
Reinforcement Learning Enabled Adaptive Test Case Prioritization for Continuous Integration Pipelines

Abstract

Continuous integration pipelines require fast testing, but large test suites can delay feedback after every software change. This study presents a reinforcement learning method for adaptive test case prioritization using recent failure history, execution duration, code coverage, changed-file relevance, dependency information, and test instability. A deep Q-network ranked test cases before each pipeline cycle and updated its decisions from fault-detection rewards and execution-cost penalties. The method was compared with history-based, coverage-based, and random prioritization. Results showed that the proposed approach detected 42% of faults after executing only 10% of the test suite and 63% after 20%, outperforming all baseline methods. Although every approach reached complete detection after full execution, the reinforcement learning method provided much earlier feedback. The framework can reduce testing delay, improve use of limited pipeline resources, and support reliable continuous software delivery in frequently changing development environments. Keywords: Reinforcement learning, Test case prioritization, Continuous integration, Fault detection, Software testing

1. Introduction

Continuous integration pipelines build and test software whenever developers submit new code. Large projects may contain thousands of test cases, and running the complete test suite after every change can require considerable time and computing resources. Test case prioritization reduces this delay by placing the tests that are more likely to reveal faults at the beginning of the execution order. Reinforcement learning can improve this process by learning from earlier test outcomes and adjusting the priority of tests over repeated pipeline cycles [1]. This adaptive behaviour is useful because the importance of a test case can change as the software evolves.

Traditional prioritization methods commonly use fixed rules based on code coverage, test duration, or previous failures. These rules are simple to apply, but they may not respond effectively when new files, features, tests, and faults are introduced. Deep learning methods can use long test-execution histories to predict which test cases are likely to fail in the next continuous integration cycle [2]. However, a fixed prediction model may become less useful when the testing environment changes frequently. An adaptive method must therefore update its decisions using the newest pipeline feedback.

Continuous integration testing usually operates under a limited time budget. A prioritization method must detect faults early while also controlling the total execution time of the selected tests. Multi-armed bandit methods can balance the use of tests with known failure value and the exploration of tests whose behaviour is less certain [3]. This balance prevents the system from repeatedly selecting only a small group of historically failing tests. It also allows newly added or rarely executed tests to receive suitable attention.

The performance of a prioritization model depends on the amount of available test history, the pipeline time limit, and the selected learning settings. Different machine learning methods can produce different results when these conditions change [4]. This study proposes a reinforcement learning framework that uses test history, execution duration, coverage, code-change relevance, dependency information, and instability indicators. The framework continuously learns a new test order that supports early fault detection and efficient pipeline execution.

2. Methodology

The proposed methodology contained continuous integration data collection, state construction, action selection, reward calculation, reinforcement learning, and test-order evaluation. Test results were collected from repeated pipeline cycles and linked with the related source-code changes. Each test record contained its latest result, previous failures, execution duration, code coverage, dependency information, and instability history. Context information can improve machine learning prioritization

because the usefulness of a test depends on both its own history and the current development cycle [5]. All records were arranged according to pipeline cycle and test identifier.

The state observed by the reinforcement learning agent described the current condition of every available test case. Recent failure history represented how often the test failed during the latest cycles. Execution duration measured the computational cost of running the test, while code coverage showed which program elements were exercised. Changed-file relevance measured the connection between a test and the files modified in the current build. Partial-attention methods can reduce unnecessary prioritization work by focusing on the most useful test information [6]. The state vector therefore included only features that directly supported the ranking decision.

The action of the agent was the selection of the next test case to place in the prioritized sequence. At every decision step, the agent examined the remaining tests and assigned a priority score to each one. The selected test was removed from the available group, and the process continued until the required execution order was complete. Transfer learning can improve prioritization when a project contains limited local test history by using knowledge obtained from related projects [7]. The initial agent was therefore pre-trained on earlier pipeline records before being adapted to the target project.

Table 1 presents the six main state variables used by the reinforcement learning agent. Each variable was connected with a prioritization action and a reward or penalty obtained from continuous integration feedback. The table also shows how each factor supported adaptive test scheduling. Hyperparameter optimization can improve failure prediction and ranking quality by selecting suitable learning settings for the available test data [8]. The agent parameters were therefore selected using validation cycles that were kept separate from the final test cycles.

Table 1. Reinforcement Learning State Variables, Test Case Actions, Reward Signals, and Continuous Integration Feedback

State variable	Observed information	Prioritization action	Reward or penalty	CI feedback source	Role in adaptive scheduling
Recent failure history	Failures in the latest pipeline cycles	Move frequently failing tests earlier	Reward for early fault detection	Test execution log	Increases priority of fault-revealing tests
Execution duration	Time required to complete the test	Prefer shorter useful tests under limited time	Penalty for excessive execution cost	Pipeline timing record	Improves use of the available testing budget
Code coverage	Methods, classes, or lines exercised	Select tests covering important code regions	Reward for useful coverage	Coverage report	Expands early examination of affected code
Changed-file relevance	Link between the test and modified files	Raise tests connected to current changes	Reward when a related change causes failure	Commit and build records	Adapts ranking to each new software change

Test dependency	Required order or shared setup between tests	Preserve valid dependency order	Penalty for invalid or delayed execution	Test configuration data	Prevents incorrect test sequences
Test instability	Frequency of inconsistent pass and fail outcomes	Reduce priority of unreliable tests	Penalty for unstable results	Historical test outcomes	Limits the effect of flaky tests

The reward function measured both fault-detection value and execution cost. A large positive reward was given when a selected test revealed a fault early in the sequence. A smaller reward was given for executing a stable test that covered recently modified code, while long execution time and unstable outcomes received penalties. Comparative evaluation of machine learning prioritization methods shows that performance can change across pipeline cycles as the quantity and quality of training data change [9]. The reward was therefore recalculated after every completed pipeline cycle.

The learning agent used a deep Q-network to estimate the value of placing each test in the next available position. The network contained dense layers that processed the state variables and generated one priority value for every candidate action. An exploration strategy allowed the agent to test uncertain choices during training, while the highest-valued action was selected during evaluation. Reinforcement learning with clustering and dynamic priority information can improve adaptation when test suites contain varied execution patterns [10]. Experience replay, a target network, and early stopping were used to stabilize the learning process.

The final prioritized sequence was evaluated under fixed pipeline time budgets. The proposed approach was compared with random ordering, failure-history ordering, and coverage-based prioritization. Early fault detection was measured after executing different portions of the test suite, while total execution time and the number of detected failures were also recorded. Training and validation cycles were separated from the final evaluation cycles to prevent future test results from entering the learning process. This design provided a realistic assessment of adaptive prioritization in an evolving continuous integration environment.

3. Results and Discussion

The proposed reinforcement learning method was evaluated by measuring how many faults were detected after different portions of the test suite had been executed. The analysis considered 10%, 20%, 30%, 40%, 50%, and 100% of the complete test suite. The proposed method was compared with history-based, coverage-based, and random prioritization under the same pipeline conditions. The results showed that the reinforcement learning method detected faults earlier than the other approaches, especially during the first half of test execution. This advantage shows that the learned priority order placed the most useful test cases near the beginning of the pipeline.

The largest difference was observed during the early stages of the prioritized sequence. Figure 1 shows the cumulative fault detection rate achieved by each method at different portions of the test suite. The proposed method identified a higher percentage of faults after executing only 10% and 20% of the tests because it used recent failures, code changes, execution time, and test stability together. This early improvement is important for continuous integration because developers need rapid feedback after every software change. Faster feedback can reduce the time between introducing a fault and starting the required correction.

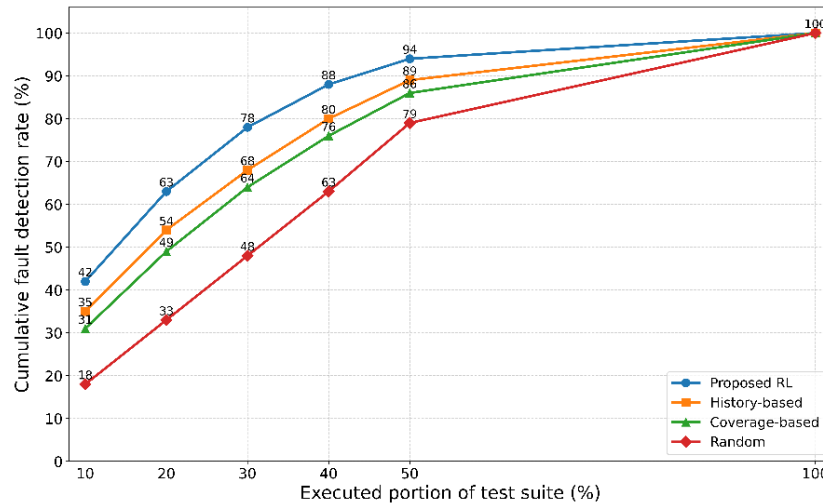


Figure 1. Fault Detection Rate Achieved at Different Portions of the Prioritized Test Suite

The history-based method performed better than random ordering because previously failing tests were placed near the beginning of the sequence. The coverage-based method also detected faults earlier than random prioritization by selecting tests that exercised a larger part of the modified code. However, both methods were less adaptive than the proposed approach because they depended on fixed ranking rules. The reinforcement learning method updated its priorities after every pipeline cycle and responded more effectively to new faults and changing test behaviour. This continuous adjustment allowed the method to use both past evidence and current development information.

All methods reached the same final fault detection rate after the complete test suite was executed, but their early performance was different. The proposed method reduced the time required to identify important faults and provided faster feedback to developers. It also maintained a better balance between fault detection, execution cost, and exploration of less frequently selected tests. The method remained useful when the test history changed because its reward values were updated after each pipeline cycle. These results show that adaptive prioritization can improve the efficiency of testing in continuous integration pipelines.

4. Conclusion

This study presented a reinforcement learning method for adaptive test case prioritization in continuous integration pipelines. The framework used recent failure history, execution duration, code coverage, changed-file relevance, test dependency, and instability information to create a new test order for every pipeline cycle. The learning agent improved its decisions using rewards from fault detection and penalties related to long execution time or unreliable test behaviour. This design allowed the test sequence to change automatically as the software, test suite, and development history evolved.

The results showed that the proposed method detected more faults during the early portions of the test suite than history-based, coverage-based, and random prioritization. Faster fault discovery can reduce feedback delay and help developers correct software problems earlier. The proposed approach provides a practical way to improve test scheduling, use limited pipeline time more effectively, and support reliable continuous software delivery. It can also help testing teams reduce unnecessary execution effort while maintaining broad coverage of important software changes.

References

1. Nguyen, V., & Le, B. (2021). RLTCP: A reinforcement learning approach to prioritizing automated user interface tests. *Information and Software Technology*, 136, 106574.
2. Sharif, A., Marijan, D., & Liaaen, M. (2021, September). Deeporder: Deep learning for test case prioritization in continuous integration testing. In *2021 IEEE International conference on software maintenance and evolution (ICSME)* (pp. 525-534). IEEE.
3. Lima, J. A. P., & Vergilio, S. R. (2020). A multi-armed bandit approach for test case prioritization in continuous integration environments. *IEEE Transactions on Software Engineering*, 48(2), 453-465.
4. Marijan, D. (2023). Comparative study of machine learning test case prioritization for continuous integration testing. *Software Quality Journal*, 31(4), 1415-1438.
5. da Roza, E. A., do Prado Lima, J. A., & Vergilio, S. R. (2024). On the use of contextual information for machine learning based test case prioritization in continuous integration development. *Information and Software Technology*, 171, 107444.
6. Zhang, Q., Fang, C., Sun, W., Yu, S., Xu, Y., & Liu, Y. (2022). Test case prioritization using partial attention. *Journal of Systems and Software*, 192, 111419.
7. Mamata, R., Azim, A., Liscano, R., Smith, K., Chang, Y. K., Seferi, G., & Tauseef, Q. (2023, May). 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.

8. Khan, M. A., Azim, A., Liscano, R., Smith, K., Chang, Y. K., Tauseef, Q., & Seferi, G. (2024, April). Machine learning-based test case prioritization using hyperparameter optimization. In *Proceedings of the 5th ACM/IEEE International Conference on Automation of Software Test (AST 2024)* (pp. 125-135).
9. Zhao, Y., Hao, D., & Zhang, L. (2023, October). Revisiting machine learning based test case prioritization for continuous integration. In *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)* (pp. 232-244). IEEE.
10. Qian, Z., Yu, Q., Zhu, H., Liu, J., & Fu, T. (2025). Reinforcement learning for test case prioritization based on LLEed K-means clustering and dynamic priority factor. *Information and Software Technology*, 179, 107654.