International Journal of Scientific Research & Engineering Trends
Volume 12, Issue 4, July-Aug-2026, ISSN (Online): 2395-566X

AI Based Dynamic Test Case Prioritisation for TestNG Frameworks

Balvir Singh Thakur[1], Devanshi[1]
[1]Department of Computer Science, University Institute of Technology, Himachal Pradesh University, Shimla, India

Abstract — Regression testing is a vital part of software testing. Executing all test cases from large test suites without considering their historical failure behaviour may lead to time-consuming and inefficient execution. Running the test cases that are more prone to failure first can improve this process. In this paper, an AI-based test case prioritisation approach for the automated testing framework based on TestNG using machine learning and historical test execution data is proposed. The proposed approach extracts execution-based features like total test runs, number of failures, failure rate, average execution time and recent failure information and uses a Random Forest classifier to identify high-risk test cases. The predicted risk information is then used to produce an ordered list of the test suite, ordered by priority. The approach was tested on 70 TestNG test cases with 20 previous executions compared to a normal execution order and an alphabetically ordered static baseline using the Average Percentage of Faults Detected (APFD) metric. The experimental results show that the AI-prioritized execution order results in an APFD of 0.7100, compared to 0.5643 for the normal order and 0.5729 for the static priority order, improvements of 25.82% and 23.94%, respectively. The results show that taking into account historical failure behaviour in machine learning-based test prioritisation can improve the early detection of failing test cases and provide a more risk-aware execution strategy for regression testing.

Keywords— Test Case Prioritisation, TestNG, Regression Testing, Machine Learning, Random Forest, APFD, Continuous Integration

I. INTRODUCTION

Software testing is a necessary activity of the software development life cycle, designed to verify that a system works as intended and to identify defects before they reach production. Regression testing is in a special position among all forms of testing: every time new code is added or existing code is modified, regression testing helps to ensure that functionality that was previously working has not been broken. Software systems evolve continuously, especially in agile and DevOps based environments, and regression testing is performed very often, often many times a day.

With the introduction of Continuous Integration and Continuous Deployment (CI/CD) practices, regression testing has already increased in importance and frequency. In the average CI/CD pipeline, each commit results in an automated build and test cycle, and developers expect quick, reliable feedback on whether their changes are safe to merge. However, as the project grows, the regression test suite tends to be much larger, often with hundreds of test cases covering many different functional areas. Running those large suites in entirety, on every commit, can take tens of minutes, or even hours. This is in direct conflict with the goals of comprehensive coverage and rapid feedback to the developer.

Test Case Prioritisation (TCP) is a solution for this problem, which reorders the test execution so that test cases more likely to reveal faults are executed earlier without reducing the total execution time of a full suite. Roughly, traditional TCP approaches can be categorised into coverage-based prioritisation (which needs source code instrumentation), history-based prioritisation (based on past pass/fail results), and manual/static prioritisation (where developers assign fixed priority values based on intuition or domain knowledge). In TestNG suites, static prioritisation is done using the @Test(priority) attribute or, if it is not specified, the default is the order of declaration.

Static and manual strategies are easy to implement, but they are static at design time and do not evolve as the actual failure behaviour of a test suite changes over time. Test cases that were considered high priority today can become stable and rarely fail, while a previously low-priority test can start failing frequently as the feature it exercises becomes fragile. This motivates the main research question of this paper: How can historical data on test executions be used to automatically and dynamically prioritise test cases without manual configuration or static, unchanging rules?

In this paper, we propose a dynamic test case prioritisation system for TestNG based on AI. The system reads historical TestNG XML execution reports, extracts execution based

features (failure rate, average duration and recent failure history), builds a Random Forest classifier to predict the failure risk, and ranks the test executions in descending order of risk score. The approach is evaluated with the APFD metric against normal and static baselines on a TestNG suite of 70 test cases run over 20 historical runs. The key contributions of this work are: (1) a fully implemented pipeline to extract historical execution data directly from TestNG XML reports; (2) a lightweight execution-based feature set that does not require access to source code or coverage instrumentation; (3) a Random Forest-based risk prediction model validated using cross-validation suitable for small, imbalanced datasets; and (4) a rigorous three-way empirical evaluation, including an explicit correction for data leakage in the evaluation methodology.

II. LITERATURE REVIEW

Rothermel et al. [1] performed one of the first and most influential empirical studies on test case prioritisation, establishing the core idea of reordering test cases such that faults can be detected earlier, and evaluating several prioritisation techniques against unordered and random baselines. Do, Rothermel and Kinneer [2] extended this line of research to Java programs tested with JUnit, experimentally demonstrating that prioritisation could significantly improve the rate of fault detection in a JUnit testing environment; as TestNG shares JUnit's annotation-driven, Java-based design, this work is among the most directly applicable foundational references for the present study. Mei et al. [3] proposed a static approach to prioritise JUnit test cases based on the call-graph relationships extracted from the source code rather than dynamic execution information, unlike the current study which is based on dynamic historical behaviour.

There are several approaches based on history and on CI that are similar to the one proposed here. A similar combined usage of failure_rate and avg_duration_ms is seen in the closely related work by Huang and Peng [4] that proposed a history-based, cost-cognisant prioritisation technique that combines historical failure records with test execution cost. Kim, Jeong and Lee [5] proposed FHD-Prioritization, which utilises failed-test history and correlation information to prioritise tests that are likely to fail; the main idea of this study is similar, that is, a test case's failure history is a strong predictor of its future behaviour. Spieker et al. [6] examined history- and diversity-based prioritisation in the setting of CI environments across six open-source projects, and found that prior failure information has strong predictive power even without large volumes of historical data, directly supporting the design of this study's comparatively modest 20-run dataset. Prado Lima et al. [9] performed a systematic mapping study on 35 CI-based TCP studies and concluded that about 80% of them are history-based, providing strong external validation for the motivation of this study. Cho, Kim and Lee [15] proposed a statistical, history-based technique, AFSAC, evaluated on Apache Tomcat and Camel, offering a potential direction to extend the current feature set with failure-correlation information.

The broader context of this work is machine-learning-based TCP research. A systematic review of 29 ML-based test case selection and prioritisation studies published between 2006 and 2020 was performed by Pan, Bagherzadeh, Ghaleb and Briand [8], who mapped the techniques, features and evaluation metrics used in the field, highlighting the increasing role of ML-based approaches in CI environments; this review is one of the most relevant references for the present work. Ferrarini et al. [10] recently argued that, beyond simple failure history, contextual CI and domain information can improve ML-based prioritisation accuracy, creating a research gap for the current study, which uses only execution-based historical features. The evolution of the wider field of regression testing optimisation and the evaluation practices such as the APFD metric used in this study can be seen in several broader surveys: Lou et al. [7], Yoo and Harman [11], Singh et al. [12], Catal and Mishra [13], and Pardha Sagar and Prasad [14].

Most relevant to this work is Kim and Ali [16], who introduced an automation and prioritisation approach for regression testing of the PB Tech web application using Selenium WebDriver, TestNG and the Page Object Model, the same automation stack used in this work. They combined human evaluation with statistical information on features of high-earning websites, and showed a reduction in execution time compared to running the full regression suite. However, their prioritisation relies on human assessment and application-specific statistics rather than dynamically learning the probability of failure from historical TestNG execution data. This is precisely the gap that the present study aims to fill, by replacing human-evaluation-based prioritisation with an automated, machine-learning-driven approach trained directly on historical TestNG execution results.

III PROPOSED METHODOLOGY

The proposed system is implemented as a five-stage pipeline: (1) test suite execution using TestNG/Maven, (2) XML report generation (testng-results.xml), (3) historical dataset construction by parsing repeated XML reports into a structured CSV, (4) feature extraction and Random Forest model training, and (5) failure-risk prediction and prioritised test execution, evaluated using APFD. Each historical run generates a TestNG XML report that logs the name, status (PASS/FAIL/SKIP),

execution time, and start/end timestamps for each test method. A Python parser (testng_parser.py) subsequently appends the data to a cumulative CSV after each run, thereby enabling the historical dataset to expand with every new execution.

The suite under evaluation consists of 70 test cases in eight modules (Login, Dashboard, Admin, PIM, MyInfo, Leave, Recruitment, Time) of an OrangeHRM-style web application implemented using Selenium WebDriver, TestNG and the Page Object Model. The suite was run 20 times to produce the historical dataset, resulting in 1,400 individual execution records. A controlled subset of test cases was designed with non-deterministic (probabilistic) failure behaviour, resembling realistic failure modes observed in real regression suites. This was required to ensure sufficient variation in historical failure data to train the model, considering that a suite that always passes would provide no learnable failure signal for a classifier to learn from.

IV. IMPLEMENTATION

1. Feature Engineering and Fixing Data Leakage

For each test case, five features were calculated and are summarised in Table 1.

Feature	Summary
total_runs	Number of historical runs used to compute features (19, not counting the held-out run)
failure_count	Number of failed runs of the test case in the feature-history runs
failure_rate	Failure rate of the test case across the feature-history runs
avg_duration_ms	Average test case execution time, in milliseconds
recent_failure	Whether the run immediately before the held-out run was a failure (binary)

Data leakage was avoided through an important methodological correction during feature engineering: the most recent run of each test case was removed from the feature calculations and used only as the prediction label. In a first implementation, all the historical runs, including the last one, were used both for computing features and for deriving the prediction label from the same run. This artificially inflated the APFD to 0.9643, since the model was given partial access to the outcome it was supposed to predict.

This issue was fixed by explicitly removing the latest run from all feature calculations for each test case, only using the previous 19 runs to calculate total_runs, failure_count, failure_rate, avg_duration_ms, and recent_failure, and using the held-out 20th run's outcome solely as the label. This correction brought the measured APFD to a more realistic and methodologically justifiable value of 0.7100, and is viewed as an important part of the methodological rigour of this study rather than a limitation.

2. Model Building

The Random Forest algorithm was chosen for failure-risk prediction because it is well suited for small tabular data, relatively robust to overfitting through ensemble averaging, provides interpretable feature importance scores, and handles class imbalance reasonably well when combined with class-weighting. A Random Forest classifier (n_estimators = 200, class_weight = "balanced", random_state = 42) was trained to predict the risk of failure. The value of n_estimators was obtained by comparing the performance of the model at 50, 100, 200, 300 and 500 trees (Section 5); performance was only marginally better when using 50 trees, but it plateaued from 100 trees onward.

Given the size of the dataset (n = 70, with 5 positive/failing examples, 7.1% of the suite), a single train-test split was considered inappropriate, as it could leave very few, or zero, positive examples in the held-out test set. Instead, 5-fold Stratified K-Fold Cross-Validation was used, maintaining roughly the same ratio of failing and passing test cases in each fold. Predictions from scikit-learn's cross_val_predict function were used to compute performance metrics and to generate an out-of-fold risk score for each test case, so that no test case's score was influenced by a model that had seen it during training. The final model, trained on the entire dataset, was used to analyse feature importance and the final priority ranking. The model generates a continuous risk score (predict_proba) to rank the test cases for prioritised execution, which was compared to the Normal Order (original declaration order) and a Static Priority order (alphabetical, as the evaluated suite did not specify explicit @Test(priority) values).

V. RESULTS AND EVALUATION

Table 2 shows the APFD results of each of the three execution orders on the tested suite (n = 70, m = 5 fault-revealing test cases).

Order of Execution	APFD
Normal Order	0.5643
Static Priority (Alphabetical)	0.5729
AI-Prioritized (Random Forest)	0.7100

The AI-prioritized order increased the APFD by 25.82% over the normal order, and 23.94% over the static baseline. Fig. 1 shows this comparison, and Fig. 2 shows the corresponding fault detection curves, where the AI-prioritized curve (green) has a steeper slope in the initial part of execution.

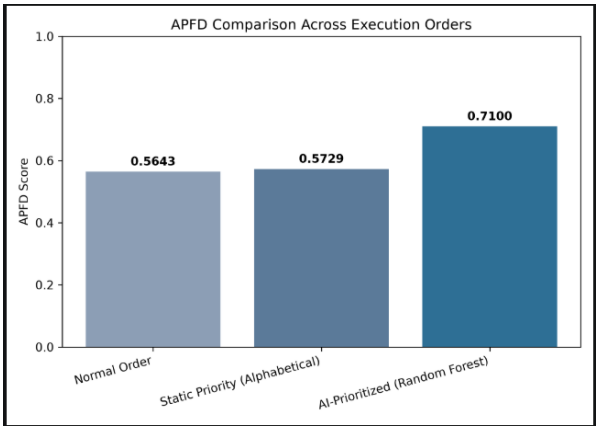

Fig. 1. APFD comparison of execution orders.

Fig. 2. Fault detection curve for Normal, Static and AI-Prioritized orders.

The Random Forest model had a cross-validated ROC-AUC score of 0.72 (Fig. 3), with a recall of 0.60 and precision of 0.33 for the failing class, correctly identifying 3 out of 5 fault-revealing test cases (Fig. 4). Table 3 summarises the cross-validated classification performance.

Metric	Class 0 (Pass)	Class 1 (Fail)
Precision	0.97	0.33
Recall	0.91	0.60
F1-score	0.94	0.43
Support	65	5

As shown in Fig. 5, avg_duration_ms and failure_count were the most influential predictive features, and Fig. 6 shows the module-wise failure distribution, where the Admin module has the maximum failure rate.

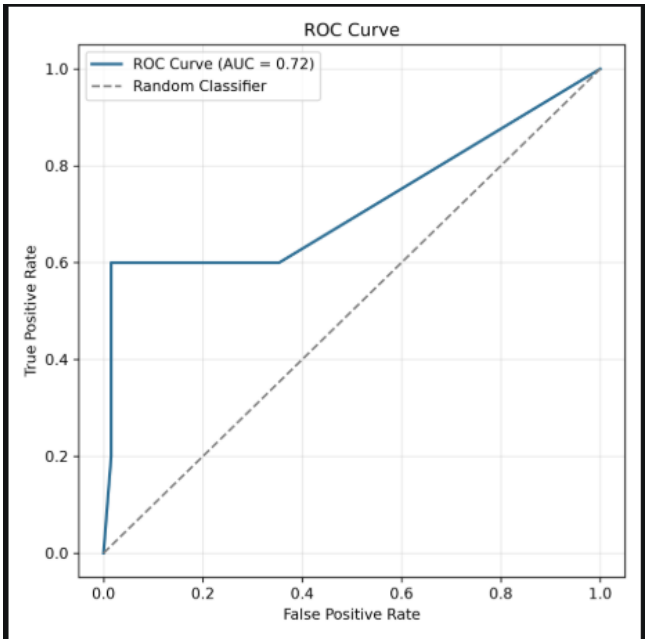

Fig. 3. ROC curve (AUC = 0.72).

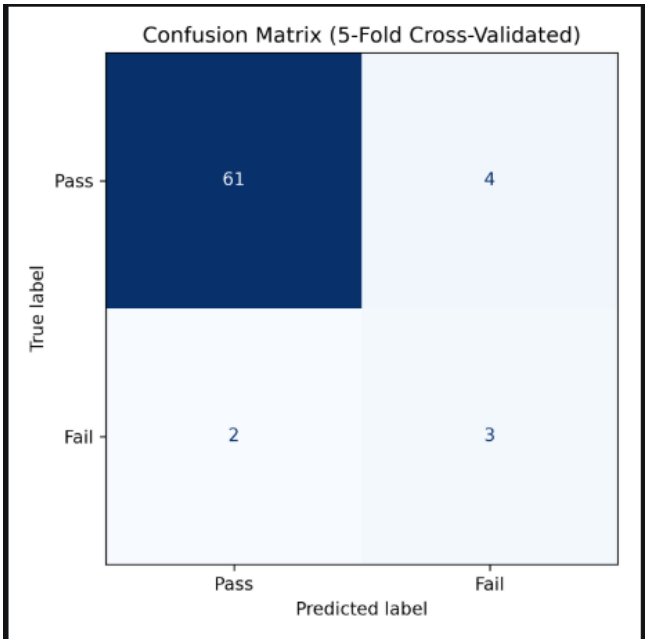

Fig. 4. Confusion matrix (5-fold cross-validated).

International Journal of Scientific Research & Engineering Trends
Volume 12, Issue 4, July-Aug-2026, ISSN (Online): 2395-566X

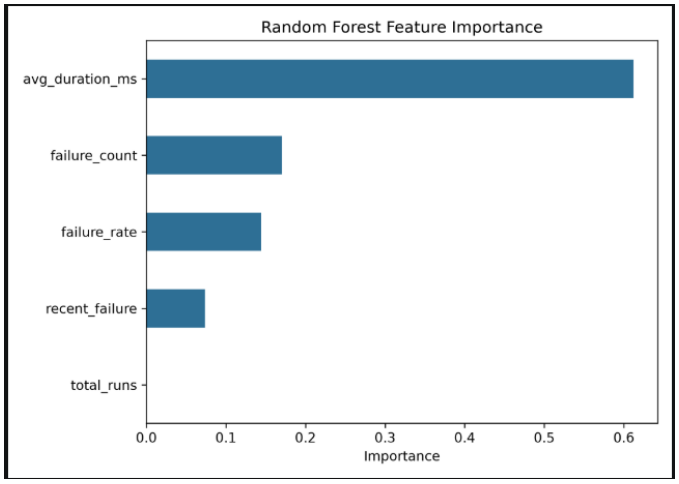

Fig. 5. Feature importance from Random Forest.

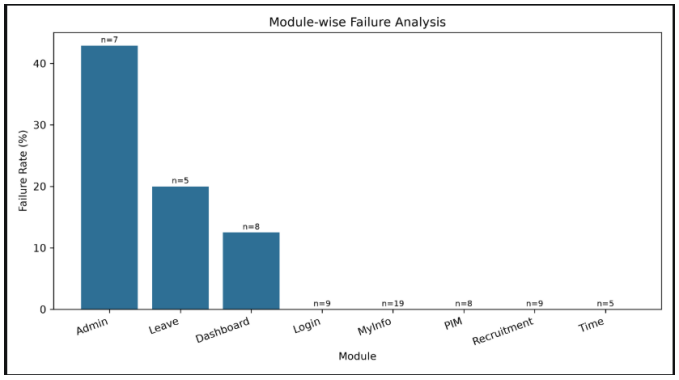

Fig. 6. Module-by-module failure analysis.

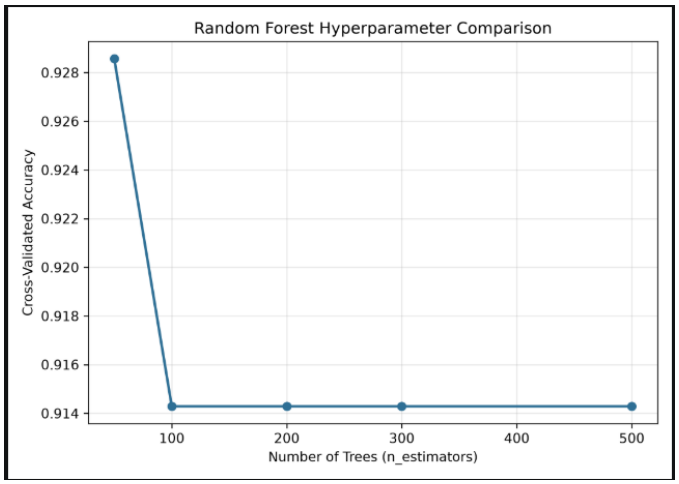

Fig. 7. Comparison of Random Forest hyperparameters (accuracy vs. number of trees).

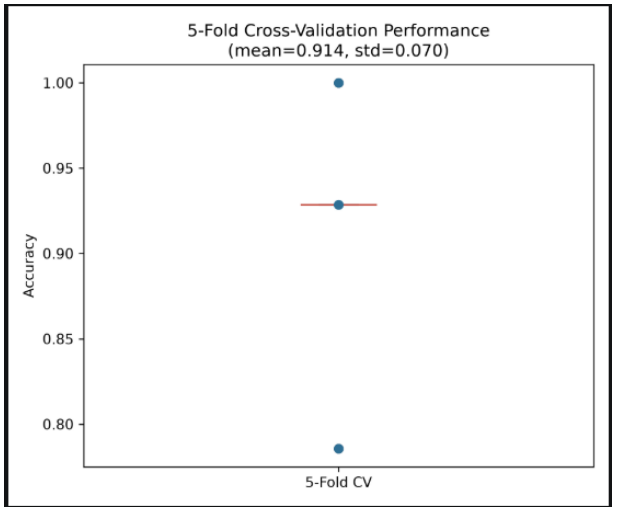

Fig. 8. 5-fold cross-validation performance (mean = 0.914, std = 0.070).

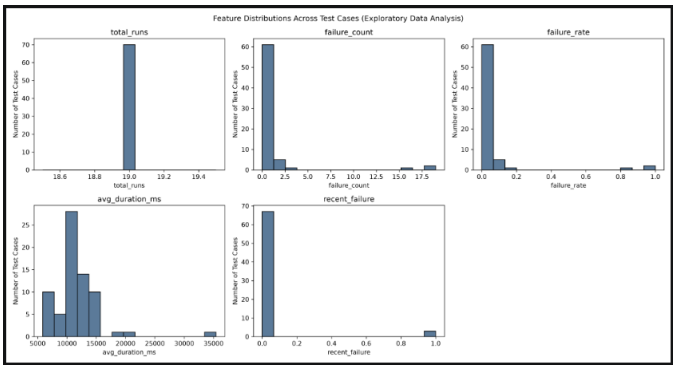

Fig. 9. Feature distributions of the test cases.

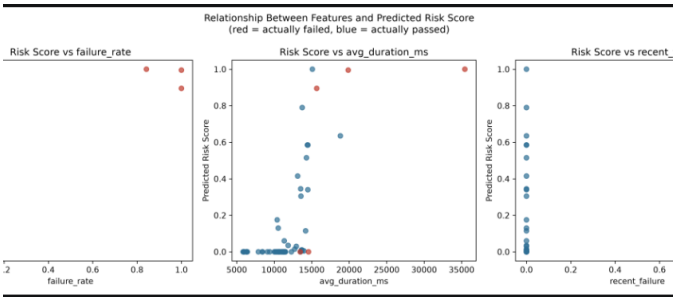

Fig. 10. Relationship between features and predicted risk score.

VI. DISCUSSION AND THREATS TO VALIDITY

The results support the hypothesis that historical, execution-based features can significantly improve early fault detection

over normal and static baselines. The explicit correction of data leakage (Section 4.1) is an important methodological contribution, providing a solid basis for the reported 25.82% improvement as a real, defensible estimate and not an artefact of information leakage. One notable finding is the importance of avg_duration_ms as the top predictive feature (Fig. 5): the high-risk test cases intentionally designed in the dataset evaluated here also happened to involve longer, more complex interactions, which indicates that duration may be serving as a proxy for test complexity, which correlates with failure-proneness.

As in any empirical study, there are two threats to validity to be mentioned. First, an internal threat: the relatively small number of fault-revealing test cases (m = 5) in the evaluation set limited the classification precision (0.33) for the failing class; this is expected to improve with more historical data and a larger, more balanced dataset, and is identified as a direct priority for future work (Section 7). Second, an external threat: the evaluation was performed on a single TestNG suite of a single application; therefore, generalisation to other projects and domains has not been empirically established yet. Extending the evaluation to multiple open-source and industrial projects is also identified as future work.

VII. CONCLUSION AND FUTURE WORK

This paper proposes an AI-based dynamic test case prioritisation system for TestNG frameworks that, based on a Random Forest classifier trained using features derived from past executions, predicts the failure risk and reorders the execution of the test cases accordingly. The system was implemented as an end-to-end pipeline from parsing the TestNG XML reports, through feature engineering, model training and APFD-based evaluation. On a 70-test-case TestNG suite over 20 historical runs, the proposed approach achieved an APFD of 0.7100, a 25.82% improvement over a normal execution order and 23.94% improvement over a static, alphabetically-ordered baseline, after an explicit and methodologically important correction for data leakage in the feature engineering process.

These results support the central hypothesis of this study, that historical, execution-based test behaviour can be exploited by machine learning to produce a meaningfully more effective test execution order than default or static prioritisation strategies, without manual configuration, source code access or coverage instrumentation. Future work includes: evaluation on larger and multiple test suites, with naturally occurring rather than synthetically induced failure patterns, to assess generalisability; comparison of Random Forest against gradient boosting methods and deep learning approaches such as DeepOrder, particularly as dataset size increases; incorporation of contextual and code-level features, such as code coverage and CI/CD contextual information, to improve model precision; and integration into live CI/CD pipelines via dynamic testng.xml regeneration or ITestNGMethod interception, closing the loop from prediction to actual prioritised execution.

REFERENCES

1. G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Test case prioritization: an empirical study," in Proc. IEEE Int. Conf. Software Maintenance, 2001.
2. H. Do, G. Rothermel, and A. Kinneer, "Empirical studies of test case prioritization in a JUnit testing environment," in Proc. ISSRE, 2004.
3. H. Mei et al., "A static approach to prioritizing JUnit test cases," IEEE Trans. Software Eng., vol. 38, no. 6, 2012.
4. Y.-C. Huang and K.-L. Peng, "A history-based cost-cognizant test case prioritization technique in regression testing," J. Systems and Software, 2011.
5. M. Kim, S. Jeong, and E. Lee, "Failure history data-based test case prioritization for effective regression test," in Proc. Int. Conf. Software Analysis, Testing and Evolution, 2017.
6. H. Spieker, A. Gotlieb, D. Marijan, and M. Mossige, "Reinforcement learning for automatic test case prioritization and selection in continuous integration," in Proc. ISSTA, 2018.
7. Y. Lou, J. Chen, L. Zhang, and D. Hao, "A survey on regression test-case prioritization," Advances in Computers, vol. 113, 2019.
8. R. Pan, M. Bagherzadeh, T. A. Ghaleb, and L. Briand, "Test case selection and prioritization using machine learning: a systematic literature review," Empirical Software Engineering, vol. 27, 2022.
9. R. A. P. Lima, S. R. Vergilio, and A. T. Endo, "Test case prioritization in continuous integration environments: a systematic mapping study," Information and Software Technology, 2020.
10. L. Ferrarini et al., "On the use of contextual information for machine learning based test case prioritization in continuous integration development," 2024.
11. S. Yoo and M. Harman, "Regression testing minimisation, selection and prioritisation: a survey," Software Testing, Verification and Reliability, vol. 22, no. 2, 2012.
12. A. Singh, K. Tarika, and R. Kumar, "Systematic literature review on regression test prioritization techniques," Informatica, vol. 36, 2012.

13. C. Catal and D. Mishra, "Test case prioritization: a systematic mapping study," Software Quality Journal, vol. 21, 2013.
14. K. Pardha Sagar and B. Prasad, "A survey on test case prioritization techniques for regression testing," Int. J. Computer Applications, 2017.
15. Y. Cho, J. Kim, and E. Lee, "History-based test case prioritization for failure information," in Proc. Int. Conf. Software Engineering Research and Practice, 2016.
16. S. Kim and M. Ali, "Automation and prioritisation technique for regression testing of PB Tech web application," 2019.