Commit interval	-0.76	0.00*	-0.37	0.05	-0.72	0.00*	-0.21	0.26	-0.44	0.01	**0.57**	0.00*
Failure rate	-0.48	0.00*	-0.40	0.05	0.39	0.00*	-0.56	0.26	-0.31	0.01	-0.53	0.00*
Execution duration	-0.28	0.14	-0.16	0.40	-0.34	0.06	-0.41	0.02	-0.20	0.28	-0.17	0.38

* Means close to but not equal to zero.

duration of *failed* and *passed* executions is different, the average execution duration is affected by the failure rate. Furthermore, the occurrence frequency and severity of queuing phenomena are determined by both commit interval and execution duration. Therefore, not all projects show a correlation between execution duration and $S_{ci\_m(M,N)}$.

2) The Impacts of Project Features on the Time Cost for Waiting Before Executing CI (RQ2.2): As shown in Table XI, the relationships between project features and $S_{wait(M,N)}$ are similar to their relationships with $S_{ci\_m(M,N)}$.

The commit interval has a significant negative correlation with $S_{wait(M,N)}$ in most cases. On project D, we do not observe a significant correlation. The range of commit interval in Project D varies from 23 to 92. When the commit interval is greater than 60, we observe that $S_{wait(M,N)}$ is less than 22.2%. However, when the commit interval is less than 60, the mean value of $S_{wait(M,N)}$ is 37.4%. Therefore, although the commit interval and $S_{wait(M,N)}$ do not show a linear correlation, there may be a negative correlation between the two. We observe a positive correlation on project F. The failure rate of Project F is very high. A lower commit interval will increase the probability that multiple commits being executed as a batch. However, a high failure rate means that more cost is required to locate defects by RIDEC.

The failure rate has a significant negative correlation with $S_{wait(M,N)}$ in most cases. We observe no significant correlation on Project D, whilst we observe a positive correlation on Project C. For project C, even in samples with smaller commit intervals, the frequency of queuing is at a low level. As the execution duration of *failed* is shorter than that of *passed*, a high failure rate would lead to an increase in $S_{wait(M,N)}$.

There is uncertainty regarding the correlation between execution duration and $S_{wait(M,N)}$. It is similar to the results we observe regarding the correlation between execution duration and $S_{ci\_m(M,N)}$.

Existing predictors can guarantee that both $S_{ci\_m(M,N)}$ and $S_{wait(M,N)}$ are greater than 0 in most cases. We analyzed the results of 180 samples and found for 93% of them, at least one predictor out of the five predictors did not produce negative effects. When the commit interval is less than 20.59 minutes or the failure rate is less than 0.61, there is a 95% possibility that at least one of the five predictors can save time costs ($S_{ci\_m(M,N)} > 0$ and $S_{wait(M,N)} > 0$). When the commit interval is less than 16.48 minutes or the failure rate is less than 0.1, the probability will increase to 100%.

Main findings for RQ2: both commit interval and failure rate are important to project features that at least significantly negatively correlate with either $S_{ci\_m(M,N)}$ or $S_{wait(M,N)}$, while the effect of execution duration is unclear and requires further investigation. Among our experimental samples, when the commit interval is less than 20.59 minutes or the failure rate is less than 0.61, there is a 95% possibility that at least one of the five predictors can save time costs ($S_{ci\_m(M,N)} > 0$ and $S_{wait(M,N)} > 0$).

C. The Impacts of Predictors' Performance (What)

Currently, in the problem of predictive CI, it is almost impossible to obtain an ideal predictor with an accuracy close to 1. Therefore, one has to choose a relatively better predictor among multiple predictors with different performance. One key issue in selecting a predictor is the trade-off between different prediction performances such as precision and recall. The trade-off should be based on maximizing time savings as much as possible. Hence, we performed the Pearson Correlation Tests on common prediction performance metrics and two metrics of relative savings (i.e. $S_{ci\_m(M,N)}$ and $S_{wait(M,N)}$). The prediction performance metrics we analyzed include Accuracy ($Acc.$), Precision of *failed* ($Pre.$), Precision of *passed* ($Pre.'$), Recall of *failed* ($Rec.$), Recall of *passed* ($Rec.'$), F-measure of *failed* (F_1), and F-measure of *passed* (F_1'). More importantly than choosing a predictor, this correlation analysis will help provide optimization directions for designing new algorithms.

We grouped the samples of all projects according to project features. For each project feature, we divided the samples into two groups based on the values of the feature. One group consists of samples with feature values greater than the median, and the other group consists of remaining samples. Therefore, we obtained a total of eight groups of samples. For each group, we performed Pearson Correlation Analysis between the prediction performance metrics and time savings. The results are shown in Tables XII and XIII. We use "l,h,h" to represent a group of samples with low commit interval, high failure rate, and high execution duration, and the same for others.

1) The Impacts of Prediction Performance on the Time Cost for Executing CI (RQ3.1): **For different groups, performance metrics related to $S_{ci\_m(M,N)}$ vary.** This suggests

TABLE XII
RESULTS OF PEARSON CORRELATION ANALYSIS BETWEEN THE PREDICTION PERFORMANCE METRICS AND $S_{ci\_m(M,N)}$

Metrics	l,l,l		l, l, h		l, h, l		l, h, h		h, l, l		h, l, h		h, h, l		h, h, h	
	r	p	r	p	r	p	r	p	r	p	r	p	r	p	r	p
$Acc.$	0.13	0.13	-0.11	0.31	-0.05	0.63	0.03	0.72	-0.03	0.66	0.23	0.08	-0.07	0.63	0.15	0.05
$Pre.$	-0.23	0.01	-0.06	0.55	-0.44	0.00*	-0.48	0.00*	0.07	0.39	0.29	0.02	-0.11	0.41	-0.21	0.01
$Pre.'$	0.24	0.01	0.24	0.02	0.34	0.00*	0.36	0.00*	-0.32	0.00*	-0.16	0.21	-0.17	0.23	0.29	0.00*
$Rec.$	-0.17	0.04	-0.02	0.82	-0.14	0.17	-0.01	0.90	-0.06	0.41	-0.19	0.14	0.01	0.93	0.00*	0.98
$Rec.'$	0.08	0.33	-0.12	0.24	0.04	0.67	0.04	0.64	0.01	0.93	0.23	0.07	-0.04	0.75	0.10	0.19
F_1	-0.25	0.00*	-0.11	0.29	-0.29	0.00*	-0.25	0.00*	0.05	0.49	0.20	0.12	0.02	0.91	-0.08	0.31
F_1'	0.12	0.17	-0.10	0.34	0.23	0.02	0.21	0.02	-0.01	0.86	0.24	0.06	-0.07	0.61	0.26	0.00*

* Means close to but not equal to zero.
h,l,l denotes the sample set with high commit interval, low failure rate, and low execution duration, others similarly.

TABLE XIII
RESULTS OF PEARSON CORRELATION ANALYSIS BETWEEN THE PREDICTION PERFORMANCE METRICS AND $S_{wait(M,N)}$

Metrics	l,l,l		l, l, h		l, h, l		l, h, h		h, l, l		h, l, h		h, h, l		h, h, h	
	r	p	r	p	r	p	r	p	r	p	r	p	r	p	r	p
$Acc.$	0.19	0.02	0.01	0.96	0.09	0.37	0.22	0.02	0.04	0.65	0.34	0.01	0.05	0.72	0.16	0.03
$Pre.$	-0.29	0.00*	-0.19	0.06	-0.50	0.00*	-0.39	0.00*	0.22	0.00*	0.21	0.10	-0.15	0.28	-0.21	0.01
$Pre.'$	0.34	0.00*	0.42	0.00*	0.39	0.00*	0.58	0.00*	-0.32	0.00*	-0.05	0.69	-0.27	0.05	0.22	0.00*
$Rec.$	-0.05	0.59	-0.09	0.37	-0.26	0.01	0.10	0.27	0.00*	0.95	-0.22	0.09	-0.15	0.29	-0.07	0.37
$Rec.'$	0.13	0.13	-0.04	0.69	0.23	0.03	0.06	0.51	0.07	0.35	0.33	0.01	0.09	0.50	0.14	0.08
F_1	-0.29	0.00*	-0.26	0.01	-0.37	0.00*	-0.02	0.85	0.23	0.00*	0.25	0.06	-0.08	0.54	-0.07	0.38
F_1'	0.19	0.03	0.02	0.83	0.43	0.00*	0.35	0.00*	0.06	0.43	0.37	0.00*	0.06	0.68	0.27	0.00*

* Means close to but not equal to zero.
h,l,l denotes the sample set with high commit interval, low failure rate, and low execution duration, others similarly.

that it is difficult to draw conclusions about what kind of predictors can save more time for executing CI regardless of project features. For samples with high commit interval and low execution duration, almost all the prediction performance metrics have no significant correlation with $S_{ci\_m(M,N)}$, because it is difficult for predictions to be effective in the situation where the frequency of queuing is at a low level. When the commit interval is low and the execution duration is high, the frequency of queuing would be high and in this case, predictive CI would play a more significant role in time saving. Furthermore, if the failure rate is also at a low level, $S_{ci\_m(M,N)}$ would only be positively correlated to the precision of *passed*. However, other scenarios are more complex as $S_{ci\_m(M,N)}$ is related to multiple performance metrics.

Accuracy, recall of *passed*, and recall of *failed* do not correlate with the effectiveness of predictors. In most groups, none of them showed a significant correlation with $S_{ci\_m(M,N)}$.

2) The Impacts of Prediction Performance on the Time Cost for Waiting Before Executing CI (RQ3.2): **For different groups, performance metrics related to $S_{wait(M,N)}$ vary.** Overall, the results of the correlation test are similar to those of RQ3.1. None of the prediction performance metrics significantly correlate with $S_{wait(M,N)}$ across all groups. However none of the metrics are completely unrelated to $S_{wait(M,N)}$ across all groups either.

Both precision of *passed* and precision of *failed* appear to be the metrics that should be given more attention. These two measures show correlation with $S_{wait(M,N)}$ in more groups, although the polarity of the correlation is uncertain. Comprehensive metrics such as accuracy and F-measure cannot effectively indicate the effectiveness of predictive CI though.

> Main findings for RQ3: the correlations between predictors' performance metrics and time savings are not strong, and the strength of correlations can be affected by project features. Overall, both the precision of *failed* and the precision of *passed* should be given more attention.

VII. DISCUSSION

This section discusses the practical and research implications of predictive CI and the benefits of simulation-based evaluation.

A. Practical and Research Implications (How)

Replication package. We share the replication package of this work on GitHub [23], which contains the data we used, the scripts for processing the data, and the source files of the CISimulator. CISimulator can be run with *AnyLogic (version 8.7.6)*, the detailed instructions are also included in the package.

Suggestions to practitioners. We recommend using CISimulator to develop a configuration plan for the CI process. As we discussed in Sec. VI-A, the effectiveness of predictors varies across projects. It is not practical to provide a definite threshold for each project feature as a watershed for determining whether CI prediction should be used. The CISimulator can help with decision-making. Furthermore, the result of RQ2 suggests that commit interval and failure rate have significant negative impacts on the effectiveness of CI. For those projects with low

failure rates and high code submission frequencies that lead to obvious queuing phenomena, we suggest trying predictive CI with RIDEC.

Instructions to practitioners. We have the following instructions for practitioners who want to use the simulation-based approach to evaluating predictive CI.

First, improving the precision of prediction remains very valuable, but mainly for the purpose of enhancing the value of fast feedback. As discussed in Sec. VI-C, the correlations between prediction performance and time savings are not strong. The improvement in prediction performance has limited benefits to increasing cost savings in many projects, especially those with a very low failure rate. That is, the main value of improving prediction performance lies in being able to provide developers with more accurate feedback, rather than more cost savings. Compared with strategies such as random skipping or other similar approaches, providing immediate feedback is the greatest advantage of predictive CI, and this advantage aligns with the goals of CI. Nevertheless, when we have to make trade-offs on different prediction performance metrics, the ability to achieve higher cost savings would be an important consideration. For example, if we want to save the average waiting time before CI, we may need to pay more attention to the precision of prediction.

Second, predictive CI should be used in conjunction with RIDEC. Whether using random skipping or a predictive CI method, the resulting increase in the difficulty of defect fixing should be considered. Our proposed RIDEC can identify defective commits from multiple commits, making defect fixing with and without prediction equally difficult. In the future, the combination of RIDEC and more fine-grained defect localization methods and the evaluation of the effect of the combination of multiple support methods that can be used in the CI pipeline are issues worth investigating.

Third, nightly execution of RIDEC may be a better option. The benefits of predictive CI can be maximized if the RIDEC is performed at night. This delays feedback on defective commits but can speed up the frequency of code integration. Predictive CI is suitable for scenarios with insufficient server resources. In the case of insufficient resources due to frequent pull requests by developers during the day, predictive CI actually plays the role of prioritization. Nightly execution of RIDEC is a reasonable resource prioritization strategy in conjunction with predictive CI.

Fourth, the value of the server varies from project to project, and the evaluation results may change accordingly. To facilitate the unification of the dimensions, we only configured one server in all simulation experiments for three RQs so that the influence of the number of servers is reflected in the resource queuing time. In fact, different projects vary in the degree of restriction of server resources. Only by configuring experiments based on the actual number of servers can more accurately evaluate the effectiveness of the predictive CI method. In the case that the number of servers is unlimited (in this case, the effectiveness improvement of CI may become a false proposition), the resource queuing time will be significantly reduced, and the proportion of each component of the time cost will also change significantly. Therefore, some of the conclusions of this study may only be applicable to scenarios under the current definition of resource value.

Last, CISimulator constructed in this research is not the only implementation. We select the discrete event simulation paradigm based on the available data and target modeling granularity. We chose the *AnyLogic* simulation tool based on our expertise. These are not the only options. Even with *AnyLogic*, the same function can be achieved through different model blocks. The fundamental purpose of simulation modeling is to quantify the real world as accurately as possible. This is the basic principle that needs to be followed.

Suggestions to researchers. Research of predictive CI should probably focus more on the class-balanced scenario. For projects with extremely **unbalanced** ratio of *passed* and *failed* (very low failure rate), e.g., project A (mean failure rate is 4%) and C (mean failure rate is 5%), we found that even if the prediction performance reaches a very high level (e.g., the ideal predictor), the gap between random skipping (e.g., skip-100) and the predictor is still very small (referring to Figs. 6 and 7). It indicates that a simple random skipping strategy may be good enough when the failure rate is at a low level. As we discussed in Sec. VI-A, for one of our selected projects with the most **balanced** ratio of *passed* and *failed*, i.e. project F, random skipping may have negative effects whilst an ideal predictor can significantly save time. However, none of the real predictors we evaluated provided satisfactory results. For most projects, predictive CI is a class-imbalance prediction problem, therefore our past research [18] and other related research [20] were primarily focused on the class-imbalance problem. We suggest that future research focus more on improving the prediction performance for projects with high failure rates.

For further research on predictive CI, we suggest researchers use CISimulator to evaluate and compare other novel predictors. It is also possible to further expand on the basis of our model, such as designing and simulating other CI strategies; modeling other stages related to CI, such as defect fixing. We also encourage researchers to apply simulation-based evaluation to other machine-learning problems in software engineering.

B. Benefits of Simulation-Based Evaluation

Simulation can be specially designed to evaluate predictors based on process concerns, such as cost. It is true that predictors that perform better on various metrics mean higher performance. However, in practice, we are unlikely to find a predictor that performs the best on all metrics. To deal with this problem, researchers usually use some comprehensive metrics, such as F-measure and accuracy. However, this still resides in the performance of the predictor itself, rather than looking holistically at the prediction problem from the perspective of the overall CI process. Our simulation experiment for RQ3 indicates that the prediction performance metrics do not have a strong correlation to the cost savings. Furthermore, evaluation metrics are susceptible to class imbalance [64], whilst simulations are not.

Simulation provides the possibility to compare the presence/absence of predictions. Software projects are constantly

evolving, and we have no chance to compare the processes with and without predictions under the premise of fully controlling variables. Even if the uncontrolled comparison is acceptable, the cost of simulation is much lower and more affordable. The machine learning metrics cannot be used for such comparison, since these metrics will not exist for a process without prediction. Simulation can indicate whether the prediction is effective for the process, which is not possible with machine learning metrics.

Simulation can be used to analyze the process and has less dependence on data. Our research shows that the features of the project and the performance of the predictor will significantly affect the choice of predictors (referring to Sec. VI-C). We can carry out more simulation experiments without collecting more data and identify the objective laws between various factors and the effectiveness of the predictor, which can neither be directly revealed through machine learning metrics.

VIII. Threats to Validity

A. External Validity

Referring to our previous work [18], we selected six representative GitHub projects as data sources and conducted a comparison of five different performance predictors. To make the results more generalizable within a limited selection, we did not follow the principle of randomness in the selection of projects and predictors but rather chose samples at different positions based on their rankings. For the data in each project, we performed random sampling by time segment multiple times, so as to obtain a richer sample set from the same project (which can control for a large number of unknown variables, such as project context). We provide a replication package, so researchers can try our method on more data.

The investigation of predictive CI is one of our research purposes. More importantly, we aim to present a simulation-based approach for evaluating the use of machine learning in software processes. From this point of view, we only used predictive CI as a single case to illustrate the value of simulation-based evaluation. It remains to be studied in the future whether the approach can be applied to more software engineering problems.

The continuous integration process we simulated is mainly referred to descriptions in the official documents of Travis CI. Although Travis CI is a popular tool and related work is usually focused on open-source projects using Travis CI, the general significance of our study is limited. Some large-scale commercial projects may contain multiple levels of integration tests with different granularity. Such scenarios are more complex and project-specific than our simulated process. These deserve more in-depth study but are beyond the scope of this work.

B. Construct Validity

We applied a simulation-based approach to evaluating the effectiveness of predictive CI. We adopt time cost metrics that are concerned in practice, and at the same time, we proposed and simulated the RIDEC method to account for the potential negative impacts of predictions. Although we verified and validated CISimulator, the simulation itself has certain limitations. CISimulator is designed at the granularity of commit, which assumes that all commits only occupy one server and complete one time of CI. In practice, a commit triggering a CI may generate multiple jobs, each of which corresponds to a server environment and requires the CI to be executed on the corresponding server. Since each type of job corresponds to a specific server, it is reasonable to take multiple jobs as a whole, and meanwhile, take the corresponding servers as a whole. The granularity of the simulation in this study is appropriate since our research was designed towards commits. Besides, more fine-grained simulation has higher requirements for data.

In the process of model calibration, we conducted the Kolmogorov-Smirnov test and selected an appropriate distribution for each variable based on the Kolmogorov-Smirnov Distance. However, we can only obtain a close approximation rather than simulate a distribution that is completely identical to the true one. We performed model validation and the results showed that the differences in distributions and other factors combined resulted in acceptable relative errors.

Simulation depicts real-world dynamic processes using limited, known variables. Hence, simulation models are inevitably based on some assumptions, as we explained in Sec. III-D. Every run of simulation will not be a true reproduction of history, that is, it will not get 100% the same results as history. Rather, it is a re-creation based on actual historical distributions. What our assumptions describe is a combination of several possible influencing factors that underlie the actual data. We used the Monte Carlo method [43] to reflect the possible distribution of the results and mitigate the bias caused by randomness in the simulation. We validated CISimulator to confirm that the actual results are within the distribution range of simulation results, which indicates that our assumptions are acceptable (the impacts are tolerable). The gap between reality and simulation is manifested as the distance between such uncertainty and determinism, which arises from unknown variables in the real world. In the simulation, we attempt to describe them using assumptions.

C. Internal Validity

We sampled enough of each project to run simulation experiments. When analyzing, we grouped according to variable features to obtain some degree of variable control. We did not conduct a simulation experiment with fully controlled variables, even though it is easy to do so in simulations. Because there may be correlations between variables, for example, the failure rate of CI may be related to commit frequency. We performed Pearson Correlation Analysis to illustrate the relationship between time cost savings and predictor features, and project features.

D. Conclusion Validity

In the experiment, we compared three scenarios, i.e. not using any strategy, using random skipping strategy, and using predictive CI strategy. We can compare the results before and after using the strategy while controlling for all other variables. In statistical tests, we ensured that the sample size was greater than 30 and set the significance level to 0.05. However, there are still

some threats to the conclusion validity. The use of predictive CI will inevitably change the feedback efficiency and feedback mechanism of CI. Developers in the process may respond to this change, such as increasing the frequency of code submission, as well as exploring some means to make code submission easier to be predicted as *passed*. The possibility, extent, and potential impact of such reactions are unknown.

IX. Conclusion

This study presents a novel approach to evaluating the impact of predictors on the CI process using process simulation techniques. We develop a DES model (CISimulator) to simulate the CI process with and without predictive CI, which can help researchers holistically evaluate the value of predictive CI more practically from a process perspective. We find that although the existing predictors do not perform well on common evaluation metrics of machine learning predictors, simulations suggest that they are still able to save the time cost for executing CI as well as the average waiting time before executing CI. The simulation results indicate that their deviation from the ideal predictor is also limited. The value of prediction is far more than the reduction of the executions of CI that existing studies [20], [24] focused on, and the value in reducing queue time for waiting servers is even greater. Predictive CI also has limitations, in the cases of very small proportions of *failed*, predictive CI may not be significantly better than random skipping. Future research should pay more attention to prediction performance in scenarios with high failure rates. The effectiveness of predictors is influenced by multiple factors, including project features and the prediction performance of the predictors themselves, but the project features generally have a more decisive impact than prediction performance.

This research demonstrates the value of using simulation to evaluate machine learning-based predictors. The simulation-based evaluation approach has the following advantages: 1) it solves the issue of insufficient indication of common metrics in imbalanced learning problems; 2) it solves the issue that common metrics cannot directly measure the practical value of predictors in software processes; 3) it provides the ability to analyze dynamic processes compared with statistical analysis; and 4) its cost is significantly lower than that of carrying out experiments with the actual projects (it might be the only feasible means of such evaluation in many cases).

References

[1] B. Vasilescu, Y. Yu, H. Wang, P. T. Devanbu, and V. Filkov, "Quality and productivity outcomes relating to continuous integration in GitHub," in *Proc. 10th Joint Meeting Found. Softw. Eng. (ESEC/FSE)*, 2015, pp. 805–816.

[2] J. Downs, B. Plimmer, and J. G. Hosking, "Ambient awareness of build status in collocated software teams," in *Proc. 34th Int. Conf. Softw. Eng. (ICSE)*, 2012, pp. 507–517.

[3] M. Hilton, T. Tunnell, K. Huang, D. Marinov, and D. Dig, "Usage, costs, and benefits of continuous integration in open-source projects," in *Proc. 31st IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Singapore, Sep. 2016, pp. 426–437.

[4] D. Ståhl and J. Bosch, "Experienced benefits of continuous integration in industry software product development: A case study," in *Proc. IASTED Int. Conf. Softw. Eng. (IASTED)*, 2013, pp. 736–743.

[5] E. I. Laukkanen, J. Itkonen, and C. Lassenius, "Problems, causes and solutions when adopting continuous delivery – A systematic literature review," *Inf. Softw. Technol.*, vol. 82, pp. 55–79, 2017.

[6] R. O. Rogers, "Scaling continuous integration," in *Proc. 26th Int. Conf. Softw. Eng. (ICSE)*, 2004, pp. 68–76.

[7] J. Rasmusson, "Long build trouble shooting guide," in *Proc. 4th Extreme Program. Agile Methods – XP/Agile Universe*, Berlin, Heidelberg: Springer-Verlag, 2004, pp. 13–21.

[8] M. Hilton, N. Nelson, T. Tunnell, D. Marinov, and D. Dig, "Trade-offs in continuous integration: Assurance, security, and flexibility," in *Proc. 11th Joint Meeting Found. Softw. Eng. (ESEC/FSE)*, 2017, pp. 197–207.

[9] M. Fowler. "Continuous integration." Martin Fowler. Accessed: May 1, 2006. [Online]. Available: http://martinfowler.com/articles/continuousIntegration.html

[10] T. A. Ghaleb, D. A. da Costa, and Y. Zou, "An empirical study of the long duration of continuous integration builds," *Empirical Softw. Eng.*, vol. 24, no. 4, pp. 2102–2139, 2019.

[11] R. Abdalkareem, S. Mujahid, E. Shihab, and J. Rilling, "Which commits can be CI skipped," *IEEE Trans. Softw. Eng.*, vol. 47, no. 3, pp. 448–463, Mar. 2021.

[12] A. E. Hassan and K. Zhang, "Using decision trees to predict the certification result of a build," in *Proc. 21st IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, 2006, pp. 189–198.

[13] T. Wolf, A. Schröter, D. E. Damian, and T. H. D. Nguyen, "Predicting build failures using social network analysis on developer communication," in *Proc. 31st Int. Conf. Softw. Eng. (ICSE)*, 2009, pp. 1–11.

[14] J. Xia and Y. Li, "Could we predict the result of a continuous integration build? An empirical study," in *Proc. IEEE Int. Conf. Softw. Qual., Rel. Secur. Companion (QRS-C)*, 2017, pp. 311–315.

[15] F. Hassan and X. Wang, "Change-aware build prediction model for stall avoidance in continuous integration," in *Proc. 11th ACM/IEEE Int. Symp. Empirical Softw. Eng. Meas. (ESEM)*, 2017, pp. 157–162.

[16] J. Xia, Y. Li, and C. Wang, "An empirical study on the cross-project predictability of continuous integration outcomes," in *Proc. Workshop Inf. Secur. Appl. (WISA)*, 2017, pp. 234–239.

[17] A. Ni and M. Li, "Cost-effective build outcome prediction using cascaded classifiers," in *Proc. 14th Int. Conf. Mining Softw. Repositories (MSR)*, Piscataway, NJ, USA: IEEE Press, 2017, pp. 455–458.

[18] B. Liu, H. Zhang, L. Yang, L. Dong, H. Shen, and K. Song, "An experimental evaluation of imbalanced learning and time-series validation in the context of CI/CD prediction," in *Proc. Int. Conf. Eval. Assessment Softw. Eng. (EASE)*, Trondheim, Norway. New York, NY, USA: ACM, 2020, pp. 21–30.

[19] R. Abdalkareem, S. Mujahid, and E. Shihab, "A machine learning approach to improve the detection of CI skip commits," *IEEE Trans. Softw. Eng.*, vol. 47, no. 12, pp. 2740–2754, Dec. 2021.

[20] B. Chen, L. Chen, C. Zhang, and X. Peng, "BuildFast: History-aware build outcome prediction for fast feedback and reduced cost in continuous integration," in *Proc. 35th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Australia, Sep. 2020, pp. 42–53.

[21] J. Brabec, T. Komárek, V. Franc, and L. Machlica, "On model evaluation under non-constant class imbalance," in *Proc. Int. Conf. Comput. Sci.*, 2020, pp. 74–87.

[22] T. Abdel-Hamid and S. E. Madnick, *Software Project Dynamics: An Integrated Approach*. Upper Saddle River, NJ, USA: Prentice-Hall, 1991.

[23] B. Liu, W. M. He Zhang, G. Li, S. Li, and H. Shen. "Cisimulator." GitHub. Accessed: Oct. 13, 2023. [Online]. Available: https://github.com/gyli99/CISimulator

[24] X. Jin and F. Servant, "A cost-efficient approach to building in continuous integration," in *Proc. 42nd Int. Conf. Softw. Eng. (ICSE)*, Seoul, Republic of Korea, G. Rothermel and D. Bae, Eds. New York, NY, USA: ACM, Jun. 2020, pp. 13–25.

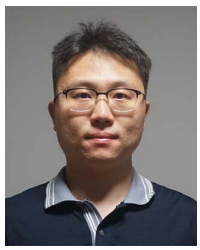
[25] X. Jin and F. Servant, "What helped, and what did not? An evaluation of the strategies to improve continuous integration," in *Proc. IEEE/ACM 43rd Int. Conf. Softw. Eng. (ICSE)*, Piscataway, NJ, USA: IEEE Press, 2021, pp. 213–225.

[26] I. Saidani, A. Ouni, M. Chouchen, and M. W. Mkaouer, "Predicting continuous integration build failures using evolutionary search," *Inf. Softw. Technol.*, vol. 128, Dec. 2020, Art. no. 106392.

[27] M. Beller, G. Gousios, and A. Zaidman, "Travistorrent: Synthesizing travis CI and GitHub for full-stack research on continuous integration," in *Proc. 14th Int. Conf. Mining Softw. Repositories (MSR)*, Buenos Aires, Argentina, May 2017, pp. 447–450.

[28] A. Schröter, "Predicting build outcome with developer interaction in Jazz," in *Proc. 32nd ACM/IEEE Int. Conf. Softw. Eng.*, 2010, vol. 2, pp. 511–512.

[29] J. Finlay, R. Pears, and A. M. Connor, "Data stream mining for predicting software build outcomes using source code metrics," *Inf. Softw. Technol.*, vol. 56, no. 2, pp. 183–198, 2014.
[30] J. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in *Proc. ACM/IEEE Int. Conf. Automated Softw. Eng. (ASE)*, Vasteras, Sweden, Sep. 15–19, 2014, pp. 313–324.
[31] T. Chen and C. Guestrin, "XGBoost: A scalable tree boosting system," in *Proc. 22nd ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining (SKDD)*, 2016, pp. 785–794.
[32] Z. Xie and M. Li, "Cutting the software building efforts in continuous integration by semi-supervised online AUC optimization," in *Proc. 27th Int. Joint Conf. Artif. Intell. (IJCAI)*, 2018, pp. 2875–2881.
[33] A. Maria, "Introduction to modeling and simulation," in *Proc. 29th Conf. Winter Simul.*, 1997, pp. 7–13.
[34] P. J. Sánchez, "As simple as possible, but no simpler: A gentle introduction to simulation modeling," in *Proc. 38th Conf. Winter Simul.*, 2006, pp. 2–10.
[35] M. I. Kellner, R. J. Madachy, and D. M. Raffo, "Software process simulation modeling: Why? What? How?" *J. Syst. Softw.*, vol. 46, nos. 2–3, pp. 91–105, 1999.
[36] H. Zhang, B. A. Kitchenham, and D. Pfahl, "Software process simulation modeling: An extended systematic review," in *Proc. New Model. Concepts Today's Softw. Processes, Int. Conf. Softw. Process(ICSP)*, Paderborn, Germany, Jul. 2010, pp. 309–320.
[37] M. Felderer and G. H. Travassos, "The evolution of empirical methods in software engineering," in *Contemporary Empirical Methods in Software Engineering*, M. Felderer and G. H. Travassos, Eds., Cham, Switzerland: Springer-Verlag, 2020, pp. 1–24.
[38] T. Baum, F. Kortum, K. Schneider, A. Brack, and J. Schauder, "Comparing pre-commit reviews and post-commit reviews using process simulation," *J. Softw., Evol. Process*, vol. 29, no. 11, p. e1865, 2017.
[39] V. Garousi and D. Pfahl, "When to automate software testing? A decision-support approach based on process simulation," *J. Softw., Evol. Process*, vol. 28, no. 4, pp. 272–285, 2016.
[40] B. Liu, H. Zhang, and S. Zhu, "An incremental V-model process for automotive development," in *Proc. 23rd Asia-Pacific Softw. Eng. Conf. (APSEC)*, 2016, pp. 225–232.
[41] L. von Rueden, S. Mayer, R. Sifa, C. Bauckhage, and J. Garcke, "Combining machine learning and simulation to a hybrid modelling approach: Current and future directions," in *Proc. Int. Symp. Intell. Data Anal.*, 2020, pp. 548–560.
[42] S. Zeiml, K. Altendorfer, T. Felberbauer, and J. Nurgazina, "Simulation based forecast data generation and evaluation of forecast error measures," in *Proc. Winter Simul. Conf.*, 2019, pp. 2119–2130.
[43] N. Metropolis and S. Ulam, "The Monte Carlo method," *J. Amer. Statistical Assoc.*, vol. 44, no. 247, pp. 335–341, 1949.
[44] G. P. Richardson and A. I. Pugh, *Introduction to System Dynamics Modeling With Dynamo*, Milton Park, Oxfordshire, U.K.: Taylor & Francis, 1981.
[45] D. M. Raffo and M. I. Kellner, "Empirical analysis in software process simulation modeling," *J. Syst. Softw.*, vol. 53, no. 1, pp. 31–41, 2000.
[46] B. A. Kitchenham, L. Pickard, S. Linkman, and P. Jones, "A framework for evaluating a software bidding model," *Inf. Softw. Technol.*, vol. 47, no. 11, pp. 747–760, 2005.
[47] C. Gao, S. Jiang, and G. Rong, "Software process simulation modeling: Preliminary results from an updated systematic review," in *Proc. Int. Conf. Softw. Syst. Process (ICSSP)*, 2014, pp. 50–54.
[48] A. Borshchev and A. Filippov, "From system dynamics and discrete event to practical agent based modeling: Reasons, techniques, tools," in *Proc. 22nd Int. Conf. Syst. Dyn. Soc. (ICSDS)*, 2004, pp. 25–29.
[49] C. Gao, H. Zhang, and S. Jiang, "Constructing hybrid software process simulation models," in *Proc. Int. Conf. Softw. Syst. Process (ICSSP)*, New York, NY, USA: ACM, 2015, pp. 157–166.
[50] T. Rausch, W. Hummer, P. Leitner, and S. Schulte, "An empirical analysis of build failures in the continuous integration workflows of Java-based open-source software," in *Proc. 14th Int. Conf. Mining Softw. Repositories (MSR)*, Piscataway, NJ, USA: IEEE Press, 2017, pp. 345–355.
[51] F. J. Massey, "The Kolmogorov-Smirnov test for goodness of fit," *J. Amer. Statistical Assoc.*, vol. 46, no. 253, pp. 68–78, 1951.
[52] H. Gong, H. Zhang, D. Yu, and B. Liu, "A systematic map on verifying and validating software process simulation models," in *Proc. 10th Int. Conf. Softw. Syst. Process (ICSSP)*, Paris, France, Jul. 2017, pp. 50–59.
[53] Q. Dai, "Research on the code-security oriented commit prioritization method in continuous integration," Master's dissertation, Nanjing Univ., China, 2021.
[54] M. R. Smith, T. R. Martinez, and C. G. Giraud-Carrier, "An instance level analysis of data complexity," *Mach. Learn.*, vol. 95, no. 2, pp. 225–256, 2014, doi: 10.1007/s10994-013-5422-z.
[55] L. Breiman, "Random forests," *Mach. Learn. Arch.*, vol. 45, no. 1, pp. 5–32, 2001.
[56] G. Louppe and P. Geurts, "Ensembles on random patches," in *Proc. Mach. Learn. Knowl. Discovery Databases – Eur. Conf. (ECML PKDD)*, Bristol, U.K., P. A. Flach, T. D. Bie, and N. Cristianini, Eds., vol. 7523. Berlin, Germany: Springer-Verlag, 2012, pp. 346–361, doi: 10.1007/978-3-642-33460-3_28.
[57] A. C. Bahnsen, D. Aouada, and B. E. Ottersten, "Ensemble of example-dependent cost-sensitive decision trees," 2015. [Online]. Available: http://arxiv.org/abs/1505.04637
[58] J. Laurikkala, "Improving identification of difficult small classes by balancing class distribution," in *Proc. 8th Conf. Artif. Intell. Med. Europe (AIME)*, Cascais, Portugal. Berlin, Germany: Springer, 2001, pp. 63–66.
[59] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone, *Classification and Regression Trees*. Wadsworth, 1984.
[60] M. M. Breunig, H. Kriegel, R. T. Ng, and J. Sander, "LOF: Identifying density-based local outliers," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, Dallas, TX, USA, W. Chen, J. F. Naughton, and P. A. Bernstein, Eds., New York, NY, USA: ACM, 2000, pp. 93–104, doi: 10.1145/342009.335388.
[61] M. D. Penta, M. Harman, and G. Antoniol, "The use of search-based optimization techniques to schedule and staff software projects: An approach and an empirical study," *Softw. Pract. Experience*, vol. 41, no. 5, pp. 495–519, 2011.
[62] G. Concas, M. I. Lunesu, M. Marchesi, and H. Zhang, "Simulation of software maintenance process, with and without a work-in-process limit," *J. Softw., Evol. Process*, vol. 25, no. 12, pp. 1225–1248, 2013.
[63] H. Neu, T. Hanne, J. Münch, S. Nickel, and A. Wirsen, "Simulation-based risk reduction for planning inspections," in *Proc. 4th Int. Conf. Product Focused Softw. Process Improvement (PROFES)*, Rovaniemi, Finland, Dec. 2002, pp. 78–93.
[64] L. A. Jeni, J. F. Cohn, and F. D. L. Torre, "Facing imbalanced data–Recommendations for the use of performance metrics," in *Proc. Humaine Assoc. Conf. Affect. Comput. Intell. Interact. (ACII)*, 2013, pp. 245–251.



Bohan Liu is an Assistant Researcher with Software Institute, Nanjing University, China. He undertakes research in software engineering, in particular software process mining and modeling, DevOps, empirical software engineering, and machine learning.

He Zhang is a Full Professor with Software Institute, the Director of the DevOps+ Research Laboratory, Nanjing University, China, and a Principal Scientist with CSIRO, Australia. He joined academia after many years working in software industry. He undertakes research in software engineering, in particular software process, software architecture, DevOps, software security, and empirical and evidence-based software engineering. He has published over 200 peer-reviewed papers in prestigious international conferences and journals.

Weigang Ma received the M.S. degree in software engineering from Nanjing University, in 2023, advised by Prof. He Zhang. His research interests include continuous integration, software process simulation, and software process mining.

Gongyuan Li received the M.S. degree from the Software Engineering Institute, Nanjing University, China, in 2023. He is now working toward the Ph.D. degree in the DevOps+ Research Laboratory at Nanjing University. His research interests include software process simulation and software engineering for artificial intelligence.

Shanshan Li received the B.S. degree in software engineering and the M.S degree in computer science and technology from China University of Petroleum (UPC), and the Ph.D. degree in software engineering from Nanjing University (NJU). She is an Assistant Researcher in Software Institute, Nanjing University. Her research interests include microservices architecture (MSA), blockchain-oriented software engineering (BOSE), and evidence-based software engineering (EBSE). She won the Best/Distinguished Paper Awards from APSEC 2016 and the Most Influential Paper Awards from Asia-Pacific Software Engineering Conference (APSEC) 2023. Currently, she is a Professional Member of China Computer Federation (CCF).

Haifeng Shen (Senior Member) is an Associate Professor in information technology and the Director of the Human-Centred Intelligent Learning & Software Technologies Research Lab (HilstLab) with the Faculty of Law and Business, Australian Catholic University. His research expertise revolves around intelligent software and interaction technologies, a unique intersection that brings together research in software engineering, human computer interaction, and artificial intelligence. He has published more than 150 peer-reviewed research papers at various journals and conferences most of which appear at high impact outlets such as IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, International Conference on Software Engineering (ICSE), International Conference on Automated Software Engineering (ASE), *ACM Transactions on Computer-Human Interaction (TOCHI)*, ACM Conference on Human Factors in Computing Systems (CHI), ACM Conference on Computer-Supported Cooperative Work and Social Computing (CSCW), ACM International Conference on Intelligent User Interfaces (IUI), and IEEE International Conference on Data Engineering (ICDE).