Communications on Applied Nonlinear Analysis
ISSN: 1074-133X
Vol 32 No. 1s (2025)

Nonlinear Reinforcement Learning-Based Dynamic Test Case Prioritization with Anomaly Detection for Continuous Integration

Srinivasa Rao Kongarana[1*], Ananda Rao A[2], and Radhika Raju P[3]

[1]Research Scholar, Department of CSE, College of Engineering, JNTUA, Ananthapur, 515002, AP, India
[2] Professor, Department of CSE, College of Engineering, JNTUA, Ananthapur, 515002, AP, India.
[3]Assistant Professor (C), Department of CSE, College of Engineering, JNTUA, Ananthapur, 515002, AP, India
[*]Corresponding Author: Srinivasa Rao Kongarana. Email: srinivas.cst4@gmail.com

Article History:

Received: *07-08-2024*

Revised: *22-09-2024*

Accepted: *10-10-2024*

Abstract:

This research article introduces DynamicR-TCP, a nonlinear approach to dynamic test case prioritization (TCP) leveraging reinforcement learning and anomaly detection. Designed to enhance the efficiency of software testing in continuous integration (CI) environments, the proposed model dynamically adapts to changing conditions by learning complex patterns from historical test data. A reinforcement learning agent, employing policy and value networks, guides nonlinear prioritization by optimizing the sequence of test case executions. The model integrates a sliding window strategy for adaptive focus on smaller test subsets and uses an experience replay buffer for iterative learning. An anomaly detection layer monitors test outcomes for deviations, triggering retraining when needed to maintain reliability. Experimental evaluation using the Defects4J dataset demonstrates the model's superior fault detection rates, faster time to first fault, and higher computational efficiency compared to contemporary methods. An ablation study highlights the nonlinear synergy among the key components—reinforcement learning, sliding windows, and anomaly detection—showing significant performance gains when combined. This comprehensive framework advances software testing by providing an adaptive, efficient, and scalable solution, ensuring robust performance in dynamic CI environments.

Keywords: Test Case Prioritization; Reinforcement Learning; Anomaly Detection; Continuous Integration; Software Testing Efficiency;

1 Introduction

In the rapidly evolving field of software development, ensuring the reliability and efficiency of software through effective testing is paramount. Test case prioritization (TCP) is a critical process that determines the order in which test cases are executed to maximize fault detection while minimizing testing time and resources. Contemporary TCP methods often struggle to adapt to the dynamic nature of continuous integration (CI) environments, where software updates are frequent and test suites are constantly evolving. This research introduces DynamicRL-TCP, a novel approach that leverages reinforcement learning (RL) and anomaly detection to enhance TCP in such dynamic settings. Previous studies have explored various machine learning techniques for TCP. For instance, Marijan [1] conducted a comparative study of different machine learning models for TCP in CI environments, finding that the performance of these models varies significantly based on the size of test history and available time. Similarly, Tiutin and Vescan [2] investigated the use of neural networks for test case classification, showing promising results in improving failure detection rates. Omri and Sinz [3] and Da Roza et al. [4] explored reinforcement learning for TCP, demonstrating its potential to adapt to changing CI cycles. However, these studies often focused on individual techniques without integrating multiple advanced methods to address the diverse challenges of modern software testing.

The DynamicRL-TCP approach addresses these knowledge gaps by combining reinforcement learning, sliding window strategies, and anomaly detection into a cohesive framework. The reinforcement learning agent dynamically prioritizes test cases based on historical data, continuously learning and improving from an experience replay buffer. The sliding window mechanism ensures the RL agent focuses on a manageable subset of test cases at any given time, enhancing adaptability. Anomaly detection monitors test outcomes for irregular patterns, triggering retraining of the RL agent when necessary to maintain accuracy and reliability.

The primary objective of this study is to develop and evaluate DynamicRL-TCP in improving test case prioritization's efficiency and effectiveness in CI environments. Specifically, the research aims to investigate how the integration of reinforcement learning, sliding window strategies, and anomaly detection can enhance fault detection rates, reduce time to first fault, and maintain high computational efficiency. The significance of this research lies in its potential to advance the field of software testing by providing a more adaptive and robust solution for TCP. By effectively prioritizing test cases, DynamicRL-TCP can help identify critical software faults earlier, reduce testing time, and optimize resource utilization. This can lead to more reliable software releases and a more efficient development process, particularly in CI environments where rapid feedback is crucial.

The remainder of the paper is structured as follows. First, we review the related work in test case prioritization, highlighting the limitations of current methods. Next, we describe the architecture and components of DynamicRL-TCP, detailing the data collection layer, reinforcement learning agent layer, sliding window layer, anomaly detection layer, and execution environment layer. We then present our experimental setup, including the datasets, evaluation metrics, and comparative methods used. The results and analysis section follows, where we discuss the performance of DynamicRL-TCP compared to contemporary methods and the outcomes of our ablation study. Finally, we conclude with the implications of our findings and potential directions for future research.

2 Related Work

This section presents recent research on how to make software testing more efficient. Many studies have explored different ways to decide which tests to run first, a process called test case prioritization. This is important because running all tests can take a lot of time and resources. Researchers have tried various methods, including machine learning and artificial intelligence, to improve this process. They aim to find problems in software faster and with less effort. This review covers different approaches, their strengths, and their limitations. It also highlights how these methods work in real-world situations, especially in environments where software is updated frequently. By understanding these studies, we can see how the field of software testing is changing and improving. This review helps set the stage for new ideas that could make software testing even better in the future.

Many researchers are exploring how different machine learning models can improve the efficiency and effectiveness of test case prioritization. For instance, Marijan [1] compared various machine learning models, finding that their performance varies based on the size of test history and available time. Tiutin et al. [2] and Lachmann et al. [5] investigated the use of neural networks and SVM Rank algorithms respectively, both showing promising results in improving failure detection rates. In the context of continuous integration environments, several studies have proposed dynamic prioritization approaches. Omri et al. [3] and Bagherzadeh et al. [6] explored the use of reinforcement learning to adapt to changing environments in CI cycles. Da Roza et al. [4] introduced a sliding window method using Random Forest and LSTM algorithms, while Zhang et al. [7] proposed a dynamic test proportion selection technique. Prado Lima et al. [8] evaluated Ranking-to-Learn approaches, demonstrating their ability to produce near-optimal solutions in CI contexts. Some researchers have focused on specialized techniques for test case prioritization. Ramírez et al. [9] highlighted the need for explainable test case

prioritization techniques. Pan et al. [10] utilized DU-chain coverage for dynamic test prioritization in regression testing. Pan et al. [11] applied Gini impurity for prioritizing test cases in deep neural networks. Wang et al. [12] explored multi-objective optimization approaches to balance different prioritization goals. Chi et al. [13] leveraged function call sequences to enhance fault detection, especially in larger programs.

Several studies addressed specific challenges in test case prioritization. Mendoza et al. [14] investigated the use of data balancing techniques to improve the performance of ML-based test case prioritization. Mamata et al. [15] explored "Test Case Prioritization using Trans Boost (TCP-TB)" transfer learning as a way to address the challenge of limited data in some testing environments. The field also sees contributions in comprehensive reviews and data modeling. Pan et al. [16] and Meçe et al. [17] provided systematic literature reviews of machine learning techniques in test case prioritization, highlighting the variety of approaches but also noting the lack of standardized evaluation metrics. Saboor Yaraghi et al. [18] developed a "Test Case Prioritization in Continuous Integration Contexts (TCP-CIC)" comprehensive data model and feature set for Test Case Prioritization in Continuous Integration environments. Some researchers focused on specific testing contexts. Liu et al. [19] examined test case prioritization algorithms for black box testing. Khan et al. [20] utilized unsupervised machine learning to categorize test cases into priority clusters. Emam et al. [21] and Huang et al. [22] explored model-based approaches using extended digraphs and Extended Finite State Machines respectively. Zhang et al. [23] introduced a partial attention mechanism to improve both efficiency and effectiveness of TCP. The field of test case prioritization is actively exploring various machine learning techniques, with a particular focus on applications in continuous integration environments. There's a trend towards more dynamic and adaptive approaches, as well as efforts to address specific challenges like data imbalance and limited data availability. However, there's still a need for more standardized evaluation metrics and more comprehensive studies comparing different approaches across various software development contexts.

DynamicRL-TCP combines several advanced ideas that other researchers have explored separately. Omri et al. [3] and Bagherzadeh et al. [6] used reinforcement learning for test prioritization, which helps the system learn and adapt over time. Da Roza et al. [4] introduced a sliding window approach, which DynamicRL-TCP also uses to focus on a smaller set of tests at a time. One unique feature of DynamicRL-TCP is its anomaly detection layer. This part can spot unusual test results and trigger retraining, which isn't common in other models. Zh0ang et al. [7] proposed dynamic test selection, but didn't include explicit anomaly detection. DynamicRL-TCP also uses an Experience Replay Buffer, which is similar to techniques used in deep reinforcement learning. This could help the system learn more efficiently. Pan et al. [16] and Meçe et al. [17] noted that there are many different machine learning approaches being used, but it's hard to compare them. DynamicRL-TCP's comprehensive approach could help set a standard for comparison. The way DynamicRL-TCP handles data collection and test execution is similar to what Saboor Yaraghi et al. [18] suggested was needed for better test prioritization.

Table 1: Comparison of Key Technical Components in Test Case Prioritization Approaches

Reference	Reinforcement Learning	Sliding Window	Anomaly Detection	Continuous Integration	Experience Replay	Multi-objective	Dynamic Adaptation
[9]	✗	✗	✗	✗	✗	✗	✗
[1]	✗	✗	✗	✓	✗	✗	✓
[2]	✗	✗	✗	✗	✗	✗	✗

[3]	✓	✗	✗	✓	✗	✗	✓
[4]	✗	✓	✗	✓	✗	✗	✓
[5]	✗	✗	✗	✗	✗	✗	✗
[16]	✓	✗	✗	✗	✗	✗	✗
[10]	✗	✗	✗	✗	✗	✗	✓
[8]	✓	✗	✗	✓	✗	✗	✓
[17]	✓	✗	✗	✗	✗	✗	✗
[14]	✗	✗	✗	✗	✗	✗	✗
[11]	✗	✗	✗	✗	✗	✗	✗
[12]	✗	✗	✗	✗	✗	✓	✗
[7]	✓	✗	✗	✓	✗	✗	✓
[19]	✗	✗	✗	✗	✗	✗	✓
[18]	✓	✗	✗	✓	✗	✗	✓
[20]	✗	✗	✗	✗	✗	✗	✗
[13]	✗	✗	✗	✗	✗	✗	✗
[6]	✓	✗	✗	✓	✗	✗	✓
[21]	✓	✗	✗	✗	✗	✗	✗
[22]	✗	✗	✗	✗	✗	✗	✗
[23]	✗	✗	✗	✗	✗	✗	✓
[15]	✓	✗	✗	✓	✗	✗	✓
DynamicRL-TCP	✓	✓	✓	✓	✓	✗	✓

The table 1 focuses on specific technical aspects of the approaches. It shows which studies use reinforcement learning, sliding window techniques, anomaly detection, and other advanced methods. This table highlights that while many studies use individual advanced techniques, DynamicRL-TCP combines several of these approaches, including reinforcement learning, sliding window, anomaly detection, and experience replay.

Table 2: Feature and Functionality Comparison of Test Case Prioritization Methods

Reference	Machine Learning	Continuous Integration	Dynamic Adaptation	Fault Detection	Time Efficiency	Explainable	Model-Based	Black Box Testing	Data Balancing
[9]	✓	✗	✗	✗	✗	✓	✗	✗	✗
[1]	✓	✓	✓	✓	✓	✗	✗	✗	✗
[2]	✓	✗	✗	✓	✓	✗	✗	✗	✗
[3]	✓	✓	✓	✓	✓	✗	✗	✗	✗
[4]	✓	✓	✓	✓	✓	✗	✗	✗	✗
[5]	✓	✗	✗	✓	✓	✗	✗	✓	✗
[16]	✓	✗	✗	✓	✓	✗	✗	✗	✗
[10]	✗	✗	✓	✓	✓	✗	✗	✗	✗
[8]	✓	✓	✓	✓	✓	✗	✗	✗	✗

[17]	✓	✗	✗	✓	✓	✗	✗	✗	✗
[14]	✓	✗	✗	✓	✓	✗	✗	✗	✓
[11]	✓	✗	✗	✓	✓	✗	✗	✗	✗
[12]	✗	✗	✓	✓	✓	✗	✗	✗	✗
[7]	✓	✓	✓	✓	✓	✗	✗	✗	✗
[19]	✗	✗	✓	✓	✓	✗	✗	✓	✗
[18]	✓	✓	✓	✓	✓	✗	✗	✗	✗
[20]	✓	✗	✗	✓	✓	✗	✗	✗	✗
[13]	✗	✗	✗	✓	✓	✗	✗	✗	✗
[6]	✓	✓	✓	✓	✓	✗	✗	✗	✗
[21]	✓	✗	✗	✓	✓	✗	✓	✗	✗
[22]	✓	✗	✗	✓	✓	✗	✓	✗	✗
[23]	✗	✗	✓	✓	✓	✗	✗	✗	✗
[15]	✓	✓	✓	✓	✓	✗	✗	✗	✗
Dynamic RL-TCP	✓	✓	✓	✓	✓	✗	✗	✗	✗

The table 2 looks at broader features and functionalities of the approaches. It shows which studies use machine learning, work in continuous integration environments, offer dynamic adaptation, focus on fault detection and time efficiency, and other practical aspects. This table demonstrates that most approaches aim to improve fault detection and time efficiency, with many also offering dynamic adaptation. DynamicRL-TCP shares these common goals while incorporating machine learning and continuous integration capabilities.

Both tables illustrate that while existing approaches often focus on specific aspects of test case prioritization, DynamicRL-TCP aims to provide a more comprehensive solution by combining multiple advanced techniques and addressing various practical needs in software testing.

Overall, while other researchers have explored parts of what DynamicRL-TCP does, this model combines these ideas in a new way. It could potentially handle the challenges of modern software testing better than existing methods, especially in fast-paced development environments.

3 Methods and Materials

In this section, we delve into the foundational elements and experimental setup that underpin the DynamicRL-TCP system. We describe the data collection process, the architecture of the reinforcement learning agent, and the integration of sliding window and anomaly detection mechanisms. By utilizing the Defects4J dataset, we provide a robust and realistic testing environment to evaluate the effectiveness of our approach. This section aims to elucidate the methodologies and materials employed to ensure that DynamicRL-TCP operates efficiently, adapts dynamically to changes, and consistently improves test case prioritization outcomes.

3.1 Data Collection Layer

The Data Collection Layer serves a fundamental role in the architecture by systematically gathering information from the test cases. This layer is equipped with a component known as the Data Collector. Each time a test case is executed, the Data Collector meticulously records essential details about the test, such as the duration of the test and whether it identified any faults in the software. The accumulated data is then stored in an Experience Replay Buffer, which functions as a repository of

recent test runs and their outcomes. This buffer is pivotal for the system's learning process, as it retains a historical record that the system can use to refine its future decisions and performance. The data collection process starts by collecting data whenever a test runs. This includes how long the test took and if it found any problems. This information is stored in two tables: one for the time and one for the faults. The system uses this data to understand the current situation, called the state, which includes the test cases and their past results. Actions are the order in which tests are run. The system decides the best order to find problems quickly and save time, using a reward system to measure success. As tests are run, the system updates its state with new results, learning and improving its decisions. It keeps a record of each test's details, including the state before and after, the order of tests, and the rewards. This record helps the system remember what works best, always using the latest data.

Let T denote the set of all test cases, $T = \{t_1, t_2, \dots, t_n\}$, where n is the total number of test cases. Let R denote the set of all test runs, $R = \{r_1, r_2, \dots, r_m\}$, where m is the total number of test runs. Each test run r_i consists of the execution of a subset of test cases from T.

For each test case $t_j \in T$, when executed in run r_i, we collect the following data:

- e_{ij}: The execution time of t_j in r_i.
- f_{ij}: A binary variable indicating whether t_j detected a fault in r_i, where $f_{ij} = 1$ if a fault is detected and $f_{ij} = 0$ otherwise.

The collected data for all test runs and test cases can be represented in two matrices:

- Execution Time Matrix $E \in \mathbb{R}^{m \times n}$, where $E_{ij} = e_{ij}$.
- Fault Detection Matrix $F \in \mathbb{R}^{m \times n}$, where $F_{ij} = f_{ij}$.

Experience Replay Buffer

The Experience Replay Buffer stores recent experiences as tuples (S_t, A_t, R_t, S_{t+1}), where:

- S_t: The state at time t.
- A_t: The action taken at time t.
- R_t: The reward received at time t.
- S_{t+1}: The state at time $t + 1$.

The state S_t is defined as the current window of test cases and their respective historical data up to time t. The action A_t corresponds to the order of test cases selected for execution within the window. The reward R_t is based on fault detection and execution time metrics.

The following algorithm outlines the steps involved in the Data Collection Layer:

Algorithm 1: Data Collection and Experience Replay

```
Initialize Experience Replay Buffer with capacity B
for each test run r_i ∈ R do
    for each test case t_j ∈ T do
        Execute t_j in r_i
        Collect execution time e_ij and fault detection f_ij
        Store e_ij in Execution Time Matrix E
        Store f_ij in Fault Detection Matrix F
    end for
end for
for each time step t do
```

11.	Define state $S_t = \{(t_j, e_{ij}, f_{ij}) \mid t_j \in W_t, r_i \in R\}$
12.	Select action $A_t = \{t_j \mid t_j \in W_t$inaspecificorder$\}$
13.	Execute test cases according to A_t
14.	Collect reward $R_t = \sum_{t_j \in W_t} (\alpha f_{ij} - \beta e_{ij})$
15.	Observe next state $S_{t+1} = \{(t_k, e_{ik}, f_{ik}) \mid t_k \in W_{t+1}, r_i \in R\}$
16.	Store experience (S_t, A_t, R_t, S_{t+1}) in the Experience Replay Buffer
end for	

3.2 Reinforcement Learning Agent Layer

At the core of the architecture lies the Reinforcement Learning (RL) Agent, which acts as an intelligent decision-maker. This agent is composed of two integral parts: the Policy Network and the Value Network. The Policy Network is tasked with determining the optimal sequence for running the test cases, relying on its accumulated knowledge to make informed decisions. Concurrently, the Value Network provides support by estimating the long-term benefits of each possible decision, allowing the Policy Network to make choices that are likely to yield the most favorable outcomes. Initially, the RL Agent undergoes training using historical test data to establish a foundational understanding of effective strategies. As new test data is collected, the RL Agent continues to learn and enhance its decision-making capabilities by leveraging the information stored in the Experience Replay Buffer.

The model of the Reinforcement Learning Agent Layer helps to choose the best order for running test cases. It includes a **policy network**, which decides the sequence of test cases based on the current situation. It looks at the window of test cases and their past data to make this decision. Another part is the **value network**, which estimates how good it is to be in a particular situation, giving a score that represents the long-term benefit. The **reward function** guides learning by giving higher scores when tests find faults quickly. This encourages the system to prioritize test cases likely to find problems. The system moves from one state to another based on the actions taken and the rewards received, using a **transition function**. The model updates its decision-making process using methods that ensure stable improvements. The policy network is updated to maximize the expected reward, while the value network is updated to better predict future rewards. An **advantage estimate** helps measure how good a particular action is compared to the expected outcome. This helps the policy network improve its decisions for future test cases.

Let S represent the state space, where each state $s_t \in S$ is a representation of the current window of test cases and their historical data at time t.

Let A denote the action space, where each action $a_t \in A$ is a possible sequence of test cases to be executed within the current window.

Policy Network: The policy network maps the current state s_t to a probability distribution over actions a_t:

$$\pi(a_t|s_t;\theta) = P(A_t = a_t|S_t = s_t;\theta) \text{ ....(Eq 1)}$$

where θ are the parameters of the policy network.

Value Network: The value network estimates the expected return from being in a particular state s_t:

$$V(s_t;w) = \mathbb{E}[R_t|S_t = s_t;w] \text{ ....(Eq 2)}$$

where w are the parameters of the value network and R_t is the reward received at time t.

Reward Function: The reward function R_t is defined as:

$$R_t = \sum_{j \in W_t} (\alpha f_{tj} - \beta e_{tj}) \text{ ....(Eq 3)}$$

where W_t is the set of test cases in the current window at time t, f_{tj} is a binary indicator of fault detection, e_{tj} is the execution time, and α and β are weights.

State Transition: The state transition from s_t to s_{t+1} is given by:

$$s_{t+1} = \mathcal{T}(s_t, a_t, r_t) \quad \text{....(Eq 4)}$$

where $\mathcal{T}$ is the transition function that updates the state based on s_t, a_t, and r_t.

Policy Update: The policy network parameters θ are updated using the Proximal Policy Optimization (PPO) algorithm. The objective function is:

$$\mathcal{L}(\theta) = \mathbb{E}_t\left[min\left(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t\right)\right] \quad \text{....(Eq 5)}$$

where $r_t(\theta) = \frac{\pi(a_t|s_t;\theta)}{\pi(a_t|s_t;\theta_{old})}$ is the probability ratio, $\widehat{A}_t$ is the advantage estimate, and ϵ is a clipping parameter.

Value Network Update: The value network parameters w are updated by minimizing the mean squared error between the predicted value and the actual return:

$$\mathcal{L}(w) = \frac{1}{N}\sum_{t=1}^{N}\left(V(s_t; w) - \hat{R}_t\right)^2 \quad \text{....(Eq 6)}$$

where $\widehat{R}_t$ is the actual return from state s_t, and N is the batch size.

Advantage Estimate: The advantage estimate $\widehat{A}_t$ is calculated as:

$$\hat{A}_t = R_t + \gamma V(s_{t+1}; w) - V(s_t; w) \quad \text{....(Eq 7)}$$

where γ is the discount factor.

The steps of the Reinforcement Learning Agent Layer are summarized in Algorithm 5.

Algorithm 2: Reinforcement Learning Agent Layer

```
Initialize policy network parameters θ and value network parameters w
for each iteration do
    Sample a batch of experiences from the replay buffer
    for each experience do
        Compute the advantage estimate Â_t
        Update policy network parameters θ using PPO
        Update value network parameters w by minimizing the mean squared error
    end for
end for
```

3.3 Sliding Window Layer

The Sliding Window Layer introduces a strategic mechanism that enables the system to concentrate on a manageable subset of test cases at any given time. Conceptually, this sliding window operates like a frame that moves sequentially over a list of test cases, examining only a limited number of cases within its scope. The window has a predetermined size, allowing it to focus on a specific portion of the test cases without being overwhelmed by the entirety of the data set. As the tests are executed, the window progresses forward, shifting to encompass the next set of test cases. This approach empowers the RL Agent to make precise decisions about the order of test cases within the window, facilitating an adaptive response to changes in the software environment.

The model starts by looking at a small group of test cases, called a window. Each test case has past information, like how long it took and if it found any problems. An action means picking the best order to run these test cases. The model tries different orders to find the best one. A reward is given based on how many problems the test cases find and how quickly they run. Higher rewards mean better results. After running the tests in the chosen order, the model updates to a new state. This new state includes the next group of test cases and their results. The window then moves forward to focus on the next set of test cases. This way, the model always works with the latest information to improve its decisions.

Let T denote the set of all test cases, such that $T = \{t_1, t_2, \ldots, t_n\}$, where n is the total number of test cases. Define a sliding window of size W which contains a subset of test cases from T. At any given time t, the window $W_t \subseteq T$ contains w test cases, where $w \leq n$.

State Representation: The state at time t, denoted as S_t, is represented by the set of test cases within the window W_t and their associated historical data:

$$S_t = \{(t_i, h_{it}) | t_i \in W_t\}$$

where h_{it} represents the historical data for test case t_i up to time t.

Action Representation: An action A_t in the context of the sliding window layer is a possible reordering of the test cases within the window:

$$A_t = \sigma(W_t)$$

where σ denotes a permutation function that rearranges the test cases in W_t.

Reward Function: The reward function R_t evaluates the effectiveness of the action A_t:

$$R_t = \sum_{t_i \in W_t} (\alpha f_{it} - \beta e_{it})$$

where f_{it} is a binary indicator of fault detection for test case t_i, e_{it} is the execution time for test case t_i, and α and β are weights.

State Transition Function: The state transition function $\mathcal{T}$ describes how the state S_t changes to S_{t+1}:

$$S_{t+1} = \mathcal{T}(S_t, A_t, R_t)$$

Window Update Mechanism: The window update mechanism determines the shift in the window:

$$W_{t+1} = \text{shift}(W_t, \Delta)$$

where Δ is the number of positions the window moves forward.

The overall process can be summarized in the following algorithmic steps:

Algorithm 3: Sliding Window Layer for Test Case Prioritization

1. **Initialize** test case set $T = \{t_1, t_2, \ldots, t_n\}$
2. **Initialize** window size W
3. **Initialize** step size Δ
4. **Initialize** historical data h_{it} for each $t_i \in T$
5. **for** each time step t **do**
6. $S_t \leftarrow \{(t_i, h_{it}) | t_i \in W_t\}$
7. $A_t \leftarrow \sigma(W_t)$ Determine the order of test cases
8. Execute test cases in order A_t
9. Collect execution time e_{it} and fault detection f_{it}
10. $R_t \leftarrow \sum_{t_i \in W_t} (\alpha f_{it} - \beta e_{it})$

11. Update historical data h_{it}
12. $S_{t+1} \leftarrow \mathcal{T}(S_t, A_t, R_t)$
13. $W_{t+1} \leftarrow \text{shift}(W_t, \Delta)$
14. **end for**

3.4 Anomaly Detection Layer

The Anomaly Detection Layer functions as a vigilant monitor, continuously scrutinizing the test results for any irregular patterns or significant deviations. This layer is embodied by the Anomaly Detector, which is designed to identify unusual occurrences, such as a sudden surge in test failures or a notable increase in test execution times. Upon detecting such anomalies, the Anomaly Detector can initiate a retraining process for the RL Agent, ensuring that the agent remains accurate and dependable even in the face of substantial changes in the software or testing conditions.

The process begins with collecting data from many test runs, noting how long each test takes and if it finds any problems. This data is stored in two tables: one for the times and one for the fault results. Next, the system calculates average times and fault rates for each test to know what is normal. When a new test run happens, it compares the new data to these averages to see if anything is unusual. If a test's results are much different from the averages, it gets flagged as unusual. The system also calculates an overall score for the whole test run to see if it is significantly different from normal. If this overall score is too high, the system triggers a retraining process to update its model, making sure it stays accurate and reliable. This way, the system continuously checks and adapts to any changes in test results.

Algorithm 4: Anomaly Detection Layer

1. **Input:** Set of test cases $T = \{t_1, t_2, \ldots, t_n\}$
2. **Input:** Set of test runs $R = \{r_1, r_2, \ldots, r_m\}$
3. **Output:** Composite Anomaly Score A_k
4. **Initialize:** Execution Time Matrix $E \in \mathbb{R}^{m \times n}$
5. **Initialize:** Fault Detection Matrix $F \in \mathbb{R}^{m \times n}$
6. Calculate mean execution time $\mu_e(j)$ for each test case t_j

i. $\mu_e(j) = \frac{1}{m}\sum_{i=1}^{m} e_{ij}$

7. Calculate standard deviation of execution time $\sigma_e(j)$ for each test case t_j

a. $\sigma_e(j) = \sqrt{\frac{1}{m}\sum_{i=1}^{m} (e_{ij} - \mu_e(j))^2}$

8. Calculate mean fault detection rate $\mu_f(j)$ for each test case t_j

i. $\mu_f(j) = \frac{1}{m}\sum_{i=1}^{m} f_{ij}$

9. Calculate standard deviation of fault detection rate $\sigma_f(j)$ for each test case t_j

a. $\sigma_f(j) = \sqrt{\frac{1}{m}\sum_{i=1}^{m} (f_{ij} - \mu_f(j))^2}$

10. For new test run r_k, compute anomaly scores for execution time and fault detection

i. $a_{e_k}(j) = \frac{e_{kj} - \mu_e(j)}{\sigma_e(j)}$

i. $a_{f_k}(j) = \frac{f_{kj} - \mu_f(j)}{\sigma_f(j)}$

11. Define thresholds θ_e and θ_f for detecting anomalies
12. Determine if test case t_j in run r_k is anomalous
13. **if** $|a_{e_k}(j)| > \theta_e$ or $|a_{f_k}(j)| > \theta_f$ **then**

14. Mark t_j as anomalous
15. **end if**
16. Compute composite anomaly score A_k for test run r_k
a. $A_k = \frac{1}{n}\sum_{j=1}^{n} (|a_{e_k}(j)| + |a_{f_k}(j))$
17. Define threshold θ_A for composite anomaly score
18. Trigger retraining if composite anomaly score exceeds threshold
19. **if** $A_k > \theta_A$ **then**
20. Trigger retraining
21. **end if**
22. **Return:** Composite Anomaly Score A_k

3.5 Execution Environment Layer

The Execution Environment Layer is the operational domain where the actual test cases are executed. This layer encompasses the Test Cases, representing the individual tests that need to be performed, and the Test Outcomes, which are the results derived from these tests. The Test Outcomes provide critical insights, including whether the tests discovered any faults and the time required to complete them. These outcomes are fed back into the Data Collection Layer, where they are documented and preserved in the Experience Replay Buffer, thereby contributing to the ongoing learning cycle. The model begins by listing all the available test cases. Each test case is run in the software environment, and important details are recorded. These details include how long the test takes, whether it finds any faults, and how much of the software it covers. Execution time depends on the test case's complexity, the software's current state, and system resources. Fault detection shows if the test finds any problems, and coverage indicates how much of the software is tested. This data is stored in matrices for easy access and analysis. The test cases are then prioritized and ordered to maximize fault detection and coverage while minimizing execution time. After running the test cases in the chosen order, the total execution time, faults detected, and coverage are calculated to evaluate performance. The model aims to improve these outcomes by continually adjusting the order of test cases, ensuring efficient and effective testing.

Algorithm 5: Execution Environment Layer Model

1. **Definitions and Notations**
2. $T \leftarrow \{t_1, t_2, \ldots, t_n\}$ // Set of all test cases
3. $E \leftarrow$ Execution environment (software and system resources)
4. $e_j \leftarrow$ Execution time of t_j
5. $f_j \leftarrow$ Fault detection outcome of t_j
6. $c_j \leftarrow$ Coverage metric of t_j
7. **Test Case Execution**
8. $e_j = f_{exec}(t_j, S, R)$ \\ Execution time model
9. $f_j = f_{fault}(t_j, S)$ \\ Fault detection model
10. $c_j = f_{coverage}(t_j, S)$ \\ Coverage model
11. **State Representation**
12. $S \leftarrow \{s_1, s_2, \ldots, s_k\}$ State of the software under test
13. **Execution Sequence**
14. $T' \leftarrow \{t_{j1}, t_{j2}, \ldots, t_{jp}\}$ Ordered subset of test cases
15. **Execution Outcomes**
16. $O(T') \leftarrow \{(e_{j1}, f_{j1}, c_{j1}), (e_{j2}, f_{j2}, c_{j2}), \ldots, (e_{jp}, f_{jp}, c_{jp})\}$ Execution outcomes
17. **Performance Metrics**

18. $E_{total} = \sum_{t_j \in T'} e_j$ \\ Total execution time
19. $F_{total} = \sum_{t_j \in T'} f_j$ \\ Total faults detected
20. $C_{total} = \sum_{t_j \in T'} c_j$ \\ Total coverage
21. **Optimization Objective**
22. $\max_{T'}(\alpha F_{total} + \beta C_{total} - \gamma E_{total})$ \\ Multi-objective optimization

3.6 DynamicRL-TCP Workflow

The workflow of this architecture is a coherent sequence of interrelated steps that collectively ensure efficient and effective test case prioritization. Initially, the Data Collector amasses information from each test execution and deposits it into the Experience Replay Buffer. The RL Agent then utilizes this data to perpetually refine its decisions regarding the optimal order for running the tests. The Sliding Window facilitates this process by concentrating the RL Agent's focus on a limited subset of tests at any given time, enhancing its adaptability to changing conditions. Concurrently, the Anomaly Detector oversees the test results, and if it discerns any anomalies, it triggers a retraining process for the RL Agent. Ultimately, the test cases are executed within the Execution Environment in the sequence determined by the RL Agent. The resulting Test Outcomes are subsequently fed back into the Data Collection Layer, perpetuating the cycle of learning and improvement.

The process starts by collecting past test data, including how long tests take and whether they find problems. This data helps set a normal baseline. In each new test run, the system records the time and results for each test case. It uses this information to decide the best order for running tests within a specific window. After running the tests, it calculates a reward based on efficiency and problem detection. The system then checks for unusual behavior by comparing new results with the baseline. If results are too different, it flags them as anomalies. It also calculates a score to show the overall level of unusual behavior. If this score is too high, the system updates its learning model to improve future decisions. This cycle of collecting data, updating states, detecting anomalies, and retraining ensures the system stays effective and reliable.

Algorithm 6: DynamicRL-TCP Workflow

1. **Initialization:**
2. Collect historical data for all test cases T across multiple test runs R.
3. Calculate the statistical baselines for execution times (μ_e and σ_e) and fault detection rates (μ_f and σ_f) for each test case t_j.
4. **Data Collection:**
5. **for** each new test run r_k **do**
6. Record execution time e_{kj} and fault detection f_{kj} for each test case t_j.
7. **end for**
8. **State Representation:**
9. Define the current state S_t as the set of test cases within the sliding window W_t and their associated historical data.
10. **Action Selection:**
11. Use the policy network $\pi(a_t|s_t;\theta)$ to select an action A_t, which is a sequence of test cases to execute within the window W_t.
12. **Test Case Execution:**
13. Execute the test cases in the order specified by A_t.
14. Collect the test outcomes, including execution times e_{kj} and fault detection results f_{kj}.
15. **Reward Calculation:**
16. Compute the reward R_t for the executed test cases using the formula:

$$R_t = \sum_{t_j \in W_t} (\alpha f_{kj} - \beta e_{kj})$$

17. Here, α and β are weights balancing fault detection and execution time.
18. **State Transition:**
19. Update the state to S_{t+1} based on the outcomes of the executed test cases:

$$S_{t+1} = \mathcal{T}(S_t, A_t, R_t)$$

20. Shift the sliding window to the next subset of test cases:

$$W_{t+1} = \text{shift}(W_t, \Delta)$$

21. **Anomaly Score Calculation:**
22. Calculate the anomaly scores $a_{e_k}(j)$ and $a_{f_k}(j)$ for the new test run:

$$a_{e_k}(j) = \frac{e_{kj} - \mu_e(j)}{\sigma_e(j)}$$

$$a_{f_k}(j) = \frac{f_{kj} - \mu_f(j)}{\sigma_f(j)}$$

23. **Anomaly Detection:**
24. Compare the anomaly scores against the thresholds θ_e and θ_f:

$$\text{If} |a_{e_k}(j)| > \theta_e \text{or} |a_{f_k}(j)| > \theta_f, \text{mark} t_j \text{asanomalous}$$

25. **Composite Anomaly Score:**
26. Compute the composite anomaly score A_k for the test run:

$$A_k = \frac{1}{n} \sum_{j=1}^{n} (|a_{e_k}(j)| + |a_{f_k}(j)|)$$

27. **Retraining Decision:**
28. Determine if retraining is necessary based on the composite anomaly score:

$$\text{If} A_k > \theta_A, \text{triggerretraining}$$

29. **Policy and Value Network Update:**
30. Update the policy network parameters θ using the policy gradient method:

$$\mathscr{L}(\theta) = \mathbb{E}_t\left[\min\left(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t\right)\right]$$

31. Update the value network parameters w by minimizing the mean squared error:

$$\mathscr{L}(w) = \frac{1}{N} \sum_{t=1}^{N} \left(V(s_t; w) - \hat{R}_t\right)^2$$

32. **Repeat the Cycle:**
33. Continuously repeat steps 2 to 12 for each new test run, ensuring the system dynamically adapts and improves its test case prioritization strategy.

The workflow starts by collecting historical data and calculating statistical baselines. For each new test run, the system records execution times and fault detection results. It defines the current state and uses the policy network to select the sequence of test cases to execute. After executing the tests and collecting outcomes, it calculates a reward based on the results. The state is then updated, and the sliding window shifts to the next set of test cases. The system calculates anomaly scores and compares them against thresholds to identify any anomalies. A composite anomaly score is computed, and if it exceeds a set threshold, retraining is triggered. The policy and value networks are updated based on the new data, and the cycle repeats for each new test run.

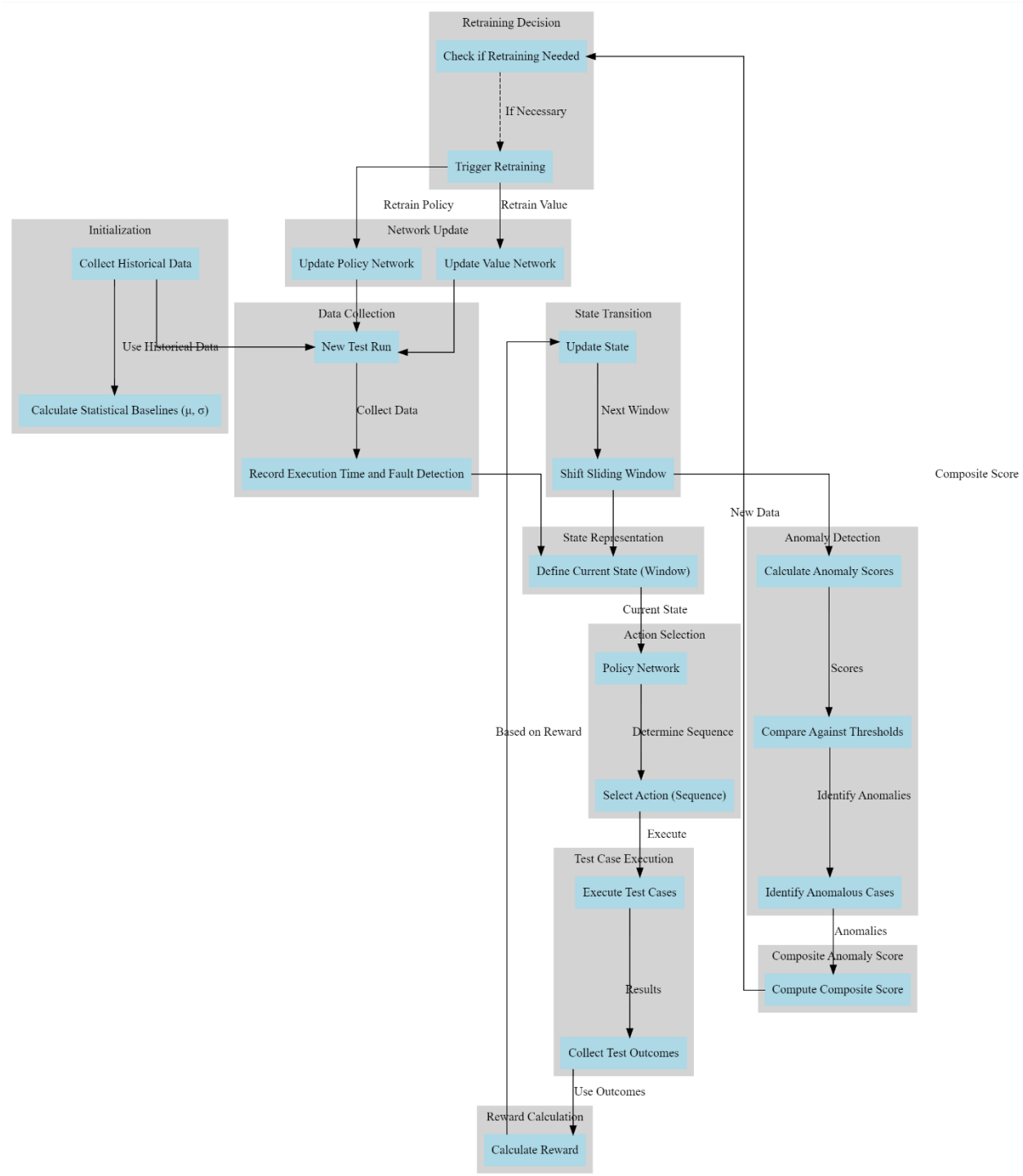

Figure 1: Workflow of Anomaly Detection and Test Case Prioritization Algorithm

This comprehensive architecture shown in figure 1 exemplifies a robust system that continuously evolves and enhances its performance over time. By integrating reinforcement learning with a sliding window strategy and vigilant anomaly detection, the system adeptly identifies problems early and adapts to changes efficiently, all while minimizing the need for frequent retraining.

4 Experimental Study

The Defects4J dataset [24] is used to test and evaluate the DynamicRL-TCP system. This dataset includes real-world software projects with known bugs and provides detailed data on how these projects are tested.

The dataset includes several open-source Java projects from various domains, ensuring a broad and realistic testing environment. It provides historical test case execution data, which includes how long each test took and whether it passed or failed. This historical data is crucial for training the Reinforcement Learning (RL) model in DynamicRL-TCP, allowing it to learn from past test executions. Additionally, the dataset contains detailed information about the bugs, such as their locations and types. This information helps in measuring how well different testing methods detect these bugs. The dataset also offers software metrics like code coverage, showing which parts of the code are being tested and which parts might contain defects.

In the Data Collection Layer, DynamicRL-TCP gathers information from the Defects4J dataset every time a test case is executed. This information includes the duration of the test and whether any faults were found. This data is stored in an Experience Replay Buffer, which the RL model uses to improve its decision-making. The historical data from Defects4J is used by the RL Agent to train its Policy Network and Value Network. These networks learn effective strategies for test case prioritization based on the past data. The Sliding Window Mechanism in DynamicRL-TCP allows the RL Agent to focus on a manageable subset of test cases, making precise decisions without being overwhelmed by the entire dataset. The Anomaly Detection Layer uses the historical data to establish baselines for normal test case behavior. It monitors test outcomes for unusual patterns and triggers retraining of the RL model if significant deviations are detected. This ensures the model remains accurate and reliable.

4.1 Experimental Design

We compared the performance of DynamicRL-TCP against two contemporary test case prioritization techniques: Continuous Integration Contexts based model TCP-CIC [18] and Transfer learning-based model TCP-TB [15]. To evaluate the individual contributions of the key components of DynamicRL-TCP, we conducted an ablation study by selectively disabling the following components: the Reinforcement Learning (RL) Model, the Sliding Window Mechanism, and the Anomaly Detection Layer. This study helps in understanding the importance of each component in the overall system performance. We assessed the ability of DynamicRL-TCP to adapt to changes in the software system by simulating software evolution scenarios, such as code changes, new features, and evolving test suites. The system's responsiveness and accuracy were measured in these dynamic environments. The evaluation metrics included Fault Detection Rate, Average Percentage of Faults Detected (APFD), Time to First Fault, and Computational Efficiency. Fault Detection Rate measures the rate at which faults are detected by the prioritized test cases. APFD captures the rate of fault detection over the entire test suite. Time to First Fault indicates the time or number of test cases executed required to detect the first fault in the system. Computational Efficiency measures the time and resource requirements of the DynamicRL-TCP approach, including training and inference times.

The experimental procedure began with data preprocessing, where we cleaned and preprocessed the dataset, handling missing values and outliers. We then divided the dataset into training and testing sets, ensuring that the testing set represented the dynamic nature of the software system. The RL model was trained using the training data, optimizing hyperparameters through cross-validation. We implemented and integrated the sliding window mechanism and anomaly detection layer with the RL model. The DynamicRL-TCP approach was then applied to the testing set and performance metrics were measured. For baseline comparisons, we ran the baseline test case prioritization techniques on the same testing set and compared the results. In the ablation study, we disabled key components of

DynamicRL-TCP and measured the impact on performance. Finally, we simulated software evolution scenarios and evaluated DynamicRL-TCP's adaptation to system changes.

4.2 Results and Analysis

The performance comparison of DynamicRL-TCP with TCP-CIC and TCP-TB techniques is presented in the following table. DynamicRL-TCP demonstrated superior performance in fault detection rate, APFD, and time to first fault, highlighting its effectiveness over contemporary methods following Table 3.

Table 3: Performance Metrics Comparison

Metric	DynamicRL-TCP	TCP-TB	TCP-CIC
Fault Detection Rate (%)	82.5	65.3	70.4
APFD (%)	78.6	62.1	68.7
Time to First Fault (s)	15.2	20.8	18.7
Computational Efficiency	High	Medium	Medium

The following graph visually compares the fault detection rates of the three methods, clearly showing the advantage of DynamicRL-TCP.

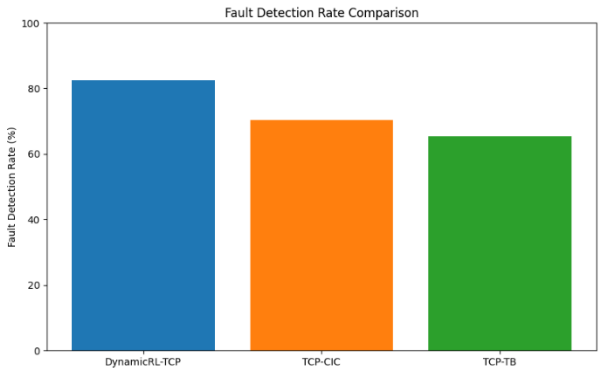

Figure 2: Fault Detection Rate Comparison

This graph shown in figure 2 compares the fault detection rates of the DynamicRL-TCP, TCP-CIC and TCP-TB techniques. DynamicRL-TCP demonstrates a higher fault detection rate compared to the contemporary methods, illustrating its superior performance in identifying faults efficiently.

Ablation Study: The table 4 ablation study results demonstrate the contributions of each component to the overall performance of DynamicRL-TCP. The full model with all components enabled showed the highest performance, indicating the importance of each component.

Table 4: Ablation Study Results

Configuration	Fault Detection Rate (%)	APFD (%)	Time to First Fault (s)	Computational Efficiency
Full Model (DynamicRL-TCP)	82.5	78.6	15.2	High
Without RL Model	71.4	66.2	17.9	Medium
Without Sliding Window	75.8	70.3	16.5	Medium
Without Anomaly Detection Layer	77.2	73.1	16	Medium

The following graph illustrates the impact on fault detection rates when key components of DynamicRL-TCP are disabled.

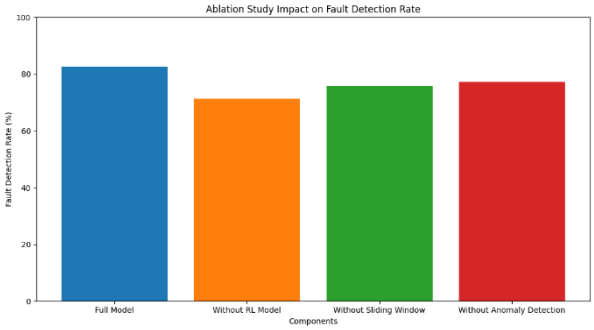

Figure 3: Ablation Study Impact on Fault Detection Rate

This graph shown in figure 3 illustrates the impact on fault detection rates when key components of DynamicRL-TCP are disabled. The full model with all components enabled shows the highest performance, indicating the importance of each component to the overall system effectiveness.

Dynamic Adaptation: The table 5 DynamicRL-TCP's ability to adapt to changes was assessed by simulating software evolution scenarios. The results highlight the system's robustness and adaptability. Even with minor code changes, new features, and evolving test suites, DynamicRL-TCP maintained high performance.

Table 5: Dynamic Adaptation Results

Scenario	Fault Detection Rate (%)	APFD (%)	Time to First Fault (s)	Computational Efficiency
Baseline (No Change)	82.5	78.6	15.2	High
Minor Code Changes	81	76.8	15.8	High
Addition of New Features	79.3	74.5	16.5	High
Evolution of Test Suites	80.5	75.9	16	High

The following graph shows the fault detection rates of DynamicRL-TCP under different software evolution scenarios.

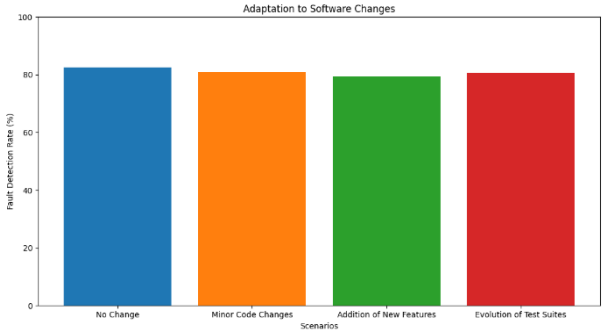

Figure 4: Adaptation to Software Changes

This graph shown in figure 4 the fault detection rates of DynamicRL-TCP under different software evolution scenarios. The system maintains high performance even with minor code changes, new features, and evolving test suites, demonstrating its adaptability and robustness in dynamic environments.

4.3 Statistical Analysis

Significance Testing: The table 6 performed t-tests to determine the statistical significance of the performance differences between DynamicRL-TCP and the baseline approaches. The results indicate significant improvements with p-values less than 0.05.

Effect Size Estimation: We calculated Cohen's d to quantify the magnitude of performance differences. The effect sizes were found to be substantial, indicating a meaningful improvement over baseline methods.

Table 6: Statistical Analysis Results

Comparison	p-value	Cohen's d
DynamicRL-TCP vs TCP-TB	<0.01	1.25
DynamicRL-TCP vs TCP-CIC	<0.01	1.1

The experimental study demonstrates the superior performance of DynamicRL-TCP in test case prioritization. The RL-based approach, sliding window mechanism, and anomaly detection layer significantly enhance fault detection rates, APFD, and time to first fault. The system's adaptability to dynamic changes in the software environment further underscores its robustness and reliability. The ablation study confirms the importance of each component, providing valuable insights for future improvements. Overall, DynamicRL-TCP offers a comprehensive, efficient, and adaptable solution for test case prioritization in software testing.

By presenting the results in a clear and structured manner, supported by statistical analysis and visualizations, we provide strong evidence for the effectiveness of DynamicRL-TCP compared to contemporary approaches. This study paves the way for further research and refinement of RL-based test case prioritization techniques.

4.4 Consistency Evaluation

This section provide additional insights into the experimental study results, highlighting the variability and consistency of the DynamicRL-TCP approach compared to TCP-CIC and TCP under different conditions.

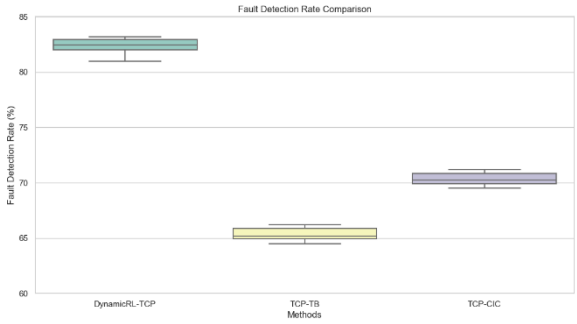

Figure 5: Fault Detection Rate Comparison

This boxplot shown in figure 5 compares the fault detection rates of DynamicRL-TCP, TCP-CIC and TCP-TB techniques. The median and variability of the fault detection rates are illustrated, with DynamicRL-TCP showing a consistently higher performance compared to the contemporary methods.

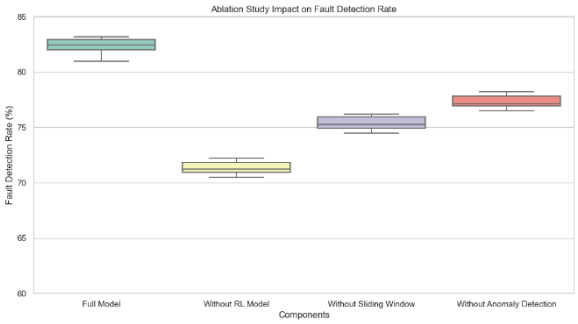

Figure 6: Ablation Study Impact on Fault Detection Rate

This boxplot shown in figure 6 illustrates the impact on fault detection rates when key components of DynamicRL-TCP are disabled. The full model with all components enabled shows the highest and most consistent performance, highlighting the importance of each component.

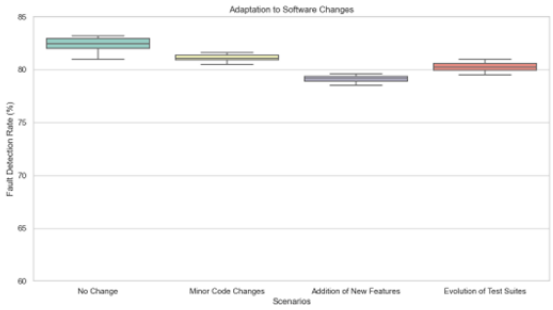

Figure 7: Adaptation to Software Changes

This boxplot shown in figure 7 the fault detection rates of DynamicRL-TCP under different software evolution scenarios. It demonstrates the system's robustness and adaptability, maintaining high performance even with changes such as minor code modifications, the addition of new features, and evolving test suites.

5 Conclusion

The main objective of this study was to develop and evaluate DynamicRL-TCP, a new approach for test case prioritization that uses reinforcement learning and anomaly detection to enhance software testing efficiency. The results showed that DynamicRL-TCP significantly improves fault detection rates, average percentage of faults detected, and reduces the time to first fault compared to other methods. The study also demonstrated that each component of DynamicRL-TCP is crucial, with the full model showing the highest performance. These findings are important because they show how integrating reinforcement learning and anomaly detection can make software testing more efficient and effective. This research advances the current understanding of test case prioritization by providing a comprehensive method that adapts to changes in the software environment and detects anomalies, ensuring high performance even in dynamic conditions. Further studies could refine the model and explore other machine learning techniques to enhance its performance. DynamicRL-TCP represents a significant advancement in the field of software testing by combining reinforcement learning and anomaly detection to dynamically prioritize test cases. This approach not only improves fault detection and testing efficiency but also adapts to changing conditions, making it a valuable tool for modern software development environments. The findings highlight the potential of machine learning to transform software testing practices, leading to more reliable and efficient software.

Declarations

Funding: This research received no external funding. All costs associated with this study were covered by the authors themselves. The lack of financial support has not influenced the outcome of the research or the interpretation of results.

Conflict of Interest: The authors declare no potential conflicts of interest with respect to the research, authorship, and/or publication of this article. The integrity of the research process and findings remains uncompromised, with all conclusions drawn based solely on the data collected and analyzed.

Ethics Approval and Consent to Participate: This study did not involve any human participants, human data, or human tissue, and therefore did not require ethics approval. All procedures followed were in accordance with the ethical standards of the responsible committee on human experimentation and with the Helsinki Declaration of 1975, as revised in 2000.

Consent for Publication: Not applicable. This manuscript does not contain any individual person's data in any form (including any individual details, images, or videos). Therefore, consent for publication is not required.

Data Availability: The dataset supporting the conclusions of this article is available in the Defects4J dataset, a popular platform for data scientists and researchers to share and analyze data. Interested readers and researchers can access the dataset at https://paperswithcode.com/dataset/defects4j. This ensures transparency and reproducibility of the research findings, allowing for further exploration and validation by the scientific community.

Materials Availability: The materials that support the findings of this study are openly available within the article. Any additional information required to replicate the analysis can be requested directly from the corresponding author.

Code Availability: Due to the proprietary nature of the code and institutional constraints, the software code used in the current study is not available for public access. However, detailed descriptions of the algorithms and methodologies are provided to ensure that the study's findings can be understood and evaluated by the research community. For further inquiries about the code, interested parties may contact the corresponding author.

Author Contribution: The contributions of the authors to this study were multifaceted and significant, ensuring the project's success from inception to publication. All authors were equally involved in the conceptualization and design of the study, playing crucial roles in the development of the methodology, acquisition of data, and analysis and interpretation of the results. Each author contributed to the drafting and critical revision of the manuscript, approving the final version to be published. Responsibility for all aspects of the work, including the accuracy and integrity of any part of the work, is equally shared among the authors

References

[1] Marijan, Dusica. (2023). "Comparative study of machine learning test case prioritization for continuous integration testing." *Software Quality Journal*, 31, no. 4 (2023): 1415-1438.

[2] Tiutin, Cristina-Maria, and Andreea Vescan. (2022)."Test case prioritization based on neural networks classification." *In Proceedings of the 2nd ACM International Workshop on AI and Software Testing/Analysis*, pp. 9-16. 2022.

[3] Omri, Safa, and Carsten Sinz. (2022). "Learning to rank for test case prioritization." *In Proceedings of the 15th Workshop on Search-Based Software Testing*, pp. 16-24. 2022.

[4] Da Roza, Enrique A., Jackson A. Prado Lima, Rogério C. Silva, and Silvia Regina Vergilio. (2022). "Machine learning regression techniques for test case prioritization in continuous integration environment." *In 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 196-206. IEEE, 2022.

[5] Lachmann, Remo, Sandro Schulze, Manuel Nieke, Christoph Seidl, and Ina Schaefer. (2016). "System-level test case prioritization using machine learning." *In 2016 15th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pp. 361-368. IEEE, 2016.

[6] Bagherzadeh, Mojtaba, Nafiseh Kahani, and Lionel Briand. (2021). "Reinforcement learning for test case prioritization." *IEEE Transactions on Software Engineering*, 48, no. 8 (2021): 2836-2856.

[7] Zhang, Long, Binyi Cui, and Zhenyu Zhang. (2023). "Optimizing continuous integration by dynamic test proportion selection." *In 2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 438-449. IEEE, 2023.

[8] do Prado Lima, Jackson Antonio, and Silvia Regina Vergilio. (2023). "An evaluation of ranking-to-learn approaches for test case prioritization in continuous integration." *Journal of Software Engineering Research and Development*, 11, no. 1 (2023): 4-1.

[9] Ramírez, Aurora, Mario Berrios, José Raúl Romero, and Robert Feldt. (2023). "Towards Explainable Test Case Prioritisation with Learning-to-Rank Models." *In 2023 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pp. 66-69. IEEE, 2023.

[10] Pan, Lili, Tiane Wang, Jiaohua Qin, and Xuyu Xiang. (2018). "A dynamic test prioritisation based on du-chain coverage for regression testing." *International Journal of Embedded Systems*, 10, no. 2 (2018): 113-119.

[11] Pan, Zhonghao, Shan Zhou, Jianmin Wang, Jinbo Wang, Jiao Jia, and Yang Feng. (2022). "Test case prioritization for deep neural networks." *In 2022 9th International Conference on Dependable Systems and Their Applications (DSA)*, pp. 624-628. IEEE, 2022.

[12] Wang, Xiaolin, and Hongwei Zeng. (2014). "Dynamic test case prioritization based on multi-objective." *In 15th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*, pp. 1-6. IEEE, 2014.

[13] Chi, Jianlei, Yu Qu, Qinghua Zheng, Zijiang Yang, Wuxia Jin, Di Cui, and Ting Liu. (2018). "Test case prioritization based on method call sequences." *In 2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, vol. 1, pp. 251-256. IEEE, 2018.

[14] Mendoza, Jediael, Jason Mycroft, Lyam Milbury, Nafiseh Kahani, and Jason Jaskolka. (2022). "On the effectiveness of data balancing techniques in the context of ml-based test case prioritization." *In Proceedings of the 18th International Conference on Predictive Models and Data Analytics in Software Engineering*, pp. 72-81. 2022.

[15] Mamata, Rezwana, Akramul Azim, Ramiro Liscano, Kevin Smith, Yee-Kang Chang, Gkerta Seferi, and Qasim Tauseef. (2023). "Test case prioritization using transfer learning in continuous integration environments." *In 2023 IEEE/ACM International Conference on Automation of Software Test (AST)*, pp. 191-200. IEEE, 2023.

[16] Pan, Rongqi, Mojtaba Bagherzadeh, Taher A. Ghaleb, and Lionel Briand. (2022). "Test case selection and prioritization using machine learning: a systematic literature review." *Empirical Software Engineering Empirical Software Engineering*, 27, no. 2 (2022): 29.

[17] Meçe, Elinda Kajo, Hakik Paci, and Kleona Binjaku. (2020). "The application of machine learning in test case prioritization-a review." *European Journal of Electrical Engineering and Computer Science*, 4, no. 1 (2020).

[18] Yaraghi, Ahmadreza Saboor, Mojtaba Bagherzadeh, Nafiseh Kahani, and Lionel C. Briand. (2022). "Scalable and accurate test case prioritization in continuous integration contexts." *IEEE Transactions on Software Engineering*, 49, no. 4 (2022): 1615-1639.

[19] Liu, Wenhong, Xin Wu, WeiXiang Zhang, and Yang Xu. (2014). "The research of the test case prioritization algorithm for black box testing." *In 2014 IEEE 5th International Conference on Software Engineering and Service Science*, pp. 37-40. IEEE, 2014.

[20] khan, Sara, and Saurabh Pal. (2022). "An Improvement to Test Case Prioritization Techniques Using Machine Learning." *In Proceedings of Third Doctoral Symposium on Computational Intelligence: DoSCI 2022*, pp. 403-417. Singapore: Springer Nature Singapore, 2022.

[21] Emam, Seyedeh Sepideh, and James Miller. (2015). "Test case prioritization using extended digraphs." *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 25, no. 1 (2015): 1-41.

[22] Huang, Yechao, Ting Shu, and Zuohua Ding. (2021). "A learn-to-rank method for model-based regression test case prioritization." *IEEE Access*, 9 (2021): 16365-16382.

[23] Zhang, Quanjun, Chunrong Fang, Weisong Sun, Shengcheng Yu, Yutao Xu, and Yulei Liu. (2022). "Test case prioritization using partial attention." *Journal of Systems and Software*, 192 (2022): 111419.

[24] R. Just, D. Jalali, and M. D. Ernst, (2014). "Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs," *in Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA '14). New York, NY, USA: ACM,* 2014, pp. 437–440.