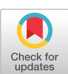

PDF Download
3779653.pdf
05 January 2026
Total Citations: 0
Total Downloads: 0

Accepted: 20 November 2025
Revised: 15 November 2025
Received: 30 May 2025

Citation in BibTeX format

RESEARCH-ARTICLE

Just-In-Time Quality Assurance with Assistance of Automated Defect Prediction and Code Testing

ACM Transactions on Software Engineering and Methodology
https://doi.org/10.1145/3779653
EISSN: 1557-7392

Just-In-Time Quality Assurance with Assistance of Automated Defect Prediction and Code Testing

DAWEI TIAN, Harbin Institute of Technology, China
LIYAN SONG*, Harbin Institute of Technology, China
XIAOHONG SU*, Harbin Institute of Technology, China

Software testing and defect prediction are widely recognized quality assurance activities that can contribute to the effectiveness and efficiency of Just-In-Time Software Quality Assurance (JIT-SQA) by automating processes and reducing software maintenance costs. However, most studies treat them separately, with limited exploration of how defect prediction can guide testing and how testing can support prediction. Existing methods are not tailored for JIT-SQA and often ignore change-specific characteristics such as concept drift and verification latency, leading to unstable defect prediction. Moreover, test annotations on recent code changes are rarely used to update models, missing an opportunity to improve accuracy. Beyond this, effectively leveraging test-annotated samples remains an open research question. On the one hand, these samples frequently suffer from one-sided label noise, as instances that are not identified as defective during testing may still contain defects. On the other hand, defect-inducing changes are relatively infrequent, leading to a significant class imbalance issue. Therefore, exploring how to utilize test annotations to mitigate class imbalance and enhance prediction robustness presents a critical challenge. To fill this research gap, we propose an automatic JIT-SQA framework that incorporates defect prediction and testing processes collaboratively. Specifically, we direct testing activities through defect prediction models and design a novel update mechanism that integrates waiting time with testing feedback to update the predictor in real time. Moreover, we further design a reinforcement learning-based dynamic selection mechanism to mitigate the instability in defect prediction performance over time. Our framework facilitates mutual guidance between automatic testing and defect prediction, leveraging shared knowledge within the JIT-SQA system. This integration holds significant potential for producing high-quality software within modern development timelines. Notably, this study is the first to investigate the integrated application of these two quality assurance activities within a unified framework for **JIT-SQA** throughout the software development process. We conducted experiments using both statically typed (Java) and dynamically typed (Python) languages by integrating various testing techniques. The results indicate that, after implementing the novel update and dynamic selection mechanisms, our framework significantly outperforms simple integrated frameworks of defect prediction and testing. Specifically, the Recall improves by 21.6%, 10.5%, and 6.3% compared with the testing-integrated versions of ODaSC, ECo-HumLa, and HumLa, respectively, while the F1 score improves by 13.2%, 7.0%, and 5.1%. At the same time, in terms of guiding testing activities, the total number of tests is reduced by 2.9%, 19.5%, and 24.8% compared with the same baselines. These findings demonstrate that our framework not only mitigates the adverse effects of concept drift, verification latency, and class imbalance in code evolution scenarios but also effectively guides testing to reduce resource consumption.

CCS Concepts: • **Software and its engineering** → Search-based software engineering; **Software evolution**; **Maintaining software**; *Software version control*; **Software testing and debugging**.

*Corresponding authors.

Authors' Contact Information: Dawei Tian, Harbin Institute of Technology, Harbin, Heilongjiang, China, davidtian@stu.hit.edu.cn; Liyan Song, Harbin Institute of Technology, Harbin, Heilongjiang, China, songly@hit.edu.cn; Xiaohong Su, Harbin Institute of Technology, Harbin, Heilongjiang, China, sxh@hit.edu.cn.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2026 Copyright held by the owner/author(s).
ACM 1557-7392/2026/1-ART
https://doi.org/10.1145/3779653

Just Accepted

Additional Key Words and Phrases: software quality assurance, automated testing, software defect prediction, just-in-time software defect prediction, change-level defect prediction

1 Introduction

Modern software development cycles involve continuous delivery of high-quality software products within short timeframes, posing significant challenges for Software Quality Assurance (SQA) practices. Continuous quality assurance is crucial for early defect detection, a practice referred to as *Just-In-Time Software Quality Assurance (JIT-SQA)* [33]. JIT-SQA facilitates ongoing quality control by allowing assessments immediately after a developer commits a **code change**. Unlike conventional SQA, which operates at the package, class, or file level, JIT-SQA focuses specifically on the finer **code change** level.

Software defect prediction [18, 45, 68, 78] and testing [21, 51, 52, 56, 62, 76] are widely used quality assurance activities that contribute to producing high-quality software while reducing maintenance costs. In the concept of JIT-SQA, after each code change commit, testing would be typically required to assess whether the code change has introduced any defects. Ensuring that the affected code is adequately exercised by existing test cases—and, when necessary, augmented with additional unit tests—is essential for maintaining software reliability. However, exhaustive testing of every code change is impractical due to the high computational and time costs, potentially impeding software developing efficiency. Since the majority of code changes do not contain observable faults, selective testing strategies are widely adopted in practice, such as prioritizing changes that affect critical functionalities or are associated with bug fixes. A principle and cost-effective strategy involves employing defect prediction techniques to estimate the likelihood that a change is fault-inducing, thereby enabling targeted testing of high-risk changes.

In the research community, prior studies have explored how defect prediction can guide testing to enhance efficiency. Furthermore, some works — such as FIP [82] — have aimed to leverage test feedback to refine defect prediction models based on logistic regression, addressing the limitation of these models in making effective predictions during the early stages of cross-project scenarios. However, these studies mainly focus on class-, module-, or method-level granularity, making them merely applicable within traditional SQA settings.

In the context of JIT-SQA, data exhibits a streaming nature, where code changes continuously arrive over time. The key challenge lies in achieving timely and effective defect prediction and quality assurance within such an evolving data environment. This stands in contrast to traditional SQA scenarios, which typically rely on statically collected historical data for model building and evaluation. However, data streams of code changes often exhibit temporal nature, whose data distribution would shift over time, known as *concept drift* [5, 7] (see Section 2.2) in Just-In-Time Software Defect Prediction (JIT-SDP). To mitigate the impact of concept drift, recent studies have mainly adopted online learning techniques [7, 43, 67, 70, 74], which continuously update models with newly labeled data. However, in real-world software development, whether a change introduces a defect is typically unknown at the commit time of changes, a challenge known as *verification latency* [13, 17, 35, 72] (see Section 2.2). Cabral et al. [7] introduced a waiting time update mechanism to address verification latency; however, this approach still risks producing outdated data. Some studies have explored incorporating human-assisted update mechanisms to facilitate more prompt labeling of code changes [70], but these approaches tend to be labor-intensive, challenging to automate, and thus impractical for large-scale implementation. Since defect prediction models are often used to guide testing, it is natural to consider leveraging test outcomes for immediate model updates. However, test outcomes are affected by one-sided noise – samples labeled as clean may still contain defects – and defect-inducing changes are relatively uncommon, leading to significant class imbalance. Thus, a key challenge lies in how to exploit test outcomes to both reduce label noise and alleviate class imbalance.

On the other hand, due to the combined effects of concept drift and verification latency, the performance of JIT-SDP models may fluctuate over time. From the perspective of predictive performance (e.g., F1-score), the same time-sequenced data stream of code changes can vary at different prediction time. In contrast, traditional software

Just Accepted

defect prediction (SDP) models typically remain unchanged after training, and relying on a fixed, pre-defined threshold to direct the allocation of testing resources. Such static threshold-based strategy is infeasible in JIT-SDP due to its constantly changing model performance. That being said, the fluctuation of predictive performance of JIT-SDP models suggests that one should only trust the output of the defect predictor when it can provide sufficiently good predictive performance. Yet, existing JIT-SDP approaches often overlook this trustworthiness assessment, leading to either over-testing or missed defects. Moreover, the inability to promptly obtain true labels of code changes complicates the assessment of the true performance of the SDP models. This challenge makes it difficult to determine the trustworthiness of the predictor's results. Consequently, there is a pressing need to design a dynamic selection mechanism to assess the model's reliability and guides decisions on whether to trust its outputs, depending on which one would decide whether software testing should be conducted.

To date, there has been no unified JIT-SQA framework that can effectively integrate defect prediction and automated testing activities in a collaborative manner. This study presents the first attempt to explore this integration by creating an automated framework for conducting JIT-SQA throughout the development process. Specifically, we designed a unified framework for JIT-SQA referred to as **Au**tomated **D**efect **I**dentifier with **T**esting and D**e**fect Pr**e**diction (AuDITee). The AuDITee framework implements the mutual coordination between the JIT-SDP model and the testing algorithm. Specifically, we propose a novel update mechanism for JIT-SDP that combines the **W**aiting **T**ime Update Mechanism with **T**esting (WTT), aiming to eliminate one-sided noise in test outcomes while simultaneously alleviating class imbalance. Moreover, we propose an advanced version of AuDITee, named AuDITee-RL, which leverages reinforcement learning (RL) as a dynamic selector. By integrating the outcomes of the defect prediction model, AuDITee-RL can dynamically decide whether testing should be applied, thereby alleviating the issue of performance fluctuate in the defect prediction model.

To assess the effectiveness and efficiency of the proposed JIT-SQA framework, we pose the following Research Questions (RQs):

- RQ1 To what extent can the proposed WTT update strategy (i.e., AuDITee and AuDITee-RL) – and the RL–based selector module (i.e., AuDITee-RL) mitigate the challenges posed by code change scenarios, which often degrade the performance of simple integrations of defect prediction and testing?
- RQ2 Compared to the simple integration of defect prediction and testing, can our framework (i.e., AuDITee and AuDITee-RL) detect a larger number of unique defects?
- RQ3 Compared with conventional defect prediction modules, how effectively can defect predictors enhanced with the proposed update mechanism (i.e., AuDITee and AuDITee-RL) reduce testing resource consumption? Furthermore, can the RL–based dynamic selector, which intelligently determines when to trigger testing (i.e., AuDITee-RL), further optimize resource utilization and decrease testing costs? Are the results sensitive to the choice of underlying testing frameworks?
- RQ4 To what extent can the proposed update mechanism (i.e., AuDITee and AuDITee-RL) mitigate the negative effects of concept drift and verification latency on prediction performance? How effective can it address the challenge of class imbalance? Furthermore, to what extent can the RL–based dynamic selector (i.e., AuDITee-RL) improve the performance of the defect prediction model by selecting high-quality samples?
- RQ5 Compared with conventional defect prediction modules, how much can defect predictors enhanced with the proposed update mechanism (i.e., AuDITee and AuDITee-RL) improve the quality of guided testing? Can the RL–based dynamic selector (i.e., AuDITee-RL), which intelligently determines when to trigger testing, further enhance the quality of guided testing?

These RQs represent distinct perspectives on the shared goal of finding early defects. RQ1, RQ2 and RQ3 focuses on the overall performance improvement and the resource savings of the framework; RQ4, RQ5 provide a detailed analysis of the performance of the defect prediction module and the testing module within the proposed framework. Furthermore, these research questions explore the factors contributing to the overall improvement in

Just Accepted

the performance of the framework. Our experimental evaluation, based on five large-scale open-source projects, demonstrates that the proposed AuDITee with newly WTT update mechanism and RL-based dynamic selector effectively enhances both the effectiveness and efficiency of JIT-SQA practices.

This paper makes several contributions. 1) We propose a novel update mechanism for JIT-SDP that combines the waiting time strategy with testing. 2) We design a dynamic selection mechanism to mitigate the issue of dynamic performance changes in the JIT-SDP model, and introduce a dynamic selector based on reinforcement learning tailored to specific project needs within the AuDITee framework, enabling a focus on either resource efficiency in testing or enhanced defect detection capabilities. 3) We reference commonly used metrics in the broader testing field and design a completely new metric to evaluate both testing effectiveness and efficiency in the JIT-SQA context, providing a comprehensive assessment of the framework's potential impact and establishing a benchmark for future research. 4) We conduct an extensive empirical evaluation based on 43,011 code changes from 5 open-source Java and Python projects to assess the effectiveness and efficiency of the proposed JIT-SQA framework.

2 Background

This section reviews key methodologies in defect prediction, encompassing both offline and online learning scenarios. It also examines complementary software testing techniques, including unit testing, automated test case generation, and test optimization strategies.

2.1 Offline Defect Prediction

Offline techniques leverage historical data and static code metrics to build predictive models that operate in a batch-processing mode. Two principal levels of analysis are commonly adopted:

Class-Level Prediction [47, 79]: This approach assesses each software class based on intrinsic attributes such as complexity, coupling, and cohesion, along with historical defect records. Because data for all classes are available simultaneously, offline models can compute defect probabilities for every class and rank them accordingly. This ranking system enables developers to prioritize classes that are statistically more likely to contain defects, facilitating efficient allocation of testing and maintenance resources.

Code-Change-Level Prediction [36, 85]: Compared to class-level prediction techniques, this method (called JIT-SDP) enables the more timely detection of defects by focusing on modifications between code revisions, thereby facilitating prompt intervention during the software development process. These models analyze metrics related to code churn, change size, and other change-specific characteristics to evaluate defect likelihood. Although offline methods can theoretically rank a subset of code changes based on risk, collecting a set of changes before conducting defect ranking undermines the timeliness of fine-grained defect prediction. This delay hampers the rapid identification of defects during the software development process, thereby reducing the potential value and practical significance of such an approach. The primary aim is to assess the risk associated with individual changes rather than to generate a comprehensive ranking across the entire codebase.

2.2 Online Defect Prediction

In contrast to offline methods, online defect prediction techniques are designed for real-time or near-real-time analysis and are integrated into Continuous Integration (CI) or Continuous Deployment (CD) pipelines. They (a.k.a. online JIT-SDP) can provide immediate feedback on defect risk as code changes occur [31, 65, 67, 70]. The key features of online prediction include:

- **Timeliness:** Predictions are generated instantly after code changes, allowing for prompt identification of potential defects during the development process.

Just Accepted

- **Adaptability:** By continuously updating predictive models with the most recent data, online approaches can effectively capture evolving code characteristics and mitigate the impact of concept drift.
- **Integration with Developing Workflows:** Online systems are embedded within the developing environments, offering real-time alerts and actionable recommendations that enable developers to address potential defects promptly.

Concept Drift: One significant challenge in online JIT-SDP is the phenomenon of *concept drift* [5, 7, 43], where the statistical properties of code features change over time due to evolving development practices, shifting codebases, and changing team dynamics. Online methods, by virtue of their real-time nature, can help mitigate concept drift by training on the most recent data. However, in the context of online JIT-SDP, there exists an intrinsic challenge: when new code changes are introduced, the corresponding defect labels are typically not available immediately.

Waiting Time Update Mechanism: Cabral et al. [7] proposed an advanced update mechanism to overcome this issue, introducing a pre-defined parameter called waiting time, which has been widely adopted in subsequent studies [67, 70]. This parameter specifies the duration to wait after a code change is committed before classifying it as clean. Either when developers eventually detect a defect or after waiting a waiting time period to assume that a change is defect-free.

Verification Latency: While this approach allows for the labeling of new data, it is still influenced by the waiting time parameter, which often leads to delays in obtaining labels. This delayed availability of labels still postpones model updates and introduces a *verification latency* [13, 17, 35, 72] problem, potentially eroding the timeliness that is central to online approaches.

Human-assisted Update Mechanism: To address the verification latency issue, some studies have explored incorporating human-assisted update mechanisms to enable quicker labeling of code changes[70]. However, such manual efforts come with their own set of drawbacks. For instance, developers or reviewers might be occupied with other critical tasks, leading to delays in manual labeling. Moreover, the introduction of human intervention significantly increases labor costs and may result in inconsistent labels due to subjective judgments. These challenges highlight the need for further research into robust and efficient strategies that can both preserve the real-time advantages of online JIT-SDP and ensure timely, reliable label acquisition.

In summary, while offline defect prediction methods excel at providing a comprehensive, ranked overview of defect-proneness across a software system, online approaches are indispensable for their ability to integrate into fast-paced development environments and offer instantaneous, actionable insights at the moment of code change. Nevertheless, issues such as concept drift and verification latency—particularly at the code-change level—pose significant challenges. Effectively addressing these challenges remains crucial for advancing online defect prediction methodologies in modern software development workflows.

2.3 Software Testing

Unit Testing and Automated Test Case Generation. Unit testing, a form of class-level software testing, is one of the most widely adopted testing techniques in software development. It involves verifying the functionality of individual components or classes to ensure that each unit of the software performs as intended. Alongside unit testing, automated test case generation techniques have gained prominence [19, 37, 38, 40, 48, 51–53, 56, 75, 80, 83, 86]. These techniques aim to automatically produce test cases that can systematically explore the behavior of software components, thus reducing manual effort and increasing test coverage. Automated approaches often utilize static and dynamic analysis, model-based testing, and search-based techniques to generate test cases

Just Accepted

that can uncover hidden defects[19, 38, 51–53, 56, 86] or utilize large language model to generate test cases [9, 38, 40, 48, 75, 80, 83].

Test Case Selection, Prioritization, and Minimization in Code Change Scenarios. In the context of code change, numerous studies have focused on methods to reduce the testing overhead by selecting[25, 39], prioritizing[4, 10, 84], and minimizing test cases[12, 50, 59] from a large pool of available tests. Test case selection aims to identify a subset of test cases that are relevant to the recent code changes, thereby reducing the overall test execution time. Test case prioritization, on the other hand, reorders the test cases to execute those with a higher likelihood of detecting faults earlier. Test case minimization further refines this approach by eliminating redundant test cases while maintaining adequate coverage. Although these techniques are effective in scenarios where every commit is subjected to testing, they typically assume that a comprehensive set of existing test cases is available in the code repository.

Defect Prediction and Selective Testing at the Class Level[54, 55]. A parallel line of research has emerged that targets testing only a subset of classes, premised on the observation that the majority of software defects are concentrated in a small portion of the classes. These studies often integrate defect prediction techniques to identify classes that are more likely to be defective, and subsequently focus testing efforts on these high-risk classes. This approach allows for a more efficient allocation of testing resources by concentrating on the parts of the software that are statistically more prone to defects.

Challenges of Testing Based on Code Change Granularity. In practice, not every code change introduces defects, and therefore it is not always necessary to execute the entire test suite for every commit. The aforementioned test case selection, prioritization, and minimization techniques are designed under the assumption that every commit warrants testing or or selectively testing the code changes identified by developers as critical, differing mainly in the execution time for each commit. However, these techniques rely on the presence of a mature test suite, and many real-world repositories either lack adequate test cases or contain tests that do not cover the majority of functionalities. Additionally, certain commits introduce new features for which no corresponding test cases exist, rendering these techniques less effective.

While testing a subset of classes based on defect prediction can mitigate some testing overhead, this strategy is not universally applicable. Many code changes affect only a single class, thereby obviating the need for a class selection process. Moreover, even if only a subset of classes is tested, this still entails running tests on every commit, regardless of whether the change introduces defects.

Overall, there remains a notable gap in the research literature: no study has specifically addressed the problem of selectively testing only a subset of code changes under the premise that not every change is defect-inducing. More specifically, there is a lack of research on leveraging online defect detection techniques at the code change granularity to assist in determining which commits warrant testing. Addressing this gap could pave the way for more efficient testing strategies that align testing efforts with the actual risk of defect introduction in a continuous integration environment.

Just Accepted

3 Related Work

3.1 Software Defect Prediction for SQA

Software defect prediction is a widely adopted SQA activity aimed at predicting software code likely to contain defects. These predictions assist practitioners in effectively allocating and prioritizing limited quality assurance resources on the most risky software patches, leading to improved SQA [33, 78]. Extensive research in this area has primarily focused on defect prediction at the class or method level [47, 79].

However, in recent years, there has been a growing interest in studying defect prediction at the commit (code change) level, which offers finer granularity in defect prediction [5, 7, 32, 33, 69, 70, 74]. Kamei et al. conducted

a landmark study in 2013, developing an change-level defect prediction model based on 14 factors derived from software change characteristics [33]. Subsequent studies have widely adopted these factors [5, 32, 44, 69]. Additionally, Tan et al. highlighted the importance of considering the chronology of code changes in defect prediction during continuous code production and the need to update prediction models over time to adapt to incoming change data [74]. Failure to do so can result in a deterioration of predictive performance due to the models' inability to capture data changes [5–7, 43].

Recent studies have demonstrated the practicality and effectiveness of change-level defect prediction, particularly through online learning paradigms in the software development process [5, 7, 67, 69, 71]. Researchers have proposed online learning methods to tackle challenges posed by changing environments [5, 73, 74]. Song et al. [67] introduced an online human-assist update mechanism to address noise associated with the delayed labeling technique, demonstrating strong predictive performance. Building upon this work, they [70] developed the HumLa method and its variant, Eco-HumLa, which incorporate human inspection to facilitate more immediate model training. This approach enhances practicality and outperforms existing methods, being state-of-the-art (SOTA) change-level defect predictors. However, both HumLa and Eco-HumLa still rely on valuable human effort for immediately labeling, limiting the automation of the prediction system and requiring costly human resources. To address these limitations, we aim to leverage their partially automated techniques and incorporate automated testing techniques to fully automate the process. Our goal is to enhance the practical applicability of change-level defect prediction, positioning our method as a more advanced JIT-SQA framework.

3.2 Automated Software Testing for SQA

Automated software testing is another widely adopted SQA practice in the field. A notable example is `Sapienz` [41] , `TestGen`[2] and `TestGen-LLM`[1], which have been utilized to enhance the quality of Facebook [27] and Meta software, respectively.

Automated software testing, specifically Search-Based Software Testing (SBST), usually employs meta-heuristic search techniques to automatically generate test cases [51]. There have been plenty of research literature proposing algorithms for automated testing case generation [20, 30, 49, 51, 52, 56, 77]. Panichella et al. [51] proposed a multi-objective optimization algorithm for automated test case generation, which was further enhanced with DynaMOSA [52]. DynaMOSA dynamically select coverage targets based on the control dependency hierarchy.

A widely-used tool is `EvoSuite` [20], commonly used for automated testing in Java applications. `EvoSuite` integrates various test case generation algorithms and test oracle generation methods, including regression oracle generation [11, 49]. This integrated tool executes the generated test cases and captures the states (i.e., assertions) of the class under test. By running the same test cases after developers make code revisions, `EvoSuite` can detect assertion violations, thereby confirming the presence or absence of defects in the source codes.

3.3 Integration of Defect Prediction and Testing

Recently, researchers started to explore the application of defect prediction results to facilitate the testing process. For example, Perera et al. proposed approaches where additional time budget would be allocated for testing code classes that are predicted to be more likely defective by defect predictors [54, 55]. Subsequently, they further suggested using defect prediction information to determine code classes that require increased testing coverage [56, 57].

Traditional SDP techniques aim to identify defect-prone code components before testing, helping optimize testing efforts and conserve resources. However, most conventional approaches train prediction models on historical data and apply them statically, without adapting to the evolving characteristics of the software under test (SUT). To address these challenges, researchers have explored ways to integrate SDP with real-time testing feedback to continuously refine prediction models.

Just Accepted

Xiao et al. [82] proposed a *feedback-based integrated prediction (FIP)* method that embeds a feedback control mechanism within the software testing process (STP). The core idea of FIP is to use test results generated during STP as feedback to dynamically adjust the defect prediction model, thereby optimizing accuracy.

While FIP represents an innovative step toward integrating defect prediction with testing feedback, several limitations restrict its applicability in modern CI/CD scenarios, which require change-based, real-time analysis, listed as follows:

- **Class Imbalance:** FIP does not address the inherent class imbalance problem, which biases the model toward predicting changes as clean and makes defect-inducing changes difficult to detect.
- **Model Simplicity:** FIP employs a linear logistic regression model for defect prediction. However, defect-inducing changes often exhibit complex, nonlinear patterns that cannot be captured by such a simple model. Moreover, since FIP retrains the model on all samples after each testing cycle, more sophisticated models are impractical under this update mechanism.
- **Ranking Unsuitability:** FIP relies on ranking algorithms to prioritize modules that are more likely to contain defects. However, in practice, code changes occur sequentially during software development, and the complete set of changes is typically unavailable for ranking. This limitation renders the approach unsuitable for JIT scenarios.
- **Noisy Negative Labels:** FIP treats test outcomes that do not reveal defects as valid negative samples. Since testing may not uncover all defects, this assumption can introduce significant noise into the training data.

We make minor adaptations to FIP to enable its application in JIT-SQA scenarios. The details of these adjustments will be presented in the experimental section 7.2, where we also provide a comparative evaluation of FIP's applicability under JIT-SQA settings.

4 The Proposed AuDITee for JIT-SQA

4.1 The Overall Framework

The AuDITee framework consists of three essential modules: 1) a selector which is responsible for determining code changes to be sent to the automated testing module; 2) an automated code testing module, which identifies defects through rigorous testing despite higher computational costs, and 3) a defect predictor that is constructed using our novel WTT update mechanism for cheaper computational resources in predicting potential code defects. These modules operate continuously throughout the development process, following the JIT-SQA manner. AuDITee iteratively operates with the following steps, in line with those illustrated in Figure 1.

Step 1 Data features are generated for each individual code change based on Kamei et al.'s metrics [33]. These features are then fed into the defect prediction module integrated within the AuDITee framework.

Step 2 We collect both the code change features and the prediction probabilities produced by the defect prediction module. AuDITee determines whether testing is required solely based on the prediction probability, representing a straightforward strategy. In contrast, AuDITee-RL (see Section 4.4) makes dynamic decisions by considering both the code change features and the prediction probabilities, thereby enabling a more adaptive judgment of whether testing is necessary.

Step 3 If the selector supposes the code change need to be tested, the automated testing module is notified and utilized to conduct testing on the corresponding changed classes. When a defect is detected during testing, the corresponding code change is immediately labeled as a defect-inducing sample with a reward of 1, and can be directly used to update both the defect prediction model and the reinforcement learning model. Conversely, if no defect is detected, the features of the code change together with its submission timestamp are stored in the waiting time buffer, and the reward is set to −0.5. Conversely, if the selector determines that the code changes do not need to be tested, they are not forwarded to the testing module.

Just Accepted

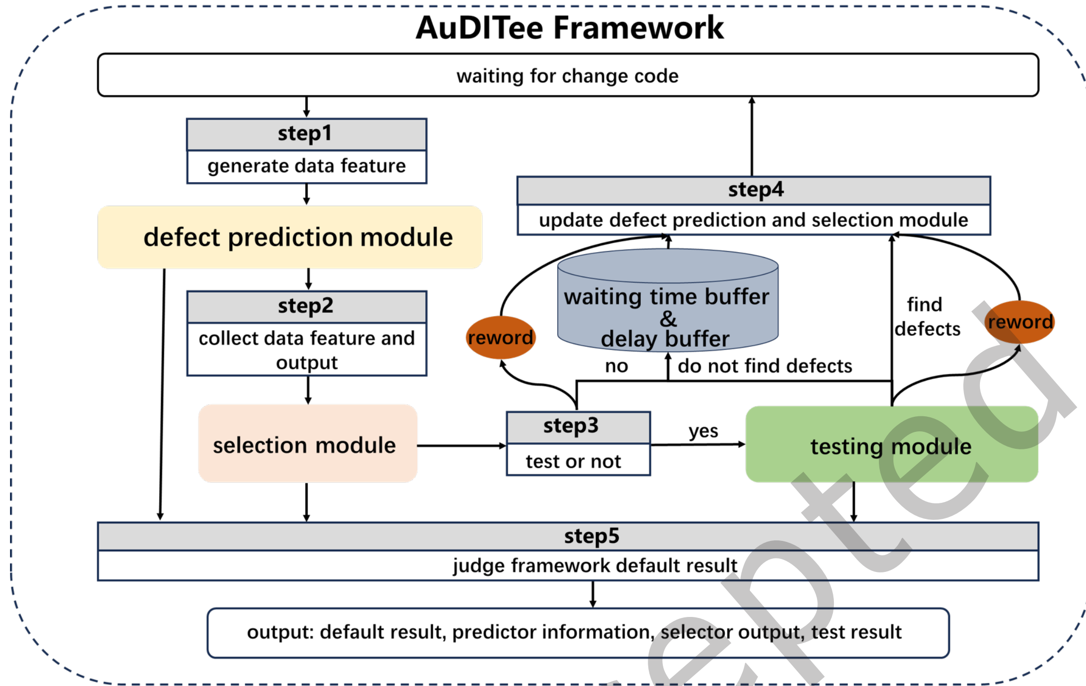

Fig. 1. The proposed AuDITee framework. Note Step 4 and Step 5 can be executed in parallel, for being independent of each other.

The features of each code change, together with its commit timestamp, are stored in the waiting time buffer and the delay buffer, with the reward for the sample initialized to 0. The waiting time buffer stores unlabeled data that will later be annotated via the waiting time mechanism and used to train the defect prediction model, whereas the delay buffer keeps untested data for which the correctness of the selector's decision cannot yet be assessed. In the latter case, the reward is temporarily set to 0 and will be corrected once the sample is labeled through the waiting time mechanism.

Step 4 Once testing is skipped or the automated testing module completes execution, AuDITee activates its update mechanism, which employs our proposed WTT update mechanism (see Section 4.3) to update the defect prediction model. For AuDITee-RL, the reinforcement learning model is also updated simultaneously based on the assigned rewards.

Step 5 The output of the framework comprises four elements: default result, predictor information, selector output, and test result. The default result is set to "1" when either the defect predictor or the testing module detects a defect; otherwise, it is set to "0". The predictor information includes both the output of the predictor and the probability that the current code change contains a defect. The selector output will be elaborated in Section 4.4. Finally, the test result reflects the output from the testing module, or it is set to "null" if no testing has been performed.

Through the iterative steps outlined above, AuDITee and AuDITee-RL achieve a complementary integration of defect prediction and software testing in JIT-SQA scenarios. They also address challenges such as concept drift and verification latency via our novel WTT update mechanism. In particular, samples for which testing does not reveal defects are not immediately used for model updates; instead, they undergo further verification

Just Accepted

through the waiting time strategy, which avoids the one-sided noise inherent in test outcomes and helps alleviate class imbalance to some extent. Moreover, AuDITee-RL continuously updates its reinforcement learning model, ensuring that testing can still be effectively guided even when the performance of the defect predictor becomes unstable. These claims will be empirically validated in the subsequent experiments.

Ideally, advanced testing algorithms and defect predictors operating at code change level can be encoded into this JIT-SQA framework, enabling improved JIT-SQA outcomes. In this study, we adopt `EvoSuite` [20] and TestGen-LLM [1] as our automatic code testing benchmark, and HumLa and ECo-HumLa [70] as our defect predictor benchmark, for them being SOTA techniques.

4.2 The Algorithm of the Overall Framework

AuDITee and AuDITee-RL is presented in Algorithm 1. AuDITee maintains a waiting time buffer to store all code change datas that have not been used to update the defect prediction model and a delay buffer to store all code change datas which rewords need further update (Line 3). When a new commit arrives, data features are immediately generated for the change based on Kamei et al.'s metrics (Line 5). After this step, the training data lists for both the predictor and the selector are initialized, which will be used to update them before the end of the current cycle. AuDITee performs model updates only after the predictor and selector have completed their decisions, thereby adhering to the test-then-train principle (Line 6-7). The defect prediction module first receives the feature of the code change and outputs two pieces of information (Line 8-9): the prediction result and the probability of having a defect. The output of the defect prediction module is recorded, and AuDITee initializes its default output with this prediction result (Line 10). Subsequently, if the framework is AuDITee-RL, a RL–based dynamic selector is employed to determine whether testing should be executed (Line 11-12). In contrast, AuDITee directly relies on the defect prediction result as the criterion for deciding whether testing is necessary (Line 13-14). If testing is not required, the reward is set to 0, and the information of the current code change is stored in the delay buffer, the selector's training data list, and the waiting time buffer (Line 17-19). If it should be tested, the testing module is called to test the classes involved in the current code change (Line 20-21). If no defect is detected by testing, the reward is set to −0.5 to penalize redundant testing behavior, and the information of the current code change is stored in the selector's training data list and the waiting time buffer (Line 22-24). If a defect is detected during testing, the default output of AuDITee is corrected to align with the testing result (Line 25-26), then the reward is set to 1 to reinforce effective testing behavior, and the information of the current code change is stored in both the selector's training data list and the predictor's training data list (Line 27-28). The framework will output four pieces of information (Line 31): the default result F (Line 10, 26), the predictor's information (Line 10), the selector's output (Line 12, 14), and the testing result (Line 21). Samples labeled by the waiting time mechanism in the waiting time buffer and delay buffer are transferred to the training data lists of the predictor and selector, respectively, for model updates at the end of the current cycle (see Algorithm 2, Line 32). If the framework is AuDITee-RL, the RL–based selector is updated (Line 33-34). If the predictor's training data list is not empty, the defect prediction model is updated (Line 36-37).

Algorithm 2 presents the update process for the waiting time buffer, the predictor's training data list, the delay buffer, and the selector's training data list. The overall data update strategy involves labeling samples in the waiting time buffer and adjusting the rewards for samples in the delay buffer. Specifically, the system iterates over all samples in the waiting time buffer (Line 1). If a defect is identified by developers for a given sample, it is assigned with "1" with a reward of −1 (Line 2-3), since the selector failed to choose the defective sample for testing and incurs penalty. If the elapsed time since the sample's commit exceeds the waiting time and the sample has not yet been labeled, it is assigned a label of "0" with a reward of 0 (Line 4-5). Notably, once a sample is labeled, it is always added to the predictor's training data list (Line 10). Samples labeled as "1" are removed from the waiting time buffer (Line 11, 12), while those labeled as "0" are retained, as the waiting time mechanism

Just Accepted

Algorithm 1 The proposed AuDITee framework

Require: pre-trained defect predictor $\mathcal{M}_0(\cdot)$, automatic testing module $\mathcal{T}(\cdot)$, reinforcement learning model $RL(\cdot)$ and waiting time Δt

Ensure: set $INFO$(default result F, predictor information M_{info}, selector output s, test result z)

1: $t \leftarrow 1$ record times of commit code change
2: $i \leftarrow 0$ record defect predictor $\mathcal{M}_i(\cdot)$ status
3: Initial waiting time buffer B and delay buffer DB to null: $B, DB = null$
4: **while** TRUE **do**
5: Waiting for new code change and generate data features X_t
6: Initial predictor's training data list PTB to null: $PTB = null$
7: Initial selector's training data list STB to null: $STB = null$
8: Initial the predictor's outputs (defective or not $\hat{y}_t$, defective probability dp) to null: $\hat{y}_t, dp = null$
9: Predict code change at commit t: $\hat{y}_t, dp \leftarrow \mathcal{M}_i(X_t)$
10: $M_{info} = (\hat{y}_t, dp), F = \hat{y}_t$
11: **if** use dynamic selector **then**
12: Select whether the code change need to test: $s = RL(X, dp)$
13: **else**
14: Select whether the code change need to test: $s = \hat{y}_t$
15: **end if**
16: $z = null$
17: **if** $s = 0$ **then**
18: set $reword = 0$, $DB.add(X_t, reword, t)$, $STB.add(X_t, reword, t)$
19: set $label = null$, B: $B.add(X_t, label, t)$
20: **else if** $s = 1$ **then**
21: Test the change code: $z \leftarrow \mathcal{T}(X_t)$
22: **if** $z = 0$ **then**
23: set $reword = -0.5$, $STB.add(X_t, reword, t)$
24: set $label = null$, B: $B.add(X_t, label, t)$
25: **else if** $z = 1$ **then**
26: fix default result with test result: $F = z$
27: set $reword = 1$, $STB.add(X_t, reword, t)$
28: set $label = 1$, to PTB: $PTB.add(X_t, label, t)$
29: **end if**
30: **end if**
31: yield $INFO(F, M_{info}, s, z)$
32: update B, PTB, DB and STB: UpdateData($B, PTB, DB, STB, t, \Delta t$)
33: **if** use dynamic selector **then**
34: update selector UpdateRL($RL(\cdot), STB$)
35: **end if**
36: **if** $PTB \neq null$ **then**
37: update defect predictor: $\mathcal{M}_{i+1} \leftarrow$ UpdateModel($\mathcal{M}_i, PTB$)
38: **end if**
39: $t \leftarrow t + 1$
40: **end while**

Just Accepted

assumes that defects may still be detected beyond the threshold. If a labeled sample also exists in the delay buffer, its reward is updated, and the sample is added to the selector's training data list before being removed from the delay buffer (Line 14-16).

Algorithm 2 Procedure UpdateData

Require: waiting time buffer B, predictor's training data list PTB, delay buffer DB, selector's training data list STB, commit time t and waiting time Δt

1: **for all** b in B **do**
2:   **if** developers find defects in b **then**
3:     set $label = 1, reword = -1$
4:   **else if** $t - b.t \geq \Delta t$ and $b.label = null$ **then**
5:     set $label = 0, reword = 0$
6:   **else**
7:     continue
8:   **end if**
9:   update b in B
10:   add b to PTB: $PTB.add(b, label)$
11:   **if** $label = 1$ **then**
12:     delete b from B: $B.delete(b)$
13:   **end if**
14:   **if** b in DB **then**
15:     add b to STB: $STB.add(b, reword)$
16:     delete b from DB: $DB.delete(b)$
17:   **end if**
18: **end for**

The subsequent subsections elaborate on the proposed three modules of the proposed framework.

4.3 The Defect Prediction Module

As illustrated in Figure 1, the defect prediction module aims to integrate the widely-used defect prediction techniques into our proposed AuDITee framework. In this study, we adopt HumLa as our benchmark defect prediction model due to its SOTA performance [70]. Although we employ the same underlying model as HumLa, the update mechanism differs: HumLa relies on manual guidance, whereas our framework leverages the proposed WTT update mechanism.

When a new code change X_t is committed at time step t, the defect predictor assesses whether the code change contains defect and outputs a two-tuple $(\hat{y}_t, dp = \mathcal{M}_i(X_t))$, where $\mathcal{M}_i(\cdot)$ represents the defect predictor at state i, $\hat{y}_t \in \{0, 1\}$ indicates whether the change is defective, $dp \in [0, 1]$ denotes the probability of it being defective. Our framework then uses selector to determine whether the code change needs to be tested. Indeed, the decision of these code changes can, in return, produce labeled training examples, which can subsequently be used to update the JIT-SQA system.

Figure 2 presents the WTT update mechanism of the defect prediction module. If the selector outputs "1" and a defect is detected by testing, the corresponding code changes are regarded as training examples for the JIT-SQA system. The original HumLa and ECo-HumLa workflows [70] require human inspection to label defect-predicted code changes, enabling the creation of corresponding training examples for immediate model updates. The integration of automated testing techniques enhances the automation level in HumLa and ECo-HumLa, allowing

Just Accepted

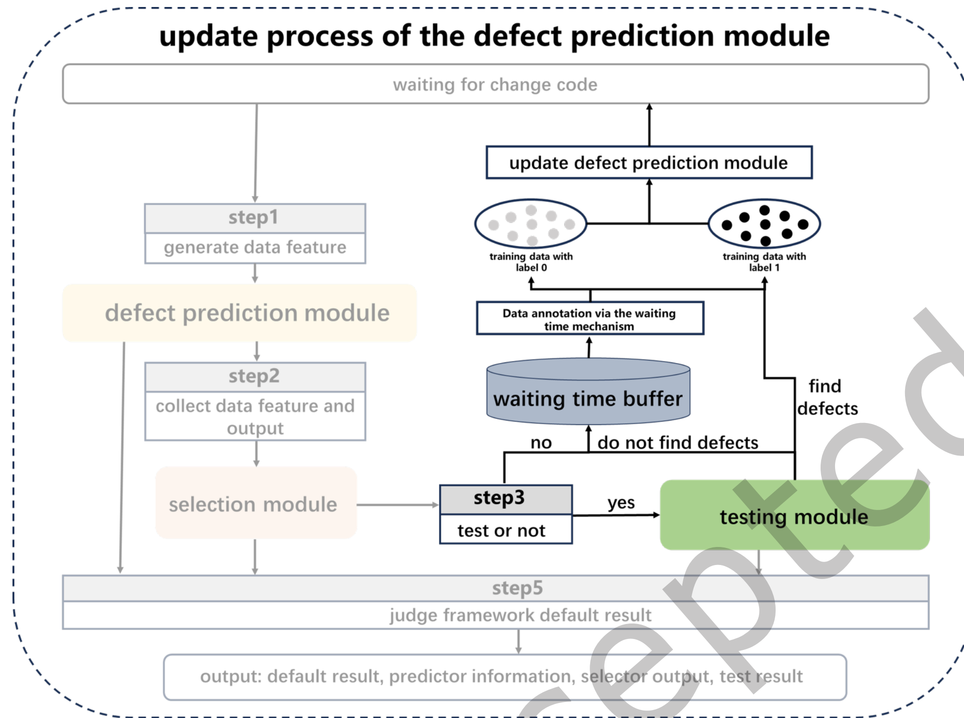

Fig. 2. Schematic illustration of the WTT update mechanism for the defect prediction module. Note that this figure only depicts the process of obtaining and updating training samples required by the defect predictor; the update procedure of the RL–based selector is presented in Section 4.4.

the JIT-SQA framework to operate fully automatically, which aligns with modern JIT-SQA practices. Conversely, when the selector outputs "0" or when testing reveals no defect, the labeling of these code changes is managed by the waiting time labeling procedure (see Algorithm 2).

This alternative process allows the AuDITee framework to confirm the final defect prediction results, facilitating the defect prediction workflow with automated testing outputs. Overall, the incorporation of automated testing techniques into the defect prediction workflow of HumLa and ECo-HumLa has the benefits of eliminating the need for human code inspections, which can be resource-intensive. This advancement moves SOTA methods towards full automation, effectively supporting JIT-SQA activities.

4.4 The Selection Module

As illustrated in Figure 1, the selection module is designed for assessing the probability of transmitting a code change to the subsequent testing module. A critical component of our AuDITee framework is the test selection mechanism, which decides whether a given code change should undergo automated testing based on its likelihood of containing defects. This mechanism serves as a bridge between defect prediction and test execution, significantly influencing both the cost-effectiveness and fault-detection capability of the quality assurance process.

In the original AuDITee framework, the selection process is performed via a fixed-threshold strategy: code changes with predicted defect scores exceeding a predefined threshold are selected for testing, while others are omitted. Although this approach is straightforward and efficient, it assumes that the defect prediction model

Just Accepted

maintains consistent performance over time. However, in practice, JIT defect prediction models are prone to temporal performance fluctuate. As a result, a static threshold may become suboptimal, leading to either excessive testing cost or missed defects.

To address this limitation, we propose an enhanced version of the framework, AuDITee-RL, which integrates a RL-based selector. This selector dynamically adapts its decision-making policy based on feedback from past testing outcomes, aiming to mitigate the impact of model performance fluctuate and achieve a better balance between testing efficiency and defect detection effectiveness.

4.4.1 Problem Statement. We formulate the decision of whether to test a code change as a Markov Decision Process (MDP). In this formulation, each incoming code change corresponds to an environment state, and the test selector acts as a decision-making agent that must choose between two discrete actions: to test the change or to skip it. This setting aligns well with the sequential nature of the software development process, where each testing decision can influence future quality assurance outcomes and resource usage.

This formulation is particularly well-suited for Reinforcement Learning, as it captures the exploration-exploitation trade-off inherent in the selection process. The agent must learn to identify valuable patterns in code changes that indicate potential defects while balancing the cost of unnecessary testing.

Crucially, the training signal for the agent comes from delayed feedback. When a selected code change is tested, its correctness (i.e., whether it triggers a test failure) provides a reward that updates the agent's policy. Conversely, when the agent decides to skip testing, the defect status of the change remains unknown unless revealed through external means, such as bug reports, which may be significantly delayed or unavailable. As a result, the feedback received by the agent is partially observable and varies in timing: some actions yield immediate and reliable rewards, while others result in unverifiable or delayed outcomes.

We refer to this setting as a *hybrid delayed feedback environment*, which poses unique challenges for standard RL algorithms. The agent must develop effective selection strategies under this uncertainty, leveraging positive reinforcement from observed test outcomes while avoiding overfitting to short-term signals or rare events.We outline how we address this challenge using a tailored RL design.

4.4.2 Core RL Algorithm. Among various RL algorithms, we adopt Deep Q-Network (DQN) as the core learning algorithm in AuDITee-RL for the following reasons.

First, the discrete nature of the action space (i.e., binary test decisions) aligns well with the design of DQN, which is tailored for decision-making in environments with finite action sets. Second, DQN is capable of handling high-dimensional and continuous state representations, allowing it to learn from rich feature vectors extracted from each code change (e.g., semantic, historical, and prediction-based features). Third, DQN supports experience replay, which is essential in our context where feedback signals are not immediately available for all actions. DQN's replay mechanism allows us to accumulate and reuse verified experiences, mitigating the impact of delayed or missing feedback.

Overall, DQN provides a balance between expressive learning capacity and efficient decision-making, making it well-suited for training a test selector under real-world constraints such as noisy predictions, partial observability, and fluctuating defect detection performance.

4.4.3 State Space Design. To enable the selector agent to make informed testing decisions, we design a rich and informative state space for each code change. Specifically, each state is represented as a feature vector that combines two sources of information: (1) the features proposed by Kamei et al. [33], which capture structural, semantic, and historical characteristics of the code change, and (2) the defect probability estimated by the underlying defect prediction model. These features provide a holistic view of the change's defect-proneness and contextual information, which are essential for the agent to learn effective testing strategies.

Just Accepted

The Kamei features have been widely validated in just-in-time defect prediction literature and offer insights into the complexity and risk of a code change. The predicted defect probability further complements these handcrafted features by summarizing learned patterns from historical data. Together, the 14-dimensional state representation allows the DQN agent to generalize across diverse code changes and adapt its behavior based on both interpretable metrics and model-driven signals.

4.4.4 Action Space Design. The action space is defined as a binary decision for each code change: 1 indicates that the change will be tested, and 0 means it will not. This discrete setup aligns with the practical objective of minimizing unnecessary testing while ensuring defect detection. It also simplifies the learning process for the agent by reducing the decision complexity.

4.4.5 Reward Function. The reward function is designed to reflect both the effectiveness and cost of testing, while also accommodating the delayed nature of feedback when tests are skipped. Specifically, the reward function is designed as follows:

- If a code change is tested:
  - A reward of +1 is given if a defect is detected.
  - A penalty of −0.5 is applied if no defect is found, representing the cost of unnecessary testing.
- If a code change is not tested:
  - A delayed reward of 0 is temporarily assigned.
  - The corresponding state is stored in the delay buffer for future update based on external feedback.

We periodically update the delay buffer based on the waiting time mechanism. Specifically, for each code change i in the buffer:

- If waiting time mechanism later confirms that the untested change i as defective, a reward of -1 is given and we move the data from the delay buffer to the agent's replay memory.
- If the elapsed time since the data's commit exceeds the waiting time the reward is settled to 0 and the data is similarly moved for training.

This mechanism introduces a hybrid reward structure, where tested changes receive immediate rewards, and untested ones receive delayed, externally triggered feedback, allowing the agent to learn from both observed outcomes and long-term consequences.

4.4.6 Learning Algorithm. The selector agent is trained using the standard DQN learning procedure. After sufficient experience is accumulated in the replay memory buffer, the agent begins learning by sampling mini-batches of transitions $(s_t, a_t, r_t, s_{t+1}, \text{done})$, where each transition is either generated from immediate feedback (for tested changes) or updated delayed feedback (for not tested changes).

Each mini-batch is used to update the Q-network via stochastic gradient descent. Given a batch of transitions, we compute the predicted Q-values for the taken actions:

$$Q(s_t, a_t) = \texttt{q_net}(s_t)[a_t] \tag{1}$$

and the target Q-values using the target network:

$$Q_{\text{target}} = r_t + \gamma \cdot \max_{a'} \texttt{target_net}(s_{t+1})[a'] \cdot (1 - \text{done}) \tag{2}$$

where γ is the discount factor, and done indicates whether the episode has ended. When delayed rewards are used, the episode is still ongoing (done = false). In contrast, when immediate rewards are used or when samples are drawn from the updated delay buffer, the episode has already terminated (done = true).

The Q-network parameters are updated by minimizing the mean squared error (MSE) between the predicted Q-values and the target values:

$$\mathcal{L} = \text{MSE}(Q(s_t, a_t), Q_{\text{target}}) \tag{3}$$

Just Accepted

The network parameters are updated using backpropagation, and the target network is synchronized with the current Q-network every fixed number of steps to improve training stability:

$$\texttt{target_net} \leftarrow \texttt{q_net} \quad \text{(every } N \text{ steps)} \tag{4}$$

This learning loop continues throughout the training phase, allowing the agent to refine its test selection policy based on both immediate and delayed feedback.

4.5 The Automated Testing Module

In this study, we select two test generation frameworks – EvoSuite [20], based on SBST-based techniques, and TestGen-LLM [1], based on large language models (LLMs) – to demonstrate that AuDITee can seamlessly integrate with different testing technologies.

Based on them, our testing module, denoted as $\mathcal{T}(\cdot)$, generates testing cases for changed classes, which are then executed to determine the defect status of the subsequent code changes. When the chosen selector outputs "1", suggesting the code change X_t need to be tested, our framework proceed with `EvoSuite` or TestGen-LLM to further explore and confirm whether there are defects. During the execution of the generated testing cases, if the result is "1" ($[z_t = \mathcal{T}(X_t)] = 1$), indicating the presence of defects through the testing module, the result is used as the label for the sample and is immediately employed to update the defect prediction model. Then our proposed framework outputs a defective result. If the test case execution yields a result of 0 ($[z_t = \mathcal{T}(X_t)] = 0$), we do not immediately use the current sample to train the defect prediction model, because a test that does not reveal a defect does not necessarily indicate that no defect was introduced. Instead, such samples are stored in the waiting time buffer, and their final labels are determined by the waiting time mechanism. This two-stage verification – combining testing with waiting time labeling – reduces noise compared to replying solely testing or only waiting time labeling.

Moreover, excluding samples without detected defects from model updates can alleviate class imbalance. We provide a formal argument as follows:

Goal: Compare S_1, the proportion of defect samples when training the predictor using all test results, and S_2, the proportion when training using only detected-defect samples.

Let the sample distribution before time t_0 be m defect-inducing samples and n clean samples. Let $t_1 \geq t_0$, and consider the samples stored in the waiting time buffer from t_0 to $t_1 - 1$, which starting being annotated at t_1.

For $t \in [t_0, t_1 - 1]$, if testing is conducted (otherwise the imbalance ratio remains unchanged), let the test result be $tr_t \in \{0, 1\}$. Then:

$$S_1 = \frac{m + \sum tr_i}{m + n + t - t_0}, \quad S_2 = \frac{m + \sum tr_i}{m + n + \sum tr_i}. \tag{5}$$

Since $\sum tr_i \leq t - t_0$, we have $S_2 > S_1$.

For $t \geq t_1$, let k_0 samples be labeled as 0 and k_1 samples as 1 during testing in $[t_0, t_1 - 1]$. Then at time $t_1 - 1$:

$$S_1 = \frac{m + k_1}{m + n + k_0 + k_1}, \quad S_2 = \frac{m + k_1}{m + n + k_1}. \tag{6}$$

Let $x = t_1 - t_0 - k_0 - k_1$ denote the remaining samples in the waiting time buffer. At time t, let x_1 samples be labeled as 1 and x_0 samples as 0 by the waiting time mechanism. Then:

$$S_1 = \frac{m + k_1 + x_1}{m + n + k_0 + k_1 + x_1 + x_0}. \tag{7}$$

For S_2, the number of samples in the waiting time buffer from $[t_0, t_1 - 1]$ is $x + k_0$. At time t, let $x_1 + w_1$ samples be labeled as 1 and $x_0 + w_0$ as 0, where x_1, x_0 come from x and w_1, w_0 come from k_0. Then:

Just Accepted

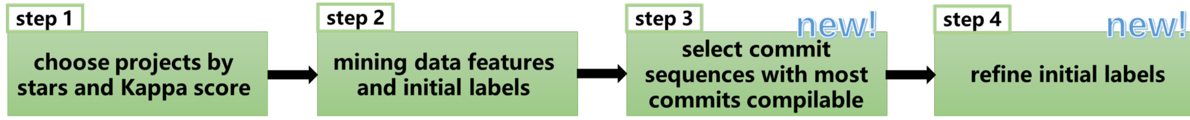

Fig. 3. Our data processing workflow.

$$S_2 = \frac{m + k_1 + x_1 + w_1}{m + n + k_1 + w_0 + w_1 + x_0 + x_1}. \tag{8}$$

Since $w_0 + w_1 \leq k_0$ and $m + k_1 + x_1 + w_1 \geq m + k_1 + x_1$, we have $S_2 \geq S_1$.

When the chosen selector outputs "0", suggesting the code change X_t do not need to be tested, our proposed framework concludes that there is no need to perform the testing procedure and outputs $\hat{y}_t$. We can see that the incorporated strategy here is to adopt a SOTA defect predictor and a selector to facilitate the judgment of performing the testing. Testing such code changes, which are likely to be defect-free, would be a waste of valuable testing resources, considering that the majority of code changes do not contain defects. In such cases, the proposed AuDITee outputs the predictor result.

This alternative process allows the AuDITee framework to produce final defect decisions, facilitating the automated testing workflow with defect predictions provided by advanced defect predictors. Overall, the incorporation of SOTA defect predictors into the testing workflow has the strong benefits of reducing testing resources, which would otherwise costed.

5 Dataset Preparation

This experiment necessitates compiling the projects and generating test cases, requirements that were absent in prior studies. Consequently, our data processing workflow involves two additional post-processing steps, as illustrated in Figure 3.

step 1 Initially, we considered six long-term maintained Java and Python projects—JGroups, BroadleafCommerce, Tomcat, Camel, Fabric8 and Django—for our analysis, as these projects are commonly involved in just-in-time (JIT) code change scenarios for software quality assurance [67, 69, 70]. However, we excluded Camel for two main reasons: 1) Compared to other project, Camel has more complex dependencies, often leading to dependency conflicts and missing dependencies, which makes large-scale experimental validation difficult. 2) Camel indeed has a low Kappa value and the Kappa value between the two reviewers is also low reported in [70], indicating that the labels in this dataset are highly ambiguous. Ultimately, we selected five high-profile open source projects—JGroups (1k stars), BroadleafCommerce (1.7k stars), Tomcat (7.4k stars), Fabric8 (1.8k stars) and Django (85.1k stars)—all of which have maintained a development history spanning over eight years.

step 2 To collect the necessary data—including input features and initial labels for code changes—we employed the `Commit Guru` tool [64]. This tool leverages the SZZ algorithm [66] to identify defect-inducing code changes, thereby refining our experimental dataset to the five selected projects. In addition, we collected supplementary information such as commit IDs and preceding commit IDs for each code change, which is essential for accurately tracing the evolution of the codebase.

step 3 Accurate defect analysis and testing require that the modified code compiles successfully. However, several factors can prevent compilation: (1) The developer's commit may inherently contain errors; (2) Dependency management tools like Maven may introduce issues, as repository changes or deprecations over time can affect code compatibility; and (3) Network fluctuations may cause failures in downloading dependencies

Just Accepted

Table 1. An overview of the software projects investigated in this work.

Dataset	Language	Total Changes	Defective%	Compilable Testing%	Dependency Tool	Time Period	Test Generation Tool
JGroups	Java	6568	24.37	90.65	Ant/Maven	08/2009 - 05/2025	Evosuite
Broadleaf	Java	9427	27.40	89.72	Maven	05/2013 - 05/2024	Evosuite
Tomcat	Java	24607	35.91	78.21	Ant	12/2006 - 03/2024	Evosuite/TestGen-LLM
Fabric8	Java	1409	27.89	98.43	Maven	10/2015 - 09/2018	Evosuite
Django	Python	1000	9.30	80.40	pip	06/2023 - 09/2024	TestGen-LLM

from remote repositories. To mitigate these issues, we carefully selected sequential fragments in which the majority of commits were compilable, aligning with the continuous SQA approach that emphasizes sequential commit analysis. Moreover, each commit was compiled ten times to further reduce the impact of transient failures and ensure reliable results. It is worth noting that, in real-world development, repositories typically yield a much higher rate of compilable commits than those observed in our experimental setup.

step 4 During data processing, we observed that `Commit Guru` occasionally misclassified changes that only affected testing code as defect-inducing. Since updates to testing code are generally routine and should not be treated as defects, we manually corrected these labels to reduce noise in our dataset. We recommend that future studies employing `Commit Guru` differentiate between changes in functional code and those in testing code to ensure more accurate defect labeling. Additionally, we tested each commit and corrected the labels assigned by Commit Guru based on the test results. Specifically, if the tests detected a defect, we labeled the corresponding commit as “1”.

Table 1 summarizes the datasets used in this study, including JGroups, BroadleafCommerce, Tomcat, and Fabric8 are Java-based projects. To improve data diversity and to validate the applicability of the AuDITee framework across multiple programming languages, we additionally include the Python-based Django project. Django achieves the highest Kappa score in Python projects [70], and provides a rich testing repository. We use this project to evaluate the effectiveness of TestGen-LLM, as this tool requires the presence of existing test cases in the repository—a condition not satisfied by other Python projects in prior JIT-SQA studies. Another project using TestGen-LLM is Tomcat. For the first 23,607 samples, test cases were generated with EvoSuite. However, the subsequent samples require compilation with JDK 21 or higher, while EvoSuite currently only supports generating test cases for projects using JDK 11 or below. Therefore, we used TestGen-LLM to generate test cases for the last 1,000 samples. In total, 2,000 code changes have test cases generated by TestGen-LLM. Compared with prior evaluations of TestGen-LLM, which used 1,979 samples, our dataset is sufficient to validate the effectiveness of integrating TestGen-LLM into the AuDITee framework. Fabric8 has not been actively maintained since 2018, and the last sample we mined corresponds to the latest commit in the repository. Considering that this project has been widely used in prior studies, we retained it in our dataset. However, the repository relied on the FuseSource Maven repository before October 19, 2015, which is no longer maintained or accessible. As a result, the compilable portion of the project that we collected starts from October 19, 2015, totaling 1,409 samples.

We adopt an online learning approach to train and evaluate the model, as it better aligns with the temporal nature of code changes. Consequently, we do not partition the dataset into separate training, validation, and test sets. Instead, the model update procedure is constrained to follow a test-then-train paradigm: new samples are first used to evaluate the model and then to update it. As described in Section 4.1, our update workflow satisfies this constraint.

When using TestGen-LLM to generate test cases, both projects initially only have 1,000 samples, which is insufficient for the defect prediction model to learn effectively. To address this, we additionally include 4,000 samples with real labels to simulate testing behavior, and we pretrain the defect predictor using the proposed WTT update mechanism. Ultimately, model performance is evaluated only on the samples for which TestGen-LLM is actually used.

Just Accepted

6 Evaluation Metrics

6.1 Metrics for SDP

We utilize five metrics to evaluate the defect prediction module and the overall framework's predictive performance (using default result, see Section 4.1). These metrics, namely Precision, Recall, F1-score, Geometric Mean (G-Mean), and Matthews Correlation Coefficient (MCC), are selected to provide a comprehensive assessment of the model's ability to predict defects at the code change level. Each metric offers valuable insights into different aspects of performance, including accuracy, balance, and reliability of defect predictions. The Recall of the defect predictor, i.e., the proportion of correctly identified buggy code, has a significant impact on the defect detection effectiveness of testing with a large effect size[58]. Given the specific nature of the defect prediction task, where detecting as many true defects as possible is crucial, **Recall** is particularly emphasized.

The following definitions are used to define performance metrics of defect prediction:

- *True Positives (TP)*: The number of defective code changes correctly identified as defective.
- *False Positives (FP)*: The number of non-defective code changes incorrectly identified as defective.
- *True Negatives (TN)*: The number of non-defective code changes correctly identified as non-defective.
- *False Negatives (FN)*: The number of defective code changes incorrectly identified as non-defective.

Table 2 summarizes the formulas used to compute the metrics for SDP.

Table 2. Metrics for SDP

metrics	formulas
$Precision_{sdp}$	$\frac{TP}{TP+FP}$
$Recall$	$\frac{TP}{TP+FN}$
$F1$	$2 \times \frac{Precision \times Recall}{Precision+Recall}$
G-Mean	$\sqrt{\frac{TN}{TN+FP} \times \frac{TP}{TP+FN}}$
MCC	$\frac{TP \times TN - FP \times FN}{\sqrt{(TP+FP)(TP+FN)(TN+FP)(TN+FN)}}$

Table 3. Metrics for testing

metrics	formulas
FDR	$\frac{\text{Detected Defective Code Changes}}{\text{Total Defective Code Changes}}$
$Precision_{testing}$	$\frac{\text{Tests that Failed}}{\text{Total Tests}}$
RTR	$\frac{\text{Tested Code Changes with No Defects}}{\text{Total Code Changes Under Test}}$
Test Count	Number of Tested Code Changes

Just Accepted

6.2 Evaluation Metrics for Testing Performance

In this study, we evaluate the performance of the testing module and the framework overall (using default result, see Section 4.1) test performance using three key metrics: False Discovery Rate (FDR), $Precision_{testing}$, and Redundant Test Rate (RTR) [1]. These metrics are designed to assess both the effectiveness and efficiency of the testing process, with particular emphasis on how well the tests can identify defects and how resources are utilized. Additionally, test count is employed to quantify the overall testing cost

Table 3 summarizes the formulas used to compute the metrics for testing.

We list our analysis and insights of these metrics as follows:

- FDR remains the central metric for evaluating the effectiveness of the testing process. A high FDR ensures that a large proportion of actual defects are detected, which is critical for maintaining software quality. However, FDR should not be evaluated in isolation. If FDR is high, but $Precision_{testing}$ is also low, it might

[1]This is a new metric, designed in this paper to especially evaluate the testing performance.

indicate that the testing process is focusing on code changes that are generally defect-free, leading to inefficient resource use.

- $Precision_{testing}$ provides an indication of how many tests fail, but on its own, it does not directly reflect the efficiency or focus of the testing process. If the $Precision_{testing}$ is low, it may simply reflect that the selected code changes are less likely to contain defects, rather than indicating high-quality testing. Therefore, it must be interpreted in the context of FDR and RTR to assess the effectiveness of the testing.
- RTR (a newly introduced metric) helps assess the resource utilization of the testing process. A high RTR means many tests are not finding defects, suggesting that testing resources are not being used effectively. By reducing RTR, the testing process can be optimized to focus on defect-prone areas, maximizing the detection of defects while minimizing wasted effort.
- Test Count serves as a key metric for evaluating the resource consumption of the testing process. A high Test Count indicates that many code changes are subjected to testing, which may ensure broader coverage but also implies greater testing effort and cost. Conversely, a low Test Count suggests a more selective testing strategy that potentially reduces resource usage.

The combination of FDR, $Precision_{testing}$, and RTR provides a balanced approach to evaluate the effectiveness and efficiency of the testing process. While FDR directly measures defect detection, $Precision_{testing}$ and RTR offer insights into the testing strategy's focus and the efficiency of resource usage. By optimizing these metrics, the testing process can be made both effective in detecting defects and efficient in utilizing testing resources.

Therefore, by optimizing the Test Count, our framework is able to significantly reduce the consumption of testing resources.

7 Experimental Setup

7.1 Experimental Details

The experiments were conducted on a system equipped with an Intel(R) Xeon(R) Platinum 9242 CPU. When using TestGen-LLM, we set the temperature parameter of the LLM to 0 to ensure that the generated test cases are deterministic and reproducible. Since EvoSuite and the defect prediction modules involve inherent randomness, we used different random seeds for each experiment and repeated the experiments ten times. This approach ensures that the results are stable and reliable despite any inherent randomness in the modules.

We measured the average time required for compiling, generating, and executing test cases for each sample. On average, EvoSuite required 128 seconds to generate test cases, while TestGen-LLM required 51 seconds. Since all comparison methods in the experiments needed to execute tests, we adopted concurrent processing and cached the test results for each code change to enable reuse, thereby accelerating the experiments. Consequently, when using EvoSuite, we generated tests ten times for each sample, whereas with TestGen-LLM, due to the temperature being set to 0, we generated tests only once for each sample. We did not account for the prediction and update time of the defect prediction module, as it typically ranged from 1 to 10 minutes depending on the number of code changes in a project, which is negligible compared to the test execution time (e.g., testing 1,000 samples with EvoSuite takes more than 2,000 minutes). In total, all experiments consumed $128\ seconds \times (6,568 + 9,427 + 23,607 + 1,409) \times 10 + 51\ seconds \times (1,000 + 1,000) = 14,610$ CPU hours of testing time.

To align with real-world software development processes, we arranged the experimental data in strict chronological order. Unlike offline approaches that randomly split the data into training, validation, and test sets, our design follows the test-then-train paradigm of online learning: at each step, a model is first evaluated on the incoming code changes and then updated using the same data. This setup better reflects realistic conditions, where code changes and defects emerge sequentially over time.

Just Accepted

7.2 Experimental Design for Defect Prediction

To implement defect prediction within AuDITee, we employ advanced approaches, specifically HumLa and ECo-HumLa, as our defect prediction backbone, as discussed in Section 4.3. For their parameter configurations within AuDITee, we have opted to use the default parameter values employed in HumLa [70]. Other advanced online defect predictors suitable for the JIT-SQA scenario can also easily integrated into our AuDITee framework, provided that they demonstrated strong performance.

We experimentally evaluate the efficacy of defect prediction by comparing the proposed JIT-SQA framework against three benchmarks. Specifically, we consider two SOTA defect prediction models with different update mechanisms that are applicable to the JIT-SQA setting. Moreover, we adapt an existing software quality assurance framework, FIP [82], which also integrates defect prediction and testing, to ensure its compatibility with the JIT-SQA scenario. The adapted FIP is then included as an additional baseline in our evaluation.

- *ODaSC:* ODaSC is an automated defect predictor tailored for JIT-SQA, facilitating early defect predictions [67]. It employs Hoeffding trees [16] to create and update defect predictors only by waiting time mechanism dynamically.
- *HumLa:* HumLa and its alteration ECo-HumLa employ a human-assisted update mechanism to update the ODaSC model, resulting in SOTA defect predictors that require human inspection of code changes predicted as defective upon their commitments [70]. This strategy promotes the timely updating of defect predictors in response to evolving environments, significantly enhancing predictive performance in early defect predictions.
- *FIP:* FIP [82] employs two logistic regression models—a global model and a local model. The models incorporate feedback from testing results, and their predictions are combined using dynamically adjusted weights. This design addresses the challenge in cross-project settings, where models often fail to fit the target project effectively during the initial phase. Since the source code of FIP was unavailable and the method could not be directly applied to our task, we re-implemented and adapted it. Specifically, whereas the original FIP ranks modules under test, we modified the framework to predict whether a given code change is defective.

7.3 Experimental Design for Testing

To operate the testing process on code changes, we employ the widely recognized `Evosuite` [20, 22] and TestGen-LLM [1] as our benchmark testing workflow (see Section 4.5). `Evosuite` is a widely used SBST tool that incorporates various advanced testing strategies, including MOSA [51] and DynaMOSA [52]. Additionally, `Evosuite` offers the capability to integrate oracle generation methods, such as regression oracle generation [49], enabling automated testing within our JIT-SQA framework. In our study, we specifically choose DynaMOSA, which won the testing tool competition at SBST'2019 [8]. This techniques implemented in `Evosuite` are compatible with our AuDITee framework, and the effectiveness of AuDITee relies on the quality of the opted testing algorithms.

In addition to comparing individual testing techniques, we also evaluate approaches that integrate SOTA defect prediction (i.e., ODaSC,ECo-HumLa and HumLa) to guide testing (i.e., DynaMOSA and TestGen-LLM). All the comparison methods are summarized as follows.

- *DynaMOSA:* DynaMOSA is the most widely used automated testing approach [52]. It employs a many-objective sorting algorithm to enhances both testing effectiveness and efficiency by treating testing objectives as multiple optimization criteria. This approach generates test cases that comprehensively cover the objectives minimizing their size.
- *TestGen-LLM:* TestGen-LLM [1] is a SOTA LLM-based test case generation technique that has been applied and validated in real industrial settings. It is designed to augment existing test suites, making it particularly

Just Accepted

suitable for scenarios involving code changes. Since the authors of TestGen-LLM did not provide the source code, we used an unofficial implementation [2], which has received 5.2k stars.

- *ODaSC4Test:* ODaSC4Test is a straightforward integration of ODaSC and testing, in which the prediction results from ODaSC are directly used to determine whether testing is necessary. Its main difference from AuDITee is that it updates the defect prediction model solely using a waiting time mechanism. The testing modules are validated separately using DynaMOSA and TestGen-LLM.
- *HumLa4Test:* HumLa4Test is a simple integration of HumLa and testing. Compared to AuDITee, its main difference lies in using a semi-automated update mechanism. HumLa has two parameters: human effort and human error. In RQ4, we will evaluate the equivalent human cost of AuDITee-RL in terms of average human effort and human noise, and use this cost as fixed parameters for HumLa4Test.
- *ECo-HumLa4Test:* ECo-HumLa4Test is a simple integration of ECo-HumLa and testing. ECo-HumLa is a variant of HumLa with a single parameter, human noise, which will also be determined in RQ4.

Both TestGen-LLM and DynaMOSA require parameter setting. For TestGen-LLM, we primarily adopted the default parameter settings provided in the implementation. In addition, we used GPT-4o-mini to generate test cases, with the temperature parameter set to 0. For DynaMOSA, however, tuning these parameters in SBST can be time-consuming and costly [3, 56]. Arcuri and Fraser [3] demonstrated that the default parameter values in `EvoSuite` yielded results comparable to those achieved with meticulously tuned parameters. Similarly, Panichella et al. [52] achieved strong performance in DynaMOSA by using these default settings. Therefore, we opt to use the default parameter settings from `EvoSuite` [20] and DynaMOSA [52], with the exception of the following parameters:

a) Termination criteria: We set a maximum time budget of two minutes for testing case generation per class, halting the search progress once all branches are covered. Developers can adjust the time allocated based on project size and available resources of their organization.

b) Test suite minimization: We disable test suite minimization, allowing `Evosuite` to retain all test cases that cover potential defect targets. This strategy, aligned with the choice in [56], increases the number of final test cases, enhancing the likelihood of finding defects.

c) Assertion strategy: The oracle problem, which concerns determining whether a test passes or fails, can be addressed using various approaches, including regression oracles [49], mutation testing-based oracles [20, 21], deep learning-based oracles [42, 77, 81], and LLM-based oracles[15, 28]. Since our framework aims to identify early defects, it is important that AuDITee finds defects as soon as they are introduced. This means that the previous code change should not contain defects; if they are present, our framework should have detected them previously, preventing their discovery in subsequent tests. Thus, we use the result of the previous commit as a testing oracle for the current commit. commonly referred to as a regression oracle [11, 20, 38, 49].

d) Testing strategy: To ensure an efficient testing process, AuDITee tests each class within a code change randomly. When a defective class is identified, testing is halted, and the code change is marked as defective. Some code changes may involve revisions to multiple classes. In our research on JIT-SQA scenarios, we consider two testing environments: one with abundant computing resources for full testing (the *full* testing scenario), and another with limited testing resources,restricting parallel testing to a subset of the code (the *partial* testing scenario). The datasets are divided accordingly, with JGroups designated for full testing and Broadleaf and Tomcat for partial testing. For the latter, a 20-minute budget is allocated, allowing random testing of up to 10 classes.

e) Flakiness Suppression Mechanisms: Flaky tests refer to tests that produce inconsistent results without any changes being made to the system under test. These tests pose a significant problem for software development, affecting testing efficiency, continuous integration, and developer productivity. Evosuite has a built-in flaky test suppression mechanism, which can significantly reduce the number of flaky tests[24].

[2]https://github.com/qodo-ai/qodo-cover.git

Just Accepted

8 Experimental Results and Analyses

In this section, we present the results of our experiments designed to assess the performance of the proposed framework for defect prediction. The results are analyzed across several metrics to demonstrate the framework's effectiveness in real-world scenarios.

8.1 RQ1: To what extent can the proposed WTT update strategy (i.e., AuDITee and AuDITee-RL) – and the RL–based selector module (i.e., AuDITee-RL) mitigate the challenges posed by code change scenarios, which often degrade the performance of simple integrations of defect prediction and testing?

8.1.1 Motivation. In code change scenarios, data are inevitably subject to concept drift, validation latency, and class imbalance. This experiment aims to investigate whether the novel WTT update mechanism can effectively alleviate the overall performance degradation caused by concept drift and validation latency, as well as mitigate the class imbalance issue.

We adopt the default output defined in Section 4.1 as the standard output for each integrated framework. In simple integration approaches—and even in AuDITee—the output fully relies on the defect prediction module, essentially propagating its results directly. In contrast, the RL–based selector module enables the framework to jointly consider both the defect prediction and testing modules when producing the output. This experiment further examines the extent to which such integration can improve the overall performance of the framework.

8.1.2 Results. We answer RQ1 by analyzing the results presented in Table 4. Since RQ1 involves comparing multiple methods, we mitigated randomness in the frameworks by running each experiment ten times with different random seeds. To determine whether there were statistically significant performance differences among the methods across these repeated runs, we applied Friedman tests [17]. The null hypothesis states that there are no statistically significant difference, and the alternative hypothesis states that they have significant difference. When the null hypothesis was rejected at the significant level $\alpha = 0.05$, we conducted pairwise Conover post-hoc tests with Bonferroni correction to control for multiple comparisons. In addition, we reported effect sizes using Cliff's Delta (δ), which quantifies the degree of difference between two methods. Following Romano et al. [63], δ values were interpreted as negligible (< 0.147), small (< 0.33), medium (< 0.474), and large (≥ 0.474).

Table 4 reports the results in terms of Recall, Precision, F1, G-Mean, MCC, and FDR. Although FDR is commonly used as a metric in testing, it can also serve as an indicator of the framework's ability to discover defects. Among these metrics, Recall, Precision, F1, and FDR primarily emphasize performance comparison, whereas G-Mean and MCC are more concerned with addressing the issue of class imbalance. In the following, we analyze the results of each metric individually.

- With respect to *Recall*:
  - Across all projects, AuDITee consistently outperforms other simple integration frameworks, with the improvements being statistically significant and exhibiting large effect sizes in most cases. This superiority holds regardless of the test case generation technique with which the frameworks are integrated. In particular, the comparison with ODaSC4Test (0.6319 v.s. 0.5505) highlights a qualitative leap brought by the WTT update mechanism. Moreover, the comparisons with HumLa4Test and Eco-HumLa4Test demonstrate that AuDITee, under fully automated conditions, can achieve performance comparable to or even better than semi-automated frameworks with high-quality human interventions, thereby reducing human effort and cost. At the same time, we observe that FIP is almost ineffective in detecting defects across projects and completely fails to identify any defects when integrated with TestGen-LLM (i.e., Django and on the last 1,000 samples of Tomcat).

Just Accepted

Table 4. Performance comparison between AuDITee, AuDITee-RL, and several simple integration frameworks that combine SOTA defect prediction and testing. For fairness, all frameworks integrate TestGen-LLM on Django and the last 1,000 samples of Tomcat, while using DynaMOSA for the other projects, ensuring that the results are not affected by differences in testing techniques. A “*” indicates that the corresponding framework shows a statistically significant difference compared with AuDITee-RL, while a “†” denotes significance compared with AuDITee. After applying the Bonferroni correction, we computed Cliff’s Delta effect sizes. Symbols [n], [s], [m], and [l] denote negligible, small, medium, and large Cliff’s Delta effect sizes compared with AuDITee-RL, respectively, while {n}, {s}, {m}, and {l} denote the corresponding effect sizes compared with AuDITee. The last six rows summarize the average values of each metric across different integration frameworks, where higher values indicate better performance.

metrics		Recall	$Precision_{sdp}$	F1	G-Mean	MCC	FDR
JGroups	ODaSC4Test	0.3481*[l]†{l}	0.6861*[l]†{l}	0.4508*[l]†{l}	0.5281*[l]†{l}	0.2155*[l]†{l}	0.3690*[l]†{l}
	HumLa4Test	0.4354*[l]{s}	0.6754*[l]{s}	0.5170*[l]{s}	0.5736*[l]{s}	0.2449*[l]{s}	0.4630[m]{n}
	Eco-HumLa4Test	0.3942*[l]†{l}	0.6876[l]†{l}	0.4920*[l]†{l}	0.5575*[l]†{l}	0.2340*[l]{s}	0.4094*[l]†{l}
	FIP	0.0078*[l]†{l}	0.8667[l]†{l}	0.0161*[l]†{l}	0.0614*[l]†{l}	0.0337*[l]†{l}	0.0072*[l]†{l}
	AuDITee	0.4515[m]	0.6685*[l]	0.5289*[l]	0.5793*[l]	0.2390*[l]	0.4734[m]
	AuDITee-RL	0.4838	0.7077	0.5668	0.6130	0.2980	0.5039
Broadleaf	ODaSC4Test	0.2793*[l]†{l}	0.6869*[l]†{l}	0.3882*[l]†{l}	0.4842*[l]†{l}	0.1864*[l]†{l}	0.2895*[l]†{l}
	HumLa4Test	0.4275[l]{m}	0.6758*[l]{s}	0.5137*[l]{m}	0.5714*[l]{m}	0.2345*[l]{m}	0.4384[l]{l}
	Eco-HumLa4Test	0.4008*[l]{l}	0.6805*[l]{m}	0.4958*[l]{s}	0.5607*[l]{n}	0.2291*[l]{n}	0.4108*[l]{m}
	FIP	0.0073*[l]†{l}	0.8832[l]†{l}	0.0141*[l]†{l}	0.0570*[l]†{l}	0.0287*[l]†{l}	0.0069*[l]†{l}
	AuDITee	0.4153*[l]	0.6742*[l]	0.5036*[l]	0.5638*[l]	0.2272*[l]	0.4242*[l]
	AuDITee-RL	0.4766	0.7062	0.5621	0.6093	0.2892	0.4838
Tomcat	ODaSC4Test	0.5721*[l]†{l}	0.6777[l]†{l}	0.6140*[l]†{l}	0.6364*[l]†{l}	0.3019*[l]†{l}	0.5698*[l]†{l}
	HumLa4Test	0.6960*[l]{s}	0.6584*[l]{s}	0.6699*[l]{n}	0.6543*[l]{n}	0.3364*[l]{n}	0.6988*[l]{n}
	Eco-HumLa4Test	0.6601*[l]†{l}	0.6635*[l]{m}	0.6566*[l]†{l}	0.6539*[l]{n}	0.3247*[l]{l}	0.6621*[l]†{l}
	FIP	0.0935*[l]†{l}	0.7527[m]†{m}	0.1089*[l]†{l}	0.1726*[l]†{l}	0.0927*[l]†{l}	0.0677*[l]†{l}
	AuDITee	0.6909*[l]	0.6602*[l]	0.6686*[l]	0.6547*[l]	0.3351*[l]	0.6961*[l]
	AuDITee-RL	0.7566	0.6826	0.7141	0.6915	0.4049	0.7609
Fabric8	ODaSC4Test	0.6626*[l]{n}	0.7251*[l]{s}	0.6894*[l]{n}	0.6985*[l]{n}	0.4099*[l]{n}	0.6499*[l]{n}
	HumLa4Test	0.6526*[l]{n}	0.7282[l]{n}	0.6854*[l]{n}	0.6968*[l]{n}	0.4088*[l]{n}	0.6522[l]{n}
	Eco-HumLa4Test	0.6588*[l]{n}	0.7214*[l]{m}	0.6858*[l]{m}	0.6947*[l]{s}	0.4020*[l]{s}	0.6547[l]{n}
	FIP	0.0075*[l]†{l}	0.8786[l]†{l}	0.0295*[l]†{l}	0.0535*[l]†{l}	0.0293*[l]†{l}	0.0097*[l]†{l}
	AuDITee	0.6583*[l]{n}	0.7299*[m]	0.6887*[l]	0.6986*[l]	0.4124*[l]	0.6478[l]
	AuDITee-RL	0.7167	0.7400	0.7250	0.7260	0.4623	0.7028
Django	ODaSC4Test	0.7564*[l]†{l}	0.7011*[l]{s}	0.7330*[l]†{l}	0.7134[s]{l}	0.4444*[l]†{l}	0.7441*[l]†{l}
	HumLa4Test	0.8459[l]{s}	0.6775[s]†{l}	0.7587*[l]{m}	0.7098[l]†{l}	0.4688[l]†{l}	0.8387[m]{s}
	Eco-HumLa4Test	0.8131*[l]{s}	0.6973*[l]{n}	0.7565*[l]{m}	0.7228[l]{n}	0.4763[s]{l}	0.7978*[l]{s}
	FIP	-	-	-	-	-	-
	AuDITee	0.8284*[l]	0.6968[l]	0.7628[s]	0.7256[l]	0.4869[l]	0.8226*[l]
	AuDITee-RL	0.8564	0.6813	0.7655	0.7162	0.4832	0.8495
Tomcat (last 1000 simples)	ODaSC4Test	0.6843*[l]†{l}	0.6990[l]†{l}	0.6849[l]{l}	0.6836[n]†{l}	0.3911[n]†{l}	0.6268*[l]†{l}
	HumLa4Test	0.7216[n]{l}	0.6705*[s]{l}	0.6904[m]{n}	0.6752[m]{l}	0.3732[m]{l}	0.6817[n]†{l}
	Eco-HumLa4Test	0.7071[s]†{l}	0.6656*[l]{l}	0.6810[l]{l}	0.6674*[l]{n}	0.3566*[l]{n}	0.6665[l]†{l}
	FIP	-	-	-	-	-	-
	AuDITee	0.7471[l]	0.6503*[l]	0.6934[s]	0.6657*[l]	0.3544*[l]	0.7220*[l]
	AuDITee-RL	0.7262	0.6805	0.6981	0.6844	0.3905	0.6787
avg	ODaSC4Test	0.5505	0.6960	0.5934	0.6240	0.3249	0.5415
	HumLa4Test	0.6298	0.6810	0.6392	0.6469	0.3444	0.6288
	Eco-HumLa4Test	0.6057	0.6860	0.6280	0.6428	0.3371	0.6002
	FIP	0.0290	0.8453	0.0422	0.0861	0.0461	0.0229
	AuDITee	0.6319	0.6800	0.6410	0.6480	0.3425	0.6310
	AuDITee-RL	0.6694	0.6997	0.6719	0.6734	0.3880	0.6633

Just Accepted

  - AuDITee-RL achieves substantially greater improvements, significantly outperforming all other methods across nearly all projects with large effect sizes. It performs slightly worse than AuDITee only on the last 1,000 samples of Tomcat, but the difference is not statistically significant. When integrated with DynaMOSA, AuDITee-RL consistently shows significant improvements over HumLa4Test and Eco-HumLa4Test across all projects. When integrated with TestGen-LLM, although its advantage on the last 1,000 samples of Tomcat is not statistically significant, it still demonstrates an overall noticeable improvement, particularly in comparison with Eco-HumLa4Test.
  - On average, the performance differences among the frameworks are clearly pronounced, with large effect sizes observed across almost all projects. AuDITee-RL achieves a recall of 0.6694, which is 5.9% higher than the second-best method, AuDITee, and far surpasses ODaSC4Test, which uses only the waiting time mechanism, by 21.6%. For HumLa4Test and Eco-HumLa4Test, we configured their parameters according to RQ3 so that their defect prediction modules would perform comparably to that of AuDITee-RL. Even under this setup, AuDITee-RL still exhibits a clear advantage, whereas AuDITee only achieves marginal improvements. This highlights the importance of the dynamic selection mechanism. Meanwhile, FIP achieves a recall of only 0.0290, demonstrating that it is almost entirely unable to detect defects across projects.
- With respect to $Precision_{sdp}$:
  - In contrast to recall, AuDITee shows a decrease in $Precision_{sdp}$. Compared with ODaSC4Test, $Precision_{sdp}$ is significantly lower in most projects, except for Fabric8 and the last 1,000 samples of Tomcat; however, the differences in each project are relatively small. Meanwhile, the differences between AuDITee and HumLa4Test or Eco-HumLa4Test are even smaller across projects and are often not statistically significant.
  - Compared with AuDITee, AuDITee-RL not only avoids a decrease in $Precision_{sdp}$ but even achieves significant improvements over other methods, showing statistically significant superiority across all projects. It is worth noting that FIP exhibits a clear advantage in $Precision_{sdp}$, significantly outperforming other methods in every project; however, its recall is extremely low, limiting its practical effectiveness.
  - On average, the frameworks exhibit clearly pronounced differences, with large effect sizes observed across all projects except Django. When excluding FIP, AuDITee-RL achieves the best overall performance. Although its advantage over ODaSC4Test is marginal—only 0.5%—considering the substantial improvements in recall achieved by AuDITee-RL and the fact that other simple integration methods, including AuDITee, perform worse than ODaSC4Test, this represents a significant breakthrough.
- With respect to *F1 Score*:
  - Compared with other simple integration frameworks, AuDITee still exhibits significant advantages with large effect sizes, except when compared with HumLa4Test. Specifically, regardless of whether DynaMOSA or TestGen-LLM is integrated, AuDITee outperforms other frameworks, with the exception of Fabric8 and the last 1,000 samples of Tomcat, where its superiority is not statistically significant. Its average F1 score is 0.6410, substantially higher than ODaSC4Test at 0.5934, representing an improvement of over 8%. Compared with HumLa4Test, AuDITee still shows a marginal advantage, with an average improvement of 0.3%, and the improvement on Django is statistically significant.
  - AuDITee-RL achieves improvements regardless of the testing approach integrated, and significantly outperforms all other simple integration methods on JGroups, Broadleaf, Tomcat, Fabric8, and Django. The largest improvement is observed on Broadleaf; when excluding FIP, AuDITee-RL outperforms the lowest-performing method, ODaSC4Test, by as much as 44.8%.

Just Accepted

  - On average, the frameworks exhibit clearly pronounced differences, with large effect sizes observed across all projects except the last 1,000 samples of Tomcat. AuDITee-RL consistently achieves the best results, with an average F1 score of 0.6719, representing an improvement of over 4.8% compared to the second-best method, AuDITee. FIP still fails to achieve satisfactory performance; although it attains very high $Precision_{sdp}$, its recall is extremely low. Notably, AuDITee and HumLa4Test exhibit comparable results (0.6410 vs. 0.6392), and for several metrics across multiple projects, the differences between these two frameworks are not statistically significant. Considering that HumLa4Test relies on high-quality human interventions, this suggests that AuDITee is already capable of replacing such manual effort. Furthermore, AuDITee-RL outperforms HumLa4Test, demonstrating that the dynamic selection mechanism substantially enhances the framework's overall performance.
- With respect to *FDR*:
  - FDR provides a more direct indication of a framework's defect detection capability. In terms of FDR, regardless of whether DynaMOSA or TestGen-LLM is integrated, AuDITee exhibits results consistent with other metrics, significantly outperforming ODaSC4Test and Eco-HumLa4Test across multiple projects. Compared with HumLa4Test, AuDITee shows no statistically significant differences on most samples.
  - Compared with ODaSC4Test, Eco-HumLa4Test, and HumLa4Test, AuDITee-RL also demonstrates superior defect detection capability. Except for the last 1,000 samples of Tomcat, where AuDITee-RL achieves an FDR of 0.6787 compared to HumLa4Test's 0.6817, the difference is not statistically significant. Overall, AuDITee-RL maintains a substantial advantage, with an average FDR of 0.6633, compared to 0.6288 for HumLa4Test.
  - On average, the frameworks exhibit clearly pronounced differences, with large effect sizes observed in most cases across all projects. Overall, the results are consistent with the F1-based conclusions. AuDITee-RL outperforms AuDITee, while AuDITee and HumLa4Test exhibit comparable performance, both surpassing ODaSC4Test and Eco-HumLa4Test. FIP achieves an average FDR of only 0.0229, further confirming its inability to effectively detect defects.
- With respect to *G-Mean*:
  - AuDITee achieves G-Mean values comparable to those of HumLa4Test and Eco-HumLa4Test (AuDITee: 0.6480, HumLa4Test: 0.6469, Eco-HumLa4Test: 0.6428), indicating that the WTT update mechanism can mitigate class imbalance nearly as effectively—or even more effectively—than the semi-automated update mechanisms that incorporate human feedback. Furthermore, regardless of whether DynaMOSA or TestGen-LLM is integrated, AuDITee significantly outperforms ODaSC4Test across all projects, except for Fabric8, Django and the last 1,000 samples of Tomcat. Although it falls 2.6% below ODaSC4Test on the last subset, it achieves an overall average improvement of 3.8%, further confirming the effectiveness of the proposed update mechanism in handling class imbalance.
  - AuDITee-RL also clearly outperforms all other simple integration methods in terms of G-Mean, achieving an improvement of over 7.9% compared to the lowest-performing method, ODaSC4Test.
  - On average, the frameworks exhibit clearly pronounced differences, with large effect sizes observed in most cases across all projects. AuDITee-RL achieves G-Mean values that are significantly higher than those of AuDITee across multiple projects, indicating that the RL–based dynamic selection mechanism can also partially mitigate class imbalance. This is likely because the dynamic selection mechanism tends to select more defective samples for testing, thereby providing the defect prediction module with additional minority-class samples. In contrast, FIP achieves a G-Mean of only 0.0861; combined with the previous metrics, this demonstrates its inability to handle class imbalance scenarios, indirectly highlighting the importance and practical significance of our framework.

Just Accepted

- With respect to *MCC*:
  - AuDITee achieves MCC results similar to its G-Mean performance, attaining values comparable to HumLa4Test and Eco-HumLa4Test, while significantly outperforming ODaSC4Test in most projects. On average, AuDITee improves by over 5.4% compared to ODaSC4Test.
  - Compared with AuDITee, AuDITee-RL demonstrates even more pronounced improvements over other simple integration methods. Specifically, it significantly outperforms ODaSC4Test, with an average improvement of over 19.4%, significantly outperforms HumLa4Test by approximately 12.7% on average, and significantly outperforms Eco-HumLa4Test by approximately 15.1% on average. These results further indicate that AuDITee-RL has superior capability in handling class imbalance, even surpassing results achieved with human intervention.
  - On average, the frameworks exhibit clearly pronounced differences, with large effect sizes observed in most cases across all projects. AuDITee-RL also significantly outperforms AuDITee in terms of MCC across most projects, with an average improvement of 13.3%. This further demonstrates that the dynamic selection mechanism is highly effective in mitigating class imbalance. At the same time, the significant improvements of AuDITee over ODaSC4Test further confirm the effectiveness of the proposed WTT update mechanism in handling class imbalance.

It is worth noting that AuDITee shows no statistically significant difference from HumLa4Test across almost all projects and metrics, indicating that the proposed WTT update mechanism achieves a level of overall performance comparable to the semi-automated update mechanism that integrates high-quality human knowledge.

Answer RQ1

Regardless of whether DynaMOSA or TestGen-LLM is integrated, the WTT update mechanism can significantly mitigate performance degradation caused by concept drift and validation delay, demonstrating effectiveness comparable to semi-automated update mechanisms that involves high-quality human intervention. The RL–based dynamic selector further enhances the overall performance of the framework by appropriately selecting outputs from both the defect prediction and testing modules. Additionally, the proposed update mechanism partially alleviates class imbalance, achieving performance level similar to human-in-the-loop approaches. Moreover, by providing additional defective samples to the defect prediction module, the dynamic selector can substantially reduce class imbalance even further.

8.2 RQ2: Compared to the simple integration of defect prediction and testing, can our framework (i.e., AuDITee and AuDITee-RL) detect a larger number of unique defects?

8.2.1 Motivation. Certain defects are inherently more challenging to expose than others. The ability of a technique to uncover these unique defects is recognized a crucial indicator of its practical effectiveness [26, 56]. Therefore, RQ2 investigates whether our proposed framework (i.e., AuDITee and AuDITee-RL) can identify significantly more unique defects compared with approaches that merely integrate defect prediction with testing.

8.2.2 Results. To address RQ2, we conducted pairwise comparisons among all frameworks. To mitigate the effects of randomness, each experiment was repeated ten times using different random seeds. Statistical significance of performance differences across repeated runs was assessed using two-tailed pairwise Wilcoxon signed-rank tests [14]. The null hypothesis states that there are no statistically significant difference, and the alternative hypothesis states that they have significant difference. Furthermore, we also reported Cliff's Delta to quantify the magnitude of observed differences. As a non-parametric effect size measure, we interpreted Cliff's Delta according to the same conventional guidelines [63] as RQ1: $0.147 < negligible$, $0.33 < small$, $0.474 < medium$, and $0.474 \leq large$.

Just Accepted

These effect sizes complement statistical significance testing by providing insights into the practical significance of the differences between methods.

Table 5. Comparison of AuDITee and AuDITee-RL against other frameworks that simply integrate SOTA defect prediction with testing in terms of their ability to discover unique defects. For fairness, all frameworks adopt TestGen-LLM on Django and the last 1,000 samples of Tomcat, while DynaMOSA is used for the other projects, ensuring that differences in testing techniques do not bias the results. A "*" indicates a statistically significant difference compared with AuDITee-RL, while "†" denotes significance compared with AuDITee. Symbols [n], [s], [m], and [l] represent negligible, small, medium, and large effect sizes compared with AuDITee-RL, whereas {n}, {s}, {m}, and {l} represent the corresponding levels compared with AuDITee. The last column reports the average number of unique defects detected across all datasets.

Dataset	JGroups	Broadleaf	Tomcat	Fabric8	Django	Tomcat (last 1000 simples)	avg
ODaSC4Test	45.2†{l}	32.7†{l}	171.3†{l}	19 {s}	1.3†{l}	1.1†{l}	53.9
AuDITee	212.3	388.4	1271	18.2	8.6	16.7	379.7
HumLa4Test	90 {s}	150.6 [l]	333.9 {s}	20 {s}	2.6 [n]	1.6†{l}	119.42
AuDITee	106.7	113.1	310.7	18.3	1.1	8.2	109.98
Eco-HumLa4Test	60.2†{l}	125.8 {s}	188.9†{l}	20.2 {s}	0.7†{l}	1.5†{l}	79.16
AuDITee	162.6	161.3	485.5	17.5	3	10.6	165.98
FIP	5.5†{l}	4.8†{l}	12.7†{l}	1.8†{l}	0†{l}	0†{l}	4.96
AuDITee	751.9	1107	1035.8	252.6	76.5	118.4	644.76
ODaSC4Test	40.5*[l]	27.9*[l]	144*[l]	19.8*[l]	0.7*[l]	2*[l]	46.58
AuDITee-RL	257.9	542.1	1827.3	40.6	10.5	11.1	535.68
HumLa4Test	100.1*[l]	144.1*[l]	431.1*[l]	17.4*[l]	0.6 [l]	5.8 [s]	138.66
AuDITee-RL	171.7	272.6	1014.6	37.3	1.6	6.5	299.56
Eco-HumLa4Test	76.9*[l]	124.4*[l]	302.8*[l]	13.7*[l]	0*[l]	4.8 [m]	103.56
AuDITee-RL	230.9	324.6	1197.6	32.6	4.8	7.4	358.1
FIP	3.1*[l]	4.6*[l]	17.1*[l]	1.1*[l]	0*[l]	0*[l]	5.18
AuDITee-RL	798.4	1264.3	1042.7	273.5	79	111.3	691.58
AuDITee	112.5 [l]	99.5*[l]	354.6*[l]	15.4*[l]	0.7*[l]	9.4*[l]	116.54
AuDITee-RL	167.5	264.6	953	37	3.2	3.8	285.06

The upper part of Table 5 presents the differences in the number of unique defects discovered by AuDITee compared with ODaSC4Test, HumLa4Test, Eco-HumLa4Test, and FIP. The lower part shows the differences between these integration frameworks (including AuDITee and AuDITee-RL). Our analysis reveals that, except for HumLa4Test, which shows no significant difference in most projects and exhibits comparable capability in discovering unique defects, AuDITee demonstrates statistically significant improvements over nearly all other simple integration frameworks. Moreover, the effect sizes are large in most projects. Specifically, AuDITee discovers more than seven times the number of unique defects as ODaSC4Test, more than twice that of Eco-HumLa4Test, and approximately 130 times that of FIP, highlighting its substantial superiority in uncovering unique defects.

Similarly, AuDITee-RL exhibits even stronger performance than AuDITee. It discovers more than twice the number of unique defects compared with AuDITee and significantly outperforms other integration methods in nearly all projects, with large effect sizes in most cases. These results indicate that AuDITee-RL is particularly effective at revealing defects that are difficult for other methods to detect.

Just Accepted

Answer RQ2

Compared to frameworks that merely integrate defect prediction with testing, AuDITee is capable of discover a larger number of unique defects. Even when compared with the semi-automated framework HumLa4Test, which benefits from high-quality human guidance, AuDITee exhibits comparable capability. Notably, AuDITee-RL outperforms all other methods, discovering more than twice the number of unique defects as both AuDITee and HumLa4Test. These results indicate that AuDITee and AuDITee-RL are particularly adept at identifying defects that are challenging for other frameworks to detect, further corroborating their superior performance as observed in RQ1.

8.3 RQ3: Compared with conventional defect prediction modules, how effectively can defect predictors enhanced with the proposed update mechanism (i.e., AuDITee and AuDITee-RL) reduce testing resource consumption? Furthermore, can the RL–based dynamic selector, which intelligently determines when to trigger testing (i.e., AuDITee-RL), further optimize resource utilization and decrease testing costs? Are the results sensitive to the choice of underlying testing frameworks?

8.3.1 Motivation. In RQ1 and RQ2, we analyzed and compared the overall performance of AuDITee and its variant AuDITee-RL, and demonstrated that our proposed framework—particularly AuDITee-RL—significantly outperforms simple integration methods which combine defect prediction models and testing algorithms. However, beyond improving effectiveness, our framework is also designed with another essential goal in mind: reducing testing costs. In particular, we aim to verify regardless of the type of testing method used, the proposed approach can substantially lower the testing effort by reducing the number of tests executed.

8.3.2 Results. We address RQ3 based on the results presented in Table 6 and Table 7. The former reports the comparison when the frameworks integrate DynaMOSA, while the latter presents the results for those using TestGen-LLM. Since RQ3 involves multiple methods, we followed the same statistical analysis procedure as in RQ1, including the Friedman test, Conover post-hoc tests with Bonferroni correction, and effect size estimation using Cliff's Delta (δ).

Table 6 reports the comparison results of AuDITee and AuDITee-RL integrated with DynaMOSA across four projects (JGroups, Broadleaf, Tomcat, and Fabric8) against frameworks that simply combine DynaMOSA with various SOTA defect prediction models. For reference, we also include the results of standalone DynaMOSA: without the guidance of a defect predictor, DynaMOSA needs to test every code change, leading to a higher Test Count. Notably, AuDITee-RL shows statistically significant differences compared to almost all other frameworks, with large effect sizes in most cases across all projects.

Overall, FIP yields the smallest number of test executions; however, its total average Test Count is only 232.3. This is clearly undesirable, as more than 20% of the samples in the current four projects are defective, and such a low number of test executions severely limits the ability to detect faults. Excluding FIP, AuDITee-RL achieves the lowest Test Count, with a total of 12,495.8 executions, which is significantly fewer than almost all other frameworks across projects. Considering its overall superior performance in RQ1, this demonstrates that AuDITee-RL not only improves prediction accuracy but also reduces testing resource consumption. Compared to DynaMOSA, the Test Count is reduced by more than 69.5% under the guidance of the proposed WTT update mechanism.

AuDITee also demonstrates strong resource-saving capability, comparable to HumLa4Test and ECo-HumLa4Test. Although it executes 29.1% more tests than ODaSC4Test, this trade-off is acceptable given that its defect detection performance substantially outperforms ODaSC4Test (as shown in RQ1). Relative to standalone DynaMOSA, AuDITee still saves considerable resources, reducing the total Test Count by nearly 60.0%. These results confirm

Just Accepted

Table 6. Comparison of the Test Count when the testing module in AuDITee and AuDITee-RL is implemented using DynaMOSA. The results are contrasted against the baseline of standalone DynaMOSA and simple integration frameworks that combine SOTA defect predictors with DynaMOSA. A “*” indicates a statistically significant difference compared with AuDITee-RL, while a “†” indicates significance compared with AuDITee. After applying the Bonferroni correction, we computed Cliff’s Delta effect sizes. Symbols [n], [s], [m], and [l] denote negligible, small, medium, and large Cliff’s Delta effect sizes compared with AuDITee-RL, respectively, while {n}, {s}, {m}, and {l} denote the corresponding effect sizes compared with AuDITee. The last column summarizes the total Test Count across all projects for each framework.

method	Test Count				
	JGroups	Broadleaf	Tomcat	Fabric8	total
DynaMOSA	6561*[l]†{l}	9428*[l]†{l}	23608*[l]†{l}	1409*[l]†{l}	41006
ODaSC4Test	1388.6[m]†{l}	1630.5*[l]†{l}	9179.4*[l]†{l}	507.9*[l]{n}	12706.4
HumLa4Test	1763.1*[l]{m}	2599.5[s]{m}	11595.3*[l]{s}	508*[l]{n}	16465.9
Eco-HumLa4Test	1577.7*[l]†{l}	2404.7*[l]{l}	10875*[l]†{l}	522.9*[l]{s}	15380.3
FIP	17.4*[l]†{l}	31.3*[l]†{l}	176.9*[l]†{l}	6.7[l]†{l}	232.3
AuDITee	1887.9*[l]	2534[m]	11487.2*[l]	500.9*[l]	16410
AuDITee-RL	1269.7	2662	8166	398.1	12495.8

that when integrated with DynaMOSA, the proposed WTT update mechanism significantly reduces the number of test executions, and the RL–based dynamic selector (AuDITee-RL) further enhances efficiency, achieving both lower resource consumption and strong fault detection capability.

Table 7. Comparison of the Test Count when the testing module in AuDITee and AuDITee-RL is implemented using TestGen-LLM. The results are contrasted against the baseline of standalone TestGen-LLM and simple integration frameworks that combine SOTA defect predictors with TestGen-LLM. A “*” indicates a statistically significant difference compared with AuDITee-RL, while a “†” indicates significance compared with AuDITee. After applying the Bonferroni correction, we computed Cliff’s Delta effect sizes. Symbols [n], [s], [m], and [l] denote negligible, small, medium, and large Cliff’s Delta effect sizes compared with AuDITee-RL, respectively, while {n}, {s}, {m}, and {l} denote the corresponding effect sizes compared with AuDITee. The last column summarizes the total Test Count across all projects for each framework.

method	Test Count		
	Django	Tomcat (last 1000 simples)	total
TestGen-LLM	1000*[l]†{l}	1000*[l]†{l}	2000
ODaSC4Test	349.6*[l]†{l}	339.7*[l]†{l}	689.3
HumLa4Test	428.4*[l]{l}	402.5*[l]†{l}	830.9
Eco-HumLa4Test	378.9*[l]{s}	398.8*[l]†{l}	777.7
FIP	-	-	-
AuDITee	392.2*[l]	452.8*[l]	845
AuDITee-RL	248.1	257.2	505.3

Table 7 presents the comparison results of AuDITee and AuDITee-RL integrated with TestGen-LLM on two projects (Django and the last 1000 samples of Tomcat), against frameworks that simply combine TestGen-LLM with different SOTA defect prediction models. We also report the results of standalone TestGen-LLM as a baseline: without guidance from a defect predictor, TestGen-LLM must execute tests for all code changes. For these two

Just Accepted

projects, the defect prediction module in FIP failed completely to predict defective samples, and therefore could not provide any guidance for testing.

Excluding FIP, the results are consistent with those observed when integrating DynaMOSA. Specifically, AuDITee-RL achieved the lowest total number of test executions (505.3), which was significantly better than all other frameworks, reducing the Test Count by more than 74.7% compared with standalone TestGen-LLM. Similarly, AuDITee exhibited Test Count comparable to HumLa4Test. Although AuDITee executed significantly more tests than ECo-HumLa4Test and HumLa4Test on the last 1000 samples of Tomcat, the overall difference was relatively small, with ECo-HumLa4Test requiring only about 8% fewer tests and HumLa4Test requiring only about 1.7% fewer tests. Nevertheless, AuDITee still reduced the number of executions by nearly 57.8% compared to TestGen-LLM.

These results confirm that, when integrated with TestGen-LLM, the proposed WTT update mechanism and the RL–based dynamic selector can also substantially reduce testing overhead, while maintaining strong defect detection capability.

Answer RQ3

The defect prediction models trained with the proposed WTT update mechanism effectively guide test selection. Although these models incur slightly higher Test Count than those trained with waiting time alone, this additional testing is both acceptable and meaningful, given the substantial overall performance gains of AuDITee and AuDITee-RL demonstrated in RQ1. Moreover, AuDITee exhibits comparable capability to models updated with high-quality manual feedback, highlighting the strength of our automated mechanism. The RL–based dynamic selector further improves efficiency by prioritizing fewer but more fault-prone samples for testing, thereby mitigating the issue of inaccurate test guidance caused by fluctuations in defect prediction performance. Importantly, the consistent superiority of our framework when integrated with both SBST-based techniques (i.e., DynaMOSA) and LLM-based techniques (i.e., TestGen-LLM) demonstrates that its advantages are robust and not dependent on the choice of testing tool.

8.4 RQ4: To what extent can the proposed update mechanism (i.e., AuDITee and AuDITee-RL) mitigate the negative effects of concept drift and verification latency on prediction performance? How effective can it address the challenge of class imbalance? Furthermore, to what extent can the RL–based dynamic selector (i.e., AuDITee-RL) improve the performance of the defect prediction model by selecting high-quality samples?

8.4.1 Motivation. The motivation for RQ4 stems from the observation that, when evaluating the overall performance of the framework, only AuDITee-RL can simultaneously consider both defect prediction and testing outputs. While RQ1 has demonstrated the benefits of jointly considering defect prediction and testing outputs, it does not yet show whether the novel WTT update mechanism or the RL–based dynamic selector can further enhance the performance of the defect prediction model and alleviate issues caused by class imbalance.

Furthermore, we aim to investigate how the update mechanisms of AuDITee and AuDITee-RL compare to human-in-the-loop update strategies, i.e., update mechanisms of HumLa and ECo-HumLa. Specifically, the defect prediction modules in AuDITee and AuDITee-RL can be viewed as leveraging human-like updates with certain levels of human effort and error, analogous to the HumLa and ECo-HumLa methods—current SOTA approaches for online just-in-time software defect prediction. By conducting this RQ, we can answer how much human effort our frameworks can potentially reduce and fix the values of human effort and human error in HumLa and ECo-HumLa, thus providing a benchmark for comparing their performance with the defect prediction modules in AuDITee and AuDITee-RL for other research questions.

Just Accepted

8.4.2 Results. We address RQ4 by analyzing Tables 8 and 9. The former compares the performance of the defect prediction modules in AuDITee and AuDITee-RL with ODaSC and predictor in FIP, involving multiple-method comparisons, we followed the same statistical analysis setup as in RQ1. Specifically, each experiment was repeated ten times with different random seeds, and performance differences were analyzed using the Friedman test, followed by pairwise Conover post-hoc tests with Bonferroni correction when significant. Cliff's Delta (δ) was also reported to indicate effect sizes following the interpretation guidelines of Romano et al. [63].

The latter identifies specific instances of HumLa and ECo-HumLa under comparable human-noise conditions that closely match the performance of the defect prediction modules in AuDITee and AuDITee-RL, we followed the same statistical procedure as in RQ2. Specifically, two-tailed Wilcoxon signed-rank tests [14] were used to assess pairwise differences, and Cliff's Delta (δ) was reported to indicate the practical significance of the differences according to the same interpretation guidelines [63].

Table 8 presents a comparison among the defect prediction modules of AuDITee, AuDITee-RL, FIP and ODaSC. As shown in the table, although the defect prediction module of FIP demonstrates excellent precision, it exhibits an almost negligible ability to detect defects, with a recall of only 0.0290. This further explains why, in RQ3, the total number of tests conducted by FIP was extremely low. After excluding FIP, we observed that all metrics, except for $Precision_{sdp}$, regardless of whether DynaMOSA or TestGen- LLM is integrated, exhibit significant improvements with AuDITee and AuDITee-RL. Moreover, the effect sizes for most metrics are generally large in most cases across the majority of projects. It is worth noting that, for the Tomcat project, the post-hoc test showed no statistically significant differences (p = 1.0), although the effect sizes were large in most cases, suggesting notable practical differences among the methods. After excluding FIP and repeating the same statistical testing procedure, significant differences were observed among ODaSC, AuDITee, and AuDITee-RL. In particular, for Recall, AuDITee achieves an improvement of nearly 14.8% compared with ODaSC, while AuDITee-RL improves Recall by over 15.8%. The drop in $Precision_{sdp}$ is subtle, with AuDITee decreasing by less than 2.3% and AuDITee-RL by nearly 1.3% compared with ODaSC, indicating that the proposed novel WTT update mechanism effectively mitigates the performance degradation of the defect prediction modules caused by concept drift and verification latency. Furthermore, both AuDITee and AuDITee-RL outperform ODaSC significantly in terms of G-Mean and MCC. Specifically, AuDITee achieves a G-Mean improvement of over 3.8% compared with ODaSC, while AuDITee-RL improves G-Mean by nearly 4.9%. For MCC, AuDITee exceeds ODaSC by more than 5.4%, and AuDITee-RL by over 9.6%. These results indicate that the proposed update mechanism can substantially alleviate the class imbalance problem affecting defect prediction modules.

Compared with Recall, AuDITee-RL improves Recall by 0.9% over AuDITee. Similarly, AuDITee-RL demonstrates significantly higher $Precision_{sdp}$ than AuDITee on most projects except Django, with an average improvement of 1%. In terms of F1, the improvement of AuDITee-RL over AuDITee is also limited, the average increase is only about 1%. Compared with these modest performance improvements, AuDITee-RL shows slightly more noticeable gains on metrics that are sensitive to class imbalance, with G-Mean increasing by nearly 1% and MCC by 4%. In summary, the RL–based selector can modestly enhance the performance of the defect prediction model while also partially alleviating the class imbalance problem it faces.

Moreover, by maintaining or improving defect prediction accuracy while using WTT update mechanism with, AuDITee provides a robust solution that can greatly reduce expert labor costs, which is explored in more detail in the Table 9.

We evaluate the efficacy of the proposed AuDITee and AuDITee-RL in conserving human noise and human effort by identifying specific instances of HumLa and ECo-HumLa that exhibit statistically similar performance AuDITee and AuDITee-RL across various prediction metrics.

Table 9 presents the results comparing AuDITee, AuDITee-RL, HumLa and ECo-HumLa, along with the human noise and human effort values for HumLa and ECo-HumLa that closely correspond to AuDITee's performance.

Just Accepted

Table 8. Comparison of the performance of the defect prediction modules in AuDITee and AuDITee-RL with ODaSC and predictor in FIP. A "*" indicates that the corresponding framework exhibits a statistically significant difference compared with AuDITee-RL, while a "†" denotes significance compared with AuDITee. After applying the Bonferroni correction, we computed Cliff's Delta effect sizes. Symbols [n], [s], [m], and [l] denote negligible, small, medium, and large Cliff's Delta effect sizes compared with AuDITee-RL, respectively, while {n}, {s}, {m}, and {l} denote the corresponding effect sizes compared with AuDITee. The last four rows summarize the average values of each metric across all defect predictors, with higher values indicating better performance.

metrics		Recall	$Precision_{sdp}$	F1	G-Mean	MCC
JGroups	ODaSC	0.3481*[l]†{l}	0.6861[n]†{l}	0.4508*[l]†{l}	0.5281*[l]†{l}	0.2155*[l]†{l}
	FIP	0.0078*[l]†{l}	0.8667*[l]†{l}	0.0161*[l]†{l}	0.0614*[l]†{l}	0.0337*[l]†{l}
	AuDITee	0.4515[n]	0.6685*[l]	0.5289[n]	0.5793[s]	0.2390[l]
	AuDITee-RL	0.4433	0.6865	0.5294	0.5842	0.2578
Broadleaf	ODaSC	0.2793*[l]†{l}	0.6869*[l]†{l}	0.3882*[l]†{l}	0.4842*[l]†{l}	0.1864*[l]†{l}
	FIP	0.0073*[l]†{l}	0.8832*[l]†{l}	0.0141*[l]†{l}	0.0570*[l]†{l}	0.0287*[l]†{l}
	AuDITee	0.4153[s]	0.6742[s]	0.5036[s]	0.5638[m]	0.2272[s]
	AuDITee-RL	0.4168	0.6770	0.5081	0.5683	0.2310
Tomcat	ODaSC	0.5721[l]{l}	0.6777[l][†l]	0.6140[l]{l}	0.6364[l]{l}	0.3019[l]{l}
	FIP	0.0935[l]{l}	0.7527[m][†m]	0.1089[l]{l}	0.1726[l]{l}	0.0927[l]{l}
	AuDITee	0.6909[l]	0.6602[l]	0.6686[l]	0.6547[l]	0.3351[l]
	AuDITee-RL	0.7258	0.6733	0.6944	0.6767	0.3728
Fabric8	ODaSC	0.6626[m]{n}	0.7251[m]{s}	0.6894[m]{n}	0.6985[m]{n}	0.4099[m]{n}
	FIP	0.0075*[l]†{l}	0.8786*[l]†{l}	0.0295[l]{l}	0.0535*[l]†{l}	0.0293*[l]†{l}
	AuDITee	0.6583[m]	0.7299[s]	0.6887[l]	0.6986[l]	0.4124[l]
	AuDITee-RL	0.6909	0.7329	0.7080	0.7127	0.4368
Django	ODaSC	0.7564*[l]†{l}	0.7011*[l]{s}	0.7330*[l]†{l}	0.7134[n]†{l}	0.4444*[l]†{l}
	FIP	-	-	-	-	-
	AuDITee	0.8284[s]	0.6968*[l]	0.7628[n]	0.7256*[l]	0.4869*[]
	AuDITee-RL	0.8515	0.6800	0.7627	0.7141	0.4776
Tomcat (last 1000 simples)	ODaSC	0.6843[m]†{l}	0.6990*[l]†{l}	0.6849[s][l]	0.6836*[l]†{l}	0.3911*[l]†{l}
	FIP	-	-	-	-	-
	AuDITee	0.7471*[l]	0.6503*[l]	0.6934[l]	0.6657[s]	0.3544[s]
	AuDITee-RL	0.6979	0.6715	0.6797	0.6705	0.3612
avg	ODaSC	0.5505	0.6960	0.5934	0.6240	0.3249
	FIP	0.0290	0.8453	0.0422	0.0861	0.0461
	AuDITee	0.6319	0.6800	0.6410	0.6480	0.3425
	AuDITee-RL	0.6377	0.6869	0.6471	0.6544	0.3562

Just Accepted

The respective human noise and human effort are reported in parentheses alongside their performance values. The final row of the table summarizes the mean human noises and human efforts for HumLa and ECo-HumLa across the performance metrics in defect prediction.

We can see that AuDITee and AuDITee-RL effectively eliminate the needs for human inspection while maintaining satisfactory predictive performance. This suggests that AuDITee significantly contributes to both the quality

and quantity of defect prediction compared to the SOTA half-automated methods like HumLa. Specifically, AuDITee aligns with ECo-HumLa at 0.152-human noise and with HumLa at 0.68-human effort and 0.156-human noise. AuDITee-RL aligns with ECo-HumLa at 0.148-human noise and HumLa at 0.672-human effort and 0.192-human noise.

These findings indicate that AuDITee and AuDITee-RL significantly reduce the reliance on expert labors in HumLa and Eco-HumLa, matching their performance in defect prediction at higher human effort levels while retaining reliable prediction capability at lower human noise levels. By replacing the need for high-quality human inspection, AuDITee and AuDITee-RL conserve human resources for reviewing defect-predicted software changes.

It is important to emphasize that HumLa and ECo-HumLa are not intended to serve as baseline methods in our evaluation, but rather as auxiliary experiments to provide a coarse estimation of potential human cost savings. Since both approaches rely on discrete parameter settings, we sample human noise and human effort values at intervals of 0.1. In some cases, AuDITee or AuDITee-RL achieves performance that is statistically superior to the best configuration of HumLa or ECo-HumLa and the effect size is large. This not only demonstrates the robustness and superiority of our frameworks, but also means that we may not always identify a parameter setting in HumLa or ECo-HumLa that produces results statistically indistinguishable from those of our frameworks.

Moreover, while HumLa and ECo-HumLa generally show better performance with lower human noise and higher human effort, this trend is not absolute. There are instances where AuDITee-RL outperforms AuDITee, yet the corresponding HumLa (or ECo-HumLa) parameter configuration appears less favorable. This further reinforces that direct performance comparison through parameter matching is not appropriate.

In summary, the inclusion of HumLa and ECo-HumLa is solely to provide a rough estimation of how much human effort our frameworks can potentially reduce. They are not designed to compete with or replace the proposed approaches.

Answer RQ4

Our newly proposed WTT update mechanism achieves significant improvements over defect prediction models that rely solely on the waiting time mechanism and performs comparably to HumLa and ECo-HumLa, which employ high-quality human updates. At the same time, the new update mechanism also helps alleviate the issue of class imbalance. The RL–based selector further provides slight improvements to the defect prediction module while also contributing to mitigating class imbalance. Finally, we fixed the parameters for HumLa at human effort = 0.672 and human noise = 0.192, and for ECo-HumLa at human noise = 0.148. Under these settings, both modules achieve predictive performance comparable to that of the defect prediction module in AuDITee-RL.

Just Accepted

Table 9. Performance comparisons between AuDITee and AuDITee-RL's defect prediction modules and specific instances of HumLa and ECo-HumLa under comparable human noise conditions that align closely with AuDITee's performance. The parameters in parentheses refer to HumLa and ECo-HumLa: HumLa includes human effort and human noise, while ECo-HumLa includes only human noise. A "*" indicates that the corresponding framework exhibits a statistically significant difference compared with AuDITee-RL, while a "†" denotes significance compared with AuDITee. Values in bold indicate the optimal solution of HumLa or ECo-HumLa. Symbols [n], [s], [m], and [l] denote negligible, small, medium, and large Cliff's Delta effect sizes compared with AuDITee-RL, respectively, while {n}, {s}, {m}, and {l} denote the corresponding effect sizes compared with AuDITee. The last row presents the average values of the parameters for HumLa and ECo-HumLa.

method		AuDITee	ECo-HumLa		HumLa			AuDITee-RL	ECo-HumLa		HumLa		
			value	human noise	value	human effort	human noise		value	human noise	value	human effort	human noise
JGroups	Recall	0.4515	**0.3915†{l}**	0	0.4541{n}	0.8	0.1	0.4433	**0.3915*[l]**	0	0.4410[n]	0.5	0
	Precision	0.6685	0.6803{l}	0.6	0.6688{n}	0.7	0.2	0.6865	0.6872[n]	0.2	0.6862[n]	0.8	0.8
	F1	0.5289	**0.4906†{l}**	0	0.5316{n}	0.6	0	0.5294	**0.4906*[l]**	0	0.5316[n]	0.6	0
	G-Mean	0.5793	**0.5568†{l}**	0	0.5785{n}	0.5	0	0.5842	**0.5568*[l]**	0	0.5846[n]	0.8	0.1
	MCC	0.239	**0.2334{s}**	0	0.2386{n}	0.8	0.3	0.2578	**0.2334*[l]**	0	0.2575[n]	0.7	0
Broadleaf	Recall	0.4153	0.4218{m}	0.1	0.4128{n}	0.4	0	0.4168	0.4218[n]	0.1	0.4128[n]	0.4	0
	Precision	0.6742	0.6759{s}	0.4	0.6744{n}	0.7	0.1	0.6770	0.6776[n]	0.3	0.6771[n]	0.5	0.1
	F1	0.5036	0.5134{l}	0.1	0.5036{n}	0.4	0	0.5081	0.5134[n]	0.1	0.5093[n]	0.8	0.3
	G-Mean	0.5638	0.5552{l}	0.2	0.5652{n}	0.4	0	0.5683	0.5732[n]	0.1	0.5686[n]	0.8	0.3
	MCC	0.2272	0.2274{n}	0.2	0.2268{n}	0.9	0.4	0.2310	0.2274[s]	0.2	0.2305[n]	1	0.5
Tomcat	Recall	0.6909	**0.6809{m}**	0	0.6917{n}	0.6	0.2	0.7258	**0.6809*[l]**	0	0.7254[s]	0.7	0
	Precision	0.6602	0.6593{n}	0	0.6601{n}	0.8	0.4	0.6733	0.6723[s]	0.4	0.6733[n]	0.1	0.1
	F1	0.6686	0.6643{m}	0	0.6695{n}	0.5	0	0.6944	**0.6643*[l]**	0	**0.6898[m]**	0.9	0
	G-Mean	0.6547	**0.6542{n}**	0.1	0.6546{n}	0.7	0.1	0.6767	**0.6542*[l]**	0.1	**0.6596*[l]**	0.6	0.2
	MCC	0.3351	0.3289{m}	0	0.3346{n}	0.6	0.1	0.3728	**0.3289*[l]**	0	**0.3627[m]**	1	0
Fabric8	Recall	0.6583	-	-	-	-	-	0.6909	-	-	-	-	-
	Precision	0.7299	-	-	-	-	-	0.7329	-	-	-	-	-
	F1	0.6887	-	-	-	-	-	0.7080	-	-	-	-	-
	G-Mean	0.6986	-	-	-	-	-	0.7127	-	-	-	-	-
	MCC	0.4124	-	-	-	-	-	0.4368	-	-	-	-	-
Django	Recall	0.8284	**0.8242{n}**	0.2	0.8298{n}	0.9	0	0.8515	**0.8242*[l]**	0.2	0.8518[s]	0.5	0
	Precision	0.6968	0.6975{n}	0	0.6968{n}	0.1	0.6	0.6800	0.6797[n]	0.3	0.6793[n]	0.7	0.1
	F1	0.7628	**0.7582{s}**	0	0.7627{n}	1	0	0.7627	**0.7582[s]**	0	0.7627[s]	1	0
	G-Mean	0.7256	**0.7250{n}**	0.1	**0.7239{s}**	1	0.1	0.7141	0.7161[n]	0.2	0.7141[n]	0.2	0
	MCC	0.4869	**0.4788{m}**	0.1	0.4869{s}	1	0.2	0.4776	0.4784[s]	0	0.4774[s]	0.5	0.1
Tomcat (last 1000 simples)	Recall	0.7471	**0.6838†{l}**	0	0.7471{s}	0.9	0.1	0.6979	**0.6838[s]**	0	0.6977[n]	0.7	0.3
	Precision	0.6503	0.6594{m}	0	0.6529{n}	1	0	0.6715	0.6654[s]	0.1	0.6710[n]	0.8	0.2
	F1	0.6934	**0.6751†{l}**	0.9	0.6934{n}	0.1	1	0.6797	**0.6751[s]**	0.9	0.6795[n]	0.8	0.4
	G-Mean	0.6657	0.6669{n}	0.4	0.6655{n}	0.8	0	0.6705	0.6712[n]	0.3	0.6706[n]	0.9	0.3
	MCC	0.3544	0.3529{n}	0.4	0.3558{n}	0.8	0	0.3612	0.3612[n]	0.2	0.3620[n]	0.5	1
avg				0.152		0.68	0.156			0.148		0.672	0.192

Note: Under the Fabric8 project, AuDITee and AuDITee-RL achieved superior performance. In contrast, HumLa and ECo-HumLa showed abnormal results: while smaller human error is normally better, they exhibited the opposite trend, peaking at human error = 1. Such unreasonable cases are excluded to avoid severe bias in the results.

Just Accepted

8.5 RQ5: Compared with conventional defect prediction modules, how much can defect predictors enhanced with the proposed update mechanism (i.e., AuDITee and AuDITee-RL) improve the quality of guided testing? Can the RL–based dynamic selector (i.e., AuDITee-RL), which intelligently determines when to trigger testing, further enhance the quality of guided testing? Finally, are the results sensitive to the choice of underlying testing frameworks?

8.5.1 Motivation. Motivated by the results of RQ2, which showed that AuDITee and AuDITee-RL effectively reduce testing resource consumption, RQ5 investigates whether defect predictors trained with a novel WTT update mechanism can produce higher-quality test cases than conventional defect prediction models using either waiting-time-only updates or high-quality manual feedback. In addition, we examine whether the RL–based dynamic selector in AuDITee-RL, which adaptively decides when to trigger testing, can further enhance the quality of guided testing.

8.5.2 Results. We answer RQ5 by analyzing the results presented in Tables 10 and 11. Table 10 presents the testing performance of AuDITee and AuDITee-RL compared to DynaMOSA and simple integration frameworks that combine SOTA defect predictors with DynaMOSA. Table 11 reports the testing performance of AuDITee and AuDITee-RL compared with TestGen-LLM and simple integration frameworks that combine SOTA defect predictors with TestGen-LLM.

Since RQ5 involves multiple methods and to mitigate the impact of randomness in the frameworks, each experiment was repeated 10 times using different random seeds. To assess whether the observed performance differences among the methods were statistically significant across these repeated runs, we applied the Friedman test [17]. When the null hypothesis was rejected ($\alpha = 0.05$), we conducted pairwise Conover post-hoc tests with Benjamini–Hochberg false discovery rate correction to account for multiple comparisons. In addition, we reported effect sizes using Kendall's W (0.1 = small, 0.3 = moderate, >0.5 = large) to indicate the magnitude of the observed differences.

Table 10 presents the experimental results of all frameworks integrated with DynaMOSA. We also report the performance of standalone DynaMOSA, representing the overall testing quality without defect prediction guidance. Below, we analyze the differences observed across each metric.

- With respect to *False Discovery Rate (FDR)*:
  - FIP still exhibits the worst performance in terms of FDR, with an average of 0.0081. After excluding FIP, compared to DynaMOSA, both testing modules in AuDITee and AuDITee-RL exhibit consistently and significantly lower FDR across all projects. On average, the testing modules in AuDITee and AuDITee-RL detect nearly 40.9% and 48.1% fewer defects than DynaMOSA, respectively. However, as shown in RQ3, the number of test cases selected by AuDITee is only about 30% of that used by DynaMOSA, with AuDITee-RL using even fewer. This highlights that our frameworks are able to detect relatively more defects per unit of testing effort, indicating a substantial improvement in testing efficiency.
  - Importantly, as demonstrated in RQ1, the overall defect detection ability of the full frameworks is enhanced due to the integration of the defect prediction module. Therefore, the reduced number of defects discovered solely by the testing module is not a cause for concern, as it is complemented by the prediction component.
  - We also compared multiple simple integration frameworks to examine whether AuDITee and AuDITee-RL provide superior guidance for testing compared to other SOTA models. Our results show that AuDITee and AuDITee-RL are significantly better than ODaSC, which uses only the waiting time update mechanism, on most projects, with large effect sizes. Specifically, AuDITee improves performance by over 21.7% compared to ODaSC, while AuDITee-RL achieves nearly a 7% improvement.

Just Accepted

Table 10. Testing performance of AuDITee and AuDITee-RL when integrated with DynaMOSA, compared to the standalone DynaMOSA and simple integration frameworks that combine various SOTA defect predictors with DynaMOSA. A "*" indicates a statistically significant difference compared with AuDITee-RL, while a "†" indicates significance compared with AuDITee. Following the Bonferroni correction, Cliff's Delta was calculated to quantify the pairwise effect sizes. Symbols [n], [s], [m], and [l] denote negligible, small, medium, and large Cliff's Delta effect sizes compared with AuDITee-RL, respectively, while {n}, {s}, {m}, and {l} denote the corresponding effect sizes compared with AuDITee. The last seven column summarizes the average metrics across all projects for each framework.

metrics		FDR	$Precision_{testing}$	RTR(new)
JGroups	DynaMOSA	0.3895*[l]†{l}	0.0950*[l]†{l}	0.7560*[l]{l}
	ODaSC4Test	0.1422[s]†{l}	0.1630[m]{n}	0.5741[m]{l}
	HumLa4Test	0.1856*[m]{n}	0.1681[m]{n}	0.5791[l]{l}
	Eco-HumLa4Test	0.1627[n]†{m}	0.1652[m]{n}	0.5841[l]{l}
	FIP	0.0036*[l]†{l}	0.2588[s]{s}	0.3787[l]†{l}
	AuDITee	0.1946*[m]	0.1643[m]	0.5971*[l]{l}
	AuDITee-RL	0.1552	0.1900	0.5599
Broadleaf	DynaMOSA	0.3260*[l]†{l}	0.0913*[l]†{l}	0.7199*[l]†{l}
	ODaSC4Test	0.0921*[l]†{l}	0.1493[s]{n}	0.5310*[l]†{l}
	HumLa4Test	0.1437[m]{n}	0.1460[m]{n}	0.5544*[l]{s}
	Eco-HumLa4Test	0.1335*[l]{s}	0.1475[s]{n}	0.5485*[l]{m}
	FIP	0.0012*[l]†{l}	0.0979*[l]{l}	0.4091*[l]†{l}
	AuDITee	0.1398[m]	0.1454[m]	0.5578[l]
	AuDITee-RL	0.1555	0.1534	0.5725
Tomcat	DynaMOSA	0.3810 *[l] †{l}	0.1406 *[l] †{l}	0.6311*[l]†{l}
	ODaSC4Test	0.2089 *[l]†{l}	0.1981* [l]{s}	0.4594*[l]†{l}
	HumLa4Test	0.2622* [l]{s}	0.1969 *[l]{m}	0.4752*[l]{s}
	Eco-HumLa4Test	0.2470 *[l] †{l}	0.1979 *[l]{m}	0.4697*[l]{s}
	FIP	0.0218 *[l] †{l}	0.1879 *[l]{n}	0.4136[s]†{l}
	AuDITee	0.2634 *[l]	0.1997 [l]	0.4722*[l]
	AuDITee-RL	0.2321	0.2472	0.4232
Fabric8	DynaMOSA	0.3534*[l]†{l}	0.0986*[l]†{l}	0.7211*[l]†{l}
	ODaSC4Test	0.2608*[l]{n}	0.2029[n]{n}	0.4962*[l]{s}
	HumLa4Test	0.2662*[l]{n}	0.2065[n]{n}	0.4944[l]{n}
	Eco-HumLa4Test	0.2646*[l]{n}	0.1998[s]{n}	0.5071*[l]{m}
	FIP	0.0056[l]†{l}	0.2909[s]{s}	0.3356[n]{s}
	AuDITee	0.2588*[l]	0.2030[n]	0.4901*[l]
	AuDITee-RL	0.2104	0.2077	0.4464
avg	DynaMOSA	0.3625	0.1064	0.7070
	ODaSC4Test	0.1760	0.1783	0.5152
	HumLa4Test	0.2144	0.1794	0.5258
	Eco-HumLa4Test	0.2020	0.1776	0.5274
	FIP	0.0081	0.2089	0.3843
	AuDITee	0.2142	0.1781	0.5293
	AuDITee-RL	0.1883	0.1996	0.5005

Just Accepted

Moreover, AuDITee demonstrates FDR comparable to HumLa4Test across all projects, further confirming the effectiveness of defect predictors trained with the novel WTT update mechanism in guiding testing. When incorporating the dynamic selector, AuDITee-RL exhibits a lower FDR than AuDITee, decreasing by approximately 12.1%. However, as reported in RQ3, AuDITee-RL tests roughly 23.9% fewer code changes than AuDITee, indicating that the reduction in detected defects is within a reasonable range. Furthermore, according to RQ1, when considering both defect prediction and testing outputs, AuDITee-RL shows a significantly superior ability to identify defects compared to AuDITee.

- With respect to $Precision_{testing}$:
  - $Precision_{testing}$ significantly improves in both testing modules in AuDITee and AuDITee-RL compared to DynaMOSA, across all datasets. The average precision rises from 0.1064 (DynaMOSA) to 0.1781 (AuDITee) and 0.1996 (AuDITee-RL). These results confirm that although fewer defects are detected, the proportion of relevant (i.e., true defective) test targets increases substantially, showing that the frameworks prioritize more valuable code changes.
  - Moreover, AuDITee-RL further improves upon AuDITee in all projects. This demonstrates the effectiveness of reinforcement learning in capturing subtle feedback signals and refining test selection, resulting in more high-precision testing decisions.
  - When comparing with other frameworks that integrate defect prediction and testing, we find that the testing module of AuDITee shows no significant differences compared to the testing modules of all other integrated frameworks. AuDITee-RL, however, is significantly better than ODaSC4Test ,Eco-HumLa4Test and HumLa4Test on one projects. FIP achieves performance comparable to AuDITee-RL, with improvements of less than 4.7%. Overall, regardless of the update mechanism used to train the defect predictor, all models effectively improve the precision of testing. Differences among these models are relatively small. When excluding FIP, AuDITee-RL demonstrates the best overall performance, while AuDITee, ODaSC4Test, HumLa4Test, and Eco-HumLa4Test exhibit negligible differences.
- With respect to *Redundant Test Rate (RTR)*:
  - RTR reflects the proportion of unnecessary or duplicated tests. Lower values are better, indicating reduced test redundancy. Compared to DynaMOSA (avg RTR: 0.7070), both AuDITee and AuDITee-RL reduce redundancy significantly, reaching 0.5293 and 0.5005 on average, respectively. Across all projects, the improvements in RTR are statistically significant, with large effect sizes observed in every case. In the JGroups project, although AuDITee shows no statistically significant difference compared to DynaMOSA, the effect size remains large, suggesting a substantial practical advantage.
  - Compared to testing modules guided by other simple integration frameworks, AuDITee shows little difference from ODaSC4Test, HumLa4Test, and Eco-HumLa4Test, with the largest gap being only a 2.7% higher RTR than ODaSC4Test. With the addition of the dynamic selection mechanism, AuDITee-RL further reduces the RTR by 5.4%. Notably, FIP achieves the best result with an RTR of only 0.3843, indicating a lower proportion of redundant tests. However, considering that FIP tests only 232 code changes and achieves an FDR of merely 0.0081, its smaller redundancy ratio has limited practical significance.
  - The improvement in RTR demonstrates that the proposed frameworks not only focus on testing the most relevant changes but also avoid wasteful testing efforts. AuDITee-RL again achieves the best performance, showing that it can efficiently allocate testing resources while maintaining or even improving effectiveness.

Just Accepted

Table 11. Testing performance of AuDITee and AuDITee-RL when integrated with TestGen-LLM, compared to the standalone TestGen-LLM and simple integration frameworks that combine various SOTA defect predictors with TestGen-LLM. A "*" indicates a statistically significant difference compared with AuDITee-RL, while a "†" indicates significance compared with AuDITee. Following the Bonferroni correction, Cliff's Delta was calculated to quantify the pairwise effect sizes. Symbols [n], [s], [m], and [l] denote negligible, small, medium, and large Cliff's Delta effect sizes compared with AuDITee-RL, respectively, while {n}, {s}, {m}, and {l} denote the corresponding effect sizes compared with AuDITee. The last seven column summarizes the average metrics across all projects for each framework.

metrics		FDR	$Precision_{testing}$	RTR(new)
Django	TestGen-LLM	0.3656*[l]†{l}	0.0340*[l]†{l}	0.9070*[l]†{l}
	ODaSC4Test	0.2570[l]{m}	0.0686[n]{l}	0.8017*[l]{s}
	HumLa4Test	0.2871*[l]{s}	0.0624*[l]{l}	0.8178*[l]†{l}
	Eco-HumLa4Test	0.2699*[l]{s}	0.0663[s]{s}	0.8039*[l]{n}
	FIP	-	-	-
	AuDITee	0.2742*[l]	0.0651[s]	0.8037*[l]
	AuDITee-RL	0.1903	0.0710	0.7815
Tomcat (last 1000 simples)	TestGen-LLM	0.4451*[l]†{l}	0.0730*[l]†{l}	0.8360*[l]†{l}
	ODaSC4Test	0.2677*[l]†{l}	0.1294[l]†{l}	0.6969*[l]†{l}
	HumLa4Test	0.2933*[l]†{l}	0.1198*[l]†{l}	0.7216*[l]†{l}
	Eco-HumLa4Test	0.2878*[l]†{l}	0.1186*[l]†{l}	0.7250*[l]†{l}
	FIP	-	-	-
	AuDITee	0.3098*[l]	0.1124*[l]	0.7382*[l]
	AuDITee-RL	0.2177	0.1390	0.6772
avg	TestGen-LLM	0.4054	0.0535	0.8715
	ODaSC4Test	0.2624	0.0990	0.7493
	HumLa4Test	0.2902	0.0911	0.7697
	Eco-HumLa4Test	0.2789	0.0925	0.7645
	FIP	-	-	-
	AuDITee	0.2920	0.0888	0.7710
	AuDITee-RL	0.2040	0.1050	0.7294

Table 11 presents the experimental results of all frameworks integrated with TestGen-LLM. We also report the performance of standalone TestGen-LLM, representing the overall testing quality without defect prediction guidance. The frameworks integrated with TestGen-LLM exhibit results similar to those with DynaMOSA: all integrated frameworks are significantly weaker than the standalone testing technique in terms of FDR, but achieve significantly better $Precision_{testing}$ and RTR. Likewise, AuDITee-RL obtains significantly worse FDR than AuDITee, while achieving significantly better $Precision_{testing}$ and RTR. Notably, AuDITee-RL's FDR is even lower than that of ODaSC4Test, a result that differs from the integration with DynaMOSA. This discrepancy can be explained by the strong influence of the total number of tests on FDR. As shown in RQ3, when integrated with DynaMOSA, AuDITee-RL executes only about 1.7% fewer tests than DynaMOSA, whereas when integrated with TestGen-LLM, AuDITee-RL executes nearly 26.7% fewer tests than TestGen-LLM.

Just Accepted

Answer RQ5

In terms of guiding high-quality testing, defect predictors trained with the novel WTT update mechanism generally outperform those updated solely with waiting time, achieving performance comparable to models trained with high-quality manual feedback. When incorporating the RL–based dynamic selector, AuDITee-RL conducts substantially fewer tests, resulting in the lowest FDR; however, it achieves clear advantages in both $Precision_{testing}$ and RTR. Overall, the dynamic selector enhances testing quality despite a reduction in detected defects. Importantly, the consistent superiority of our framework when integrated with both SBST-based techniques (i.e., DynaMOSA) and LLM-based techniques (i.e., TestGen-LLM) demonstrates that its benefits are robust and not dependent on the choice of testing tool.

9 Threats to Validity

Internal Validity – Similar to previous research using the SZZ algorithm with `Commit Guru` for automatic labeling of code changes [7, 67, 69], our work is subject to noise from SZZ, particularly from the `git blame` command [29, 34, 46, 60, 61]. We used the original SZZ algorithm [66], which produces defect prediction models with comparable accuracy to the latest SZZ variant with additional noise filters [18]. To mitigate randomness in automated testing and defect prediction, our results are based on ten runs per dataset.

Consistent with prior work on software testing, the testing conducted in our experiments does not constitute exhaustive testing and therefore does not fully cover all functionalities of the software systems. However, we ensure that all configurations across the compared methods are identical, meaning that all approaches are evaluated under the same testing conditions. This allows us to isolate and validate the impact of selecting the appropriate testing moments. Furthermore, for each code change selected for testing, we execute the tests 10 times with different random seeds, in order to maximize the likelihood of defect detection and ensure robustness of the results.

Construct Validity – We utilized Recall, Precision, F1 score, G-Mean and MCC as unbiased metrics suitable for class-imbalanced defect prediction. A fading factor was applied to monitor performance fluctuations over time, as recommended for online learning [23].

External Validity – While this study analysis five open-source Java and Python projects with a substantial number of code changes, the results may not generalize to other projects. Our framework utilized advanced defect predictors (HumLa, ECo-HumLa, and ODaSC) and an advanced testing algorithm (DynaMOSA and TestGen-LLM) that have demonstrated effectiveness in prior studies [52, 67, 70]. Different defect prediction models and testing algorithms might yield varying outcomes. We focused on code changes accepted into the main branch, consistent with JIT-SQA literature, so results may not extend to rejected changes.

10 Conclusion

The primary objective of this study is to facilitate prompt defect detection for JIT-SQA. To this end, we proposed a novel JIT-SQA framework called AuDITee, which simultaneously addresses defect prediction and automated testing, thereby offering the potential for delivering high-quality software products.

Within this framework, defect predictors and automated testing process collaboratively enhance each other through shared information. Defect predictors guide the testing process to reduce resource consumption, while testing feedback helps refine the predictors, maintaining or even improving their performance. Specifically, we design a novel update mechanism called WTT to train an online, just-in-time software defect prediction model. This mechanism effectively mitigates the negative impact of concept drift and verification latency on model performance. Moreover, we introduce a RL-based dynamic selector, which offers a mechanism for adaptive selection, alleviating the issue of performance fluctuate in the defect prediction model. In parallel, the collaboration

Just Accepted

between the dynamic selector and the continuous testing process provides high-quality defect samples during training. Together, these components effectively address the class imbalance problem in code evolution scenarios.

Our experiments demonstrate that the proposed integrated framework can significantly reduce testing costs while simultaneously enhancing defect detection capabilities. Compared with simply combining the two techniques, our framework more effectively addresses concept drift, verification latency, and class imbalance.

Future work could explore the integration of advanced defect prediction models and more sophisticated automated testing algorithms to further enhance the performance of the proposed JIT-SQA framework, ensuring continuous improvement in software development processes. In particular, future studies would focus on enhancing the online learning–based defect prediction mechanism by employing adaptive concept drift detection and improving feature representation through the use of LLMs. Moreover, incorporating emerging AI approaches holds great potentials to expand the capabilities of AuDITee. For instance, building upon the current LLM-based test generation component, future work could further leverage LLMs to refine adaptive test generation and enhance feedback interpretation. In addition, the RL-based dynamic selector could adopt more advanced reinforcement learning strategies to improve decision-making in the context of evolving software environments.

Acknowledgment

This work was supported by the National Natural Science Foundation of China (NSFC) under Grant Nos. 62272132 and 62572154, and the Heilongjiang Provincial Natural Science Foundation under Grant No. JJ2024LH1948.

Data Availability

The data and source code used in this study are publicly available at: https://github.com/crazyTang-cloud/AuDITee.git. All datasets and scripts required to reproduce the results are accessible without restrictions.

Just Accepted

References

[1] Nadia Alshahwan, Jubin Chheda, Anastasia Finogenova, Beliz Gokkaya, Mark Harman, Inna Harper, Alexandru Marginean, Shubho Sengupta, and Eddy Wang. 2024. Automated Unit Test Improvement using Large Language Models at Meta. In Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering (Porto de Galinhas, Brazil) (FSE 2024). Association for Computing Machinery, New York, NY, USA, 185–196. doi:10.1145/3663529.3663839

[2] Nadia Alshahwan, Mark Harman, Alexandru Marginean, Rotem Tal, and Eddy Wang. 2024. Observation-Based Unit Test Generation at Meta. In Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering (Porto de Galinhas, Brazil) (FSE 2024). Association for Computing Machinery, New York, NY, USA, 173–184. doi:10.1145/3663529.3663838

[3] Andrea Arcuri and Gordon Fraser. 2013. Parameter tuning or default values? An empirical investigation in search-based software engineering. Empirical Software Engineering 18 (2013), 594–623. https://api.semanticscholar.org/CorpusID:18284089

[4] Abdelrahman Baz, Minchao Huang, and August Shi. 2024. Prioritizing Tests for Improved Runtime. In Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering (Sacramento, CA, USA) (ASE '24). Association for Computing Machinery, New York, NY, USA, 2273–2278. doi:10.1145/3691620.3695298

[5] George Gomes Cabral and Leandro L. Minku. 2023. Towards Reliable Online Just-in-time Software Defect Prediction. IEEE Transactions on Software Engineering 49, 3 (2023), 1342–1358. doi:10.1109/TSE.2022.3175789

[6] George G. Cabral, Leandro L. Minku, Adriano L. I. Oliveira, Dinaldo A. Pessoa, and Sadia Tabassum. 2023. An investigation of online and offline learning models for online Just-in-Time Software Defect Prediction. Empirical Softw. Engg. 28, 5 (sep 2023), 35 pages. doi:10.1007/s10664-023-10335-6

[7] George G. Cabral, Leandro L. Minku, Emad Shihab, and Suhaib Mujahid. 2019. Class Imbalance Evolution and Verification Latency in Just-in-Time Software Defect Prediction. In International Conference on Software Engineering. Monteal, Canada, 666–676. doi:10.1109/ICSE.2019.00076

[8] José Campos, Annibale Panichella, and Gordon Fraser. 2019. EvoSuite at the SBST 2019 Tool Competition. In 2019 IEEE/ACM 12th International Workshop on Search-Based Software Testing (SBST). 29–32. doi:10.1109/SBST.2019.00017

[9] Yinghao Chen, Zehao Hu, Chen Zhi, Junxiao Han, Shuiguang Deng, and Jianwei Yin. 2024. ChatUniTest: A Framework for LLM-Based Test Generation. In Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering (Porto de Galinhas, Brazil) (FSE 2024). Association for Computing Machinery, New York, NY, USA, 572–576. doi:10.1145/3663529.3663801

[10] Runxiang Cheng, Shuai Wang, Reyhaneh Jabbarvand, and Darko Marinov. 2024. Revisiting Test-Case Prioritization on Long-Running Test Suites. In Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (Vienna, Austria) (ISSTA 2024). Association for Computing Machinery, New York, NY, USA, 615–627. doi:10.1145/3650212.3680307

[11] Pynguin Contributors. 2024. Generating Assertions. Retrieved June 4, 2024 from https://web.archive.org/web/20220615155509/https://pynguin.readthedocs.io/en/latest/user/assertions.html

[12] Emilio Cruciani, Breno Miranda, Roberto Verdecchia, and Antonia Bertolino. 2019. Scalable Approaches for Test Suite Reduction. In 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). 419–429. doi:10.1109/ICSE.2019.00055

[13] Andrea Dal Pozzolo, Giacomo Boracchi, Olivier Caelen, Cesare Alippi, and Gianluca Bontempi. 2018. Credit Card Fraud Detection: A Realistic Modeling and a Novel Learning Strategy. IEEE Transactions on Neural Networks and Learning Systems 29, 8 (2018), 3784–3797. doi:10.1109/TNNLS.2017.2736643

[14] Janez Demšar. 2006. Statistical Comparisons of Classifiers over Multiple Data Sets. The Journal of Machine Learning Research 7 (2006), 1–30.

[15] Elizabeth Dinella, Gabriel Ryan, Todd Mytkowicz, and Shuvendu K. Lahiri. 2022. TOGA: a neural method for test oracle generation. In Proceedings of the 44th International Conference on Software Engineering (Pittsburgh, Pennsylvania) (ICSE '22). Association for Computing Machinery, New York, NY, USA, 2130–2141. doi:10.1145/3510003.3510141

[16] Pedro Domingos and Geoff Hulten. 2000. Mining high-speed data streams. In Proceedings of the Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (Boston, Massachusetts, USA) (KDD '00). Association for Computing Machinery, New York, NY, USA, 71–80. doi:10.1145/347090.347107

[17] Karl Dyer, Robert Capo, and Robi Polikar. 2014. COMPOSE: A Semisupervised Learning Framework for Initially Labeled Nonstationary Streaming Data. IEEE Transactions on Neural Networks and Learning Systems 25 (2014), 12–26. doi:10.1109/TNNLS.2013.2277712

[18] Yuanrui Fan, Xin Xia, Daniel Alencar da Costa, David Lo, Ahmed E. Hassan, and Shanping Li. 2021. The Impact of Mislabeled Changes by SZZ on Just-in-Time Defect Prediction. IEEE Transactions on Software Engineering 47, 8 (2021), 1559–1586. doi:10.1109/TSE.2019.2929761

[19] Federico Formica, Tony Fan, and Claudio Menghi. 2023. Search-Based Software Testing Driven by Automatically Generated and Manually Defined Fitness Functions. ACM Trans. Softw. Eng. Methodol. 33, 2, Article 40 (Dec. 2023), 37 pages. doi:10.1145/3624745

[20] Gordon Fraser and Andrea Arcuri. 2011. Evosuite: automatic test suite generation for object-oriented software. In Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering. 416–419. doi:10.1145/2025113.2025179

Just Accepted

[21] Gordon Fraser and Andrea Arcuri. 2013. Whole Test Suite Generation. IEEE Transactions on Software Engineering 39, 2 (2013), 276–291. doi:10.1109/TSE.2012.14

[22] Gordon Fraser and Andrea Arcuri. 2025. A Retrospective on Whole Test Suite Generation: On the Role of SBST in the Age of LLMs. IEEE Transactions on Software Engineering (2025), 1–5. doi:10.1109/TSE.2025.3539458

[23] Joao Gama, Raquel Sebastiao, and Pedro Pereira Rodrigues. 2013. On Evaluating Stream Learning Algorithms. Journal of Machine Learning 90, 3 (2013), 317–346. doi:10.1007/s10994-012-5320-9

[24] Martin Gruber, Muhammad Firhard Roslan, Owain Parry, Fabian Scharnböck, Phil McMinn, and Gordon Fraser. 2024. Do Automatic Test Generation Tools Generate Flaky Tests?. In Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (Lisbon, Portugal) (ICSE '24). Association for Computing Machinery, New York, NY, USA, Article 47, 12 pages. doi:10.1145/3597503.3608138

[25] Giovani Guizzo, Justyna Petke, Federica Sarro, and Mark Harman. 2021. Enhancing Genetic Improvement of Software with Regression Test Selection. In 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE). 1323–1333. doi:10.1109/ICSE43902.2021.00120

[26] Andrew Habib and Michael Pradel. 2018. How Many of All Bugs Do We Find? A Study of Static Bug Detectors. In 2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE). 317–328. doi:10.1145/3238147.3238213

[27] Mark Harman. 2018. Evolutionary algorithms for software testing in Facebook. SIGEVOlution 11, 2 (aug 2018), 7. doi:10.1145/3264700.3264702

[28] Ishrak Hayet, Adam Scott, and Marcelo d'Amorim. 2025. ChatAssert: LLM-Based Test Oracle Generation With External Tools Assistance. IEEE Transactions on Software Engineering 51, 1 (2025), 305–319. doi:10.1109/TSE.2024.3519159

[29] Steffen Herbold, Alexander Trautsch, Fabian Trautsch, and Benjamin Ledel. 2022. Problems with SZZ and Features: An Empirical Study of the State of Practice of Defect Prediction Data Collection. Empirical Software Engineering 27, 2 (2022). doi:10.1007/s10664-021-10092-4

[30] Gunel Jahangirova. 2017. Oracle problem in software testing. In Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (Santa Barbara, CA, USA) (ISSTA 2017). Association for Computing Machinery, New York, NY, USA, 444–447. doi:10.1145/3092703.3098235

[31] Siyu Jiang, Zhenhang He, Yuwen Chen, Mingrong Zhang, and Le Ma. 2024. Mobile Application Online Cross-Project Just-in-Time Software Defect Prediction Framework. ACM Trans. Softw. Eng. Methodol. 33, 6, Article 157 (June 2024), 31 pages. doi:10.1145/3664607

[32] Yasutaka Kamei, Takafumi Fukushima, Shane Mcintosh, Kazuhiro Yamashita, Naoyasu Ubayashi, and Ahmed E. Hassan. 2016. Studying Just-in-Time Defect Prediction Using Cross-project Models. Empirical Software Engineering 21, 5 (2016), 2072–2106. doi:10.1007/s10664-015-9400-x

[33] Yasutaka Kamei, Emad Shihab, Bram Adams, Ahmed E. Hassan, Audris Mockus, Anand Sinha, and Naoyasu Ubayashi. 2013. A Large-Scale Empirical Study of Just-in-Time Quality Assurance. IEEE Transactions on Software Engineering 39, 6 (2013), 757–773. doi:10.1109/TSE.2012.70

[34] Sunghun Kim, E. James Whitehead, and Yi Zhang. 2008. Classifying Software Changes: Clean or Buggy? IEEE Transactions on Software Engineering 34, 2 (2008), 181–196. doi:10.1109/TSE.2007.70773

[35] Ludmila I. Kuncheva and J. Salvador Sánchez. 2008. Nearest Neighbour Classifiers for Streaming Data with Delayed Labelling. In IEEE International Conference on Data Mining. 869–874. doi:10.1109/ICDM.2008.33

[36] Gichan Lee, Hansae Ju, and Scott Uk-Jin Lee. 2024. NeuroJIT: Improving Just-In-Time Defect Prediction Using Neurophysiological and Empirical Perceptions of Modern Developers. In Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering (Sacramento, CA, USA) (ASE '24). Association for Computing Machinery, New York, NY, USA, 594–605. doi:10.1145/3691620.3695056

[37] Myungho Lee, Jiseong Bak, Seokhyeon Moon, Yoon-Chan Jhi, and Hakjoo Oh. 2024. Effective Unit Test Generation for Java Null Pointer Exceptions. In Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering (Sacramento, CA, USA) (ASE '24). Association for Computing Machinery, New York, NY, USA, 1044–1056. doi:10.1145/3691620.3695484

[38] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K. Lahiri, and Siddhartha Sen. 2023. CodaMosa: Escaping Coverage Plateaus in Test Generation with Pre-Trained Large Language Models. In Proceedings of the 45th International Conference on Software Engineering (Melbourne, Victoria, Australia) (ICSE '23). IEEE Press, 919–931. doi:10.1109/ICSE48619.2023.00085

[39] Yu Liu, Jiyang Zhang, Pengyu Nie, Milos Gligoric, and Owolabi Legunsen. 2023. More Precise Regression Test Selection via Reasoning about Semantics-Modifying Changes. In Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2023). Association for Computing Machinery, New York, NY, USA, 664–676. doi:10.1145/3597926.3598086

[40] Andrea Lops, Fedelucio Narducci, Azzurra Ragone, and Michelantonio Trizio. 2024. AgoneTest: Automated creation and assessment of Unit tests leveraging Large Language Models. In Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering (Sacramento, CA, USA) (ASE '24). Association for Computing Machinery, New York, NY, USA, 2440–2441. doi:10.1145/3691620.3695318

[41] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: multi-objective automated testing for Android applications. In Proceedings of the 25th International Symposium on Software Testing and Analysis (Saarbrücken, Germany) (ISSTA 2016). Association for Computing

Just Accepted

Machinery, New York, NY, USA, 94–105. doi:10.1145/2931037.2931054
[42] Antonio Mastropaolo, Nathan Cooper, David Nader Palacio, Simone Scalabrino, Denys Poshyvanyk, Rocco Oliveto, and Gabriele Bavota. 2022. Using transfer learning for code-related tasks. IEEE Transactions on Software Engineering 49, 4 (2022), 1580–1598.
[43] Shane McIntosh and Yasutaka Kamei. 2018. Are Fix-Inducing Changes a Moving Target? A Longitudinal Case Study of Just-In-Time Defect Prediction. IEEE Transactions on Software Engineering 44, 5 (2018), 412–428. doi:10.1145/3180155.3182514
[44] Ayse Tosun Misirli, Emad Shihab, and Yasukata Kamei. 2016. Studying High Impact Fix-Inducing Changes. Empirical Software Engineering Journal 21, 2 (2016), 605–641. doi:10.1007/s10664-015-9370-z
[45] Audris Mockus and David M. Weiss. 2000. Predicting Risk of Software Change. Bell Labs Technical Journal 5, 2 (2000), 169–180. doi:10.1002/bltj.2229
[46] Yusuf Sulistyo Nugroho, Hideaki Hata, and Kenichi Matsumoto. 2020. How different are different diff algorithms in Git? Use–histogram for code changes. Empirical Software Engineering 25 (2020), 790–823. doi:10.1007/s10664-019-09772-z
[47] Ahmet Okutan and Olcay Taner Yıldız. 2014. Software defect prediction using Bayesian networks. Empirical Softw. Engg. 19, 1 (feb 2014), 154–181. doi:10.1007/s10664-012-9218-8
[48] Wendkuuni C. Ouedraogo, Kader Kabore, Haoye Tian, Yewei Song, Anil Koyuncu, Jacques Klein, David Lo, and Tegawende F. Bissyande. 2024. LLMs and Prompting for Unit Test Generation: A Large-Scale Evaluation. In Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering (Sacramento, CA, USA) (ASE '24). Association for Computing Machinery, New York, NY, USA, 2464–2465. doi:10.1145/3691620.3695330
[49] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. 2007. Feedback-Directed Random Test Generation. In 29th International Conference on Software Engineering (ICSE'07). 75–84. doi:10.1109/ICSE.2007.37
[50] Rongqi Pan, Taher A. Ghaleb, and Lionel Briand. 2023. ATM: Black-box Test Case Minimization based on Test Code Similarity and Evolutionary Search. In 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE). 1700–1711. doi:10.1109/ICSE48619.2023.00146
[51] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. 2015. Reformulating Branch Coverage as a Many-Objective Optimization Problem. In 2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST). 1–10. doi:10.1109/ICST.2015.7102604
[52] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. 2018. Automated Test Case Generation as a Many-Objective Optimisation Problem with Dynamic Selection of the Targets. IEEE Transactions on Software Engineering 44, 2 (2018), 122–158. doi:10.1109/TSE.2017.2663435
[53] Fabiano Pecorelli, Giovanni Grano, Fabio Palomba, Harald C. Gall, and Andrea De Lucia. 2024. Toward granular search-based automatic unit test case generation. Empirical Softw. Engg. 29, 4 (May 2024), 49 pages. doi:10.1007/s10664-024-10451-x
[54] Anjana Perera. 2020. Using Defect Prediction to Improve the Bug Detection Capability of Search-Based Software Testing. In 2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE). 1170–1174. doi:10.1145/3324884.3415286
[55] Anjana Perera, Aldeida Aleti, Marcel Böhme, and Burak Turhan. 2020. Defect Prediction Guided Search-Based Software Testing. In 2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE). 448–460. doi:10.1145/3324884.3416612
[56] Anjana Perera, Aldeida Aleti, Burak Turhan, and Marcel Böhme. 2023. An Experimental Assessment of Using Theoretical Defect Predictors to Guide Search-Based Software Testing. IEEE Transactions on Software Engineering 49, 1 (2023), 131–146. doi:10.1109/TSE.2022.3147008
[57] Anjana Perera, Burak Turhan, Aldeida Aleti, and Marcel Böhme. 2024. On the Impact of Lower Recall and Precision in Defect Prediction for Guiding Search-based Software Testing. ACM Trans. Softw. Eng. Methodol. 33, 6 (2024). doi:10.1145/3655022
[58] Anjana Perera, Burak Turhan, Aldeida Aleti, and Marcel Böhme. 2024. On the Impact of Lower Recall and Precision in Defect Prediction for Guiding Search-based Software Testing. ACM Trans. Softw. Eng. Methodol. 33, 6, Article 144 (June 2024), 27 pages. doi:10.1145/3655022
[59] Saif Ur Rehman Khan, Sai Peck Lee, Nadeem Javaid, and Wadood Abdul. 2018. A Systematic Review on Test Suite Reduction: Approaches, Experiment's Quality Evaluation, and Guidelines. IEEE Access 6 (2018), 11816–11841. doi:10.1109/ACCESS.2018.2809600
[60] Christophe Rezk, Yasutaka Kamei, and Shane McIntosh. 2022. The Ghost Commit Problem When Identifying Fix-Inducing Changes: An Empirical Study of Apache Projects. IEEE Transactions on Software Engineering 48, 9 (2022), 3297–3309. doi:10.1109/TSE.2021.3087419
[61] Gema Rodríguez-Pérez, Meiyappan Nagappan, and Gregorio Robles. 2022. Watch Out for Extrinsic Bugs! A Case Study of Their Impact in Just-In-Time Bug Prediction Models on the OpenStack Project. IEEE Transactions on Software Engineering 48, 4 (2022), 1400–1416. doi:10.1109/TSE.2020.3021380
[62] José Miguel Rojas, Mattia Vivanti, Andrea Arcuri, and Gordon Fraser. 2017. A detailed investigation of the effectiveness of whole test suite generation. Empirical Softw. Engg. 22, 2 (apr 2017), 852–893. doi:10.1007/s10664-015-9424-2
[63] Jeanine Romano, Jeffrey D Kromrey, Jesse Coraggio, and Jeff Skowronek. 2006. Appropriate statistics for ordinal level data: Should we really be using t-test and Cohen'sd for evaluating group differences on the NSSE and other surveys. In annual meeting of the Florida Association of Institutional Research, Vol. 177.
[64] Christoffer Rosen, Ben Grawi, and Emad Shihab. 2015. Commit Guru: analytics and risk prediction of software commits. In International Symposium on the Foundations of Software Engineering. 966–969. doi:10.1145/2786805.2803183

Just Accepted

[65] Hareem Sahar, Abdul Ali Bangash, Abram Hindle, and Denilson Barbosa. 2024. IRJIT: A simple, online, information retrieval approach for just-in-time software defect prediction. Empirical Softw. Engg. 29, 5 (Aug. 2024), 34 pages. doi:10.1007/s10664-024-10514-z
[66] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. 2005. When Do Changes Induce Fixes? ACM SIGSOFT Software Engineering Notes 30, 4 (2005), 1–5. doi:10.1145/1082983.1083147
[67] Liyan Song, Shuxian Li, Leandro L. Minku, and Xin Yao. 2022. A Novel Data Stream Learning Approach to Tackle One-Sided Label Noise From Verification Latency. In International Joint Conference on Neural Networks (IJCNN). 1–8. doi:10.1109/IJCNN55064.2022.9891911
[68] Liyan Song, Leandro Minku, Cong Teng, and Yao Xin. 2023. Artifact and Data for "A Practical Human Labeling Method for Online Just-in-Time Software Defect Prediction". doi:10.5281/zenodo.8272293
[69] Liyan Song and Leandro L. Minku. 2023. A Procedure to Continuously Evaluate Predictive Performance of Just-In-Time Software Defect Prediction Models During Software Development. IEEE Transactions on Software Engineering 49, 2 (2023), 646–666. doi:10.1109/TSE.2022.3158831
[70] Liyan Song, Leandro L. Minku, Cong Teng, and Xin Yao. 2023. A Practical Human Labeling Method for Online Just-In-Time Software Defect Prediction. In ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 605–617. doi:10.1145/3611643.3616307
[71] Liyan Song, Leandro L. Minku, and Xin Yao. 2023. On the Validity of Retrospective Predictive Performance Evaluation Procedures in Just-In-Time Software Defect Prediction. Empirical Software Engineering 28, 5 (2023), 1–33. doi:10.1007/s10664-023-10341-8
[72] Vinicius M.A. Souza, Diego F. Silva, Gustavo E.A.P.A. Batista, and João Gama. 2015. Classification of Evolving Data Streams with Infinitely Delayed Labels. In International Conference on Machine Learning and Applications (ICMLA). 214–219. doi:10.1109/ICMLA.2015.174
[73] Sadia Tabassum, Leandro L. Minku, Danyi Feng, George G. Cabral, and Liyan Song. 2020. An Investigation of Cross-Project Learning in Online Just-in-Time Software Defect Prediction. In International Conference on Software Engineering. 554–565. doi:10.1145/3377811.3380403
[74] Ming Tan, Lin Tan, Sashank Dara, and Caleb Mayeux. 2015. Online Defect Prediction for Imbalanced Data. In International Conference on Software Engineering. 99–108. doi:10.1109/ICSE.2015.139
[75] Yutian Tang, Zhijie Liu, Zhichao Zhou, and Xiapu Luo. 2024. ChatGPT vs SBST: A Comparative Assessment of Unit Test Suite Generation. IEEE Trans. Softw. Eng. 50, 6 (June 2024), 1340–1359. doi:10.1109/TSE.2024.3382365
[76] Paolo Tonella. 2004. Evolutionary testing of classes. ACM SIGSOFT Software Engineering Notes 29, 4 (2004), 119–128. doi:10.1145/1013886.1007528
[77] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, and Neel Sundaresan. 2022. Generating accurate assert statements for unit test cases using pretrained transformers. In Proceedings of the 3rd ACM/IEEE International Conference on Automation of Software Test (Pittsburgh, Pennsylvania) (AST '22). Association for Computing Machinery, New York, NY, USA, 54–64. doi:10.1145/3524481.3527220
[78] Zhiyuan Wan, Xin Xia, Ahmed E. Hassan, David Lo, Jianwei Yin, and Xiaohu Yang. 2020. Perceptions, Expectations, and Challenges in Defect Prediction. IEEE Transactions on Software Engineering 46, 11 (2020), 1241–1266. doi:10.1109/TSE.2018.2877678
[79] Hao Wang, Weiyuan Zhuang, and Xiaofang Zhang. 2021. Software Defect Prediction Based on Gated Hierarchical LSTMs. IEEE Transactions on Reliability 70, 2 (2021), 711–727. doi:10.1109/TR.2020.3047396
[80] Zejun Wang, Kaibo Liu, Ge Li, and Zhi Jin. 2024. HITS: High-coverage LLM-based Unit Test Generation via Method Slicing. In Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering (Sacramento, CA, USA) (ASE '24). Association for Computing Machinery, New York, NY, USA, 1258–1268. doi:10.1145/3691620.3695501
[81] Cody Watson, Michele Tufano, Kevin Moran, Gabriele Bavota, and Denys Poshyvanyk. 2020. On learning meaningful assert statements for unit test cases. In Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (Seoul, South Korea) (ICSE '20). Association for Computing Machinery, New York, NY, USA, 1398–1409. doi:10.1145/3377811.3380429
[82] Peng Xiao, Bin Liu, and Shihai Wang. 2018. Feedback-based integrated prediction: Defect prediction based on feedback from software testing process. J. Syst. Softw. 143 (2018), 159–171. https://api.semanticscholar.org/CorpusID:49695154
[83] Lin Yang, Chen Yang, Shutao Gao, Weijing Wang, Bo Wang, Qihao Zhu, Xiao Chu, Jianyi Zhou, Guangtai Liang, Qianxiang Wang, and Junjie Chen. 2024. On the Evaluation of Large Language Models in Unit Test Generation. In Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering (Sacramento, CA, USA) (ASE '24). Association for Computing Machinery, New York, NY, USA, 1607–1619. doi:10.1145/3691620.3695529
[84] Yifan Zhao, Dan Hao, and Lu Zhang. 2023. Revisiting Machine Learning based Test Case Prioritization for Continuous Integration. In 2023 IEEE International Conference on Software Maintenance and Evolution (ICSME). 232–244. doi:10.1109/ICSME58846.2023.00032
[85] Xin Zhou, DongGyun Han, and David Lo. 2024. Bridging expert knowledge with deep learning techniques for just-in-time defect prediction. Empirical Softw. Engg. 30, 1 (Dec. 2024), 44 pages. doi:10.1007/s10664-024-10591-0
[86] Zhichao Zhou, Yuming Zhou, Chunrong Fang, Zhenyu Chen, Xiapu Luo, Jingzhu He, and Yutian Tang. 2024. Coverage Goal Selector for Combining Multiple Criteria in Search-Based Unit Test Generation. IEEE Trans. Softw. Eng. 50, 4 (April 2024), 854–883. doi:10.1109/TSE.2024.3366613

Just Accepted