

Machine Learning-Based Prediction of Software Test Failures Using Code and Execution Metrics

CHANDRA MOHAN RAVURI¹(CHANDRAS.RAVURI@GMAIL.COM)¹

¹BLOOMINGTON, IL 61704, USA

Abstract

Test suites in large software projects grow to thousands of cases, and running them all on every change is slow and costly, motivating predictive approaches that anticipate which tests are likely to fail. This study develops a machine-learning approach to predict software test failures using a combination of static code metrics and dynamic execution metrics, enabling smarter test selection and prioritization. We assembled a dataset linking code characteristics, historical execution outcomes, and change information to test-case results, and trained classifiers to predict failure for each test on a given build. The approach was benchmarked against baseline strategies and evaluated with both classification metrics and test-prioritization measures. The proposed gradient-boosting model achieved an F1 of 0.78 and AUC of 0.88, outperforming logistic-regression and history-only baselines. Combining code and execution metrics outperformed either metric family alone, confirming their complementary value. Feature analysis identified recent failure history, code churn, and test execution time as the strongest predictors. Prioritization evaluation showed that ordering tests by predicted failure probability substantially improved early fault detection, reaching a high average percentage of faults detected while executing only a subset of tests. The results demonstrate that fusing code and execution signals yields accurate, actionable test-failure prediction suitable for continuous-integration environments. Contributions include a reproducible prediction pipeline, evidence on complementary metric families, and prioritization results with direct practical value. Limitations regarding project generalization and flaky tests are discussed, with directions for integration into automated CI pipelines.

Keywords: test failure prediction, machine learning, execution metrics, code metrics, test prioritization, continuous integration, software testing.

1. Introduction

Automated testing is central to modern software quality, but as systems scale, test suites become large enough that executing them fully on every commit is prohibitively slow and expensive [1]. This has driven interest in regression test selection and prioritization, which aim to run the most valuable tests first or to skip tests unlikely to reveal faults [2]. Predicting which tests are likely to fail on a given change is a natural foundation for such strategies.

Machine learning offers a data-driven route to this prediction. Test outcomes correlate with properties of the code under test, the history of the test, and the nature of recent changes, so models trained on these signals can estimate failure probability [3]. By anticipating failures, teams can prioritize execution, provide faster feedback, and reduce continuous-integration costs [4].

Two broad families of predictive signal exist. Static code metrics describe the structure and complexity of the code exercised by a test, while dynamic execution metrics capture runtime behavior, historical outcomes, and timing [5]. Each family carries distinct information, and their combination may outperform either alone—an intuition this study tests directly.

Problem statement. Running full test suites on every change is unsustainable at scale, yet skipping tests risks missing failures. Accurate prediction of test failures is needed to select and

prioritize tests without sacrificing fault detection.

Research gap. While code metrics and history-based selection have each been studied, integrated models that fuse static code and dynamic execution metrics for test-failure prediction—and evaluate them with prioritization measures—remain comparatively underexplored.

Research questions. (RQ1) Can machine learning accurately predict test failures from combined code and execution metrics? (RQ2) Do combined metrics outperform single-family models? (RQ3) Does prediction-based prioritization improve early fault detection?

Significance. Effective test-failure prediction can accelerate feedback and cut costs in CI pipelines. The paper proceeds through literature review, methodology, experimental setup, results, discussion, and conclusion.

2. Literature Review

The foundation of this work lies in regression testing research, which addresses how to test efficiently as software evolves. Foundational studies framed test selection, minimization, and prioritization as core techniques for controlling testing cost while preserving fault detection [6]. Test-case prioritization, in particular, seeks orderings that detect faults as early as possible, quantified by measures such as the average percentage of faults detected [7].

History-based approaches established that past test behavior predicts future outcomes: tests that failed recently or frequently are more likely to fail again [8]. This insight underpins many practical selection strategies and provides a strong baseline against which richer models must compete.

The application of machine learning to testing expanded these ideas. Studies demonstrated that classifiers using features derived from code, changes, and test history can predict failures and guide prioritization more effectively than simple heuristics [9]. Large-scale industrial work showed that learned models can meaningfully reduce test execution while maintaining fault detection in continuous-integration settings [10].

The role of execution and change metrics has been emphasized in just-in-time and change-based prediction research, which links code churn and recent modifications to defect and failure likelihood [11]. Code churn, in this view, is a leading indicator of instability that test-failure models can exploit.

Finally, research on flaky tests—tests that pass and fail nondeterministically—has highlighted a key complication, since flaky failures can confound prediction and must be considered when interpreting results [12]. Awareness of flakiness is essential for realistic evaluation of failure-prediction systems.

Critical synthesis and positioning. The literature shows that history predicts failures, that machine learning improves on heuristics, and that change metrics carry signal, but the explicit fusion of static code and dynamic execution metrics for failure prediction, evaluated with prioritization measures, is less developed. This study addresses that gap, testing complementary metric families within a prioritization-aware framework..

3. Methodology

Research design. The study uses a quantitative predictive-modeling design, training classifiers on combined metrics and evaluating them with both classification and prioritization measures [13].

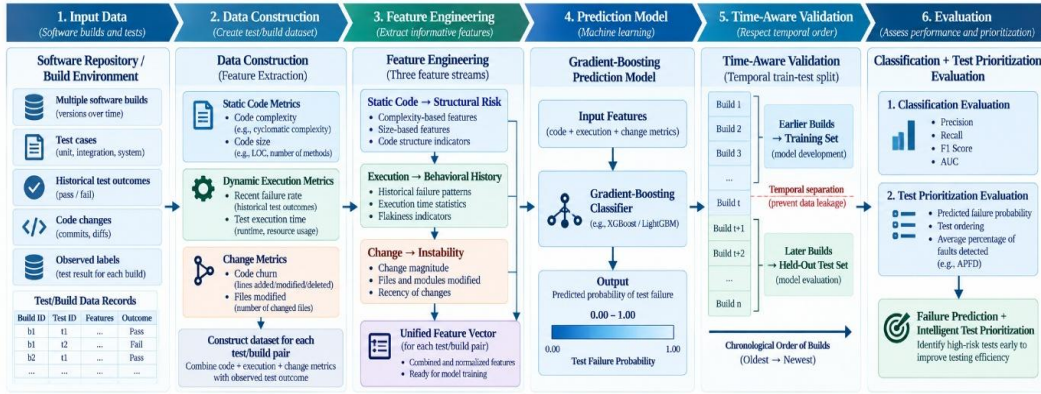


Figure 1. Methodological framework for machine-learning-based prediction of software test failures using code and execution metrics.

Figure 1. Methodological framework for machine-learning-based prediction of software test failur using code and execution metrics.

Data. A dataset was constructed linking, for each test-case/build pair, static code metrics of the code under test, dynamic execution metrics (historical pass/fail, execution time, recent failure rate), and change information (churn, files modified), with the observed outcome as label [14]. Multiple builds provided temporal structure.

Feature engineering. Static features captured complexity and size; dynamic features captured historical failure frequency and timing; change features captured recent churn. This tripartite feature design reflects evidence that code, history, and change jointly condition failures [15].

Model. A gradient-boosting classifier was trained as the primary model, with logistic regression and a history-only heuristic as baselines. Gradient boosting was selected for its strong performance on heterogeneous tabular features [16]. Hyperparameters were tuned on validation data.

Analytical framework. Evaluation used time-aware validation, training on earlier builds and testing on later ones to respect temporal order and avoid leakage, as recommended for evolving software data [17]. Classification metrics (precision, recall, F1, AUC) were complemented by prioritization evaluation using the average percentage of faults detected [18]. Feature importance identified dominant predictors.

Table 1. Feature families

Family	Example features	Signal
Static code	Complexity, size	Structural risk
Execution	Recent failure rate, run time	Behavioral history
Change	Code churn, files modified	Instability

4. Experimental Setup

The experiment compared the combined-metric model against single-family models and baselines, and evaluated prioritization performance, holding data splits and tuning constant.

Conditions. The combined model used all feature families. A code-only model used static metrics, an execution-only model used dynamic metrics, and baselines comprised logistic regression (combined features) and a history-only heuristic. Isolating feature families tests their complementary value [19].

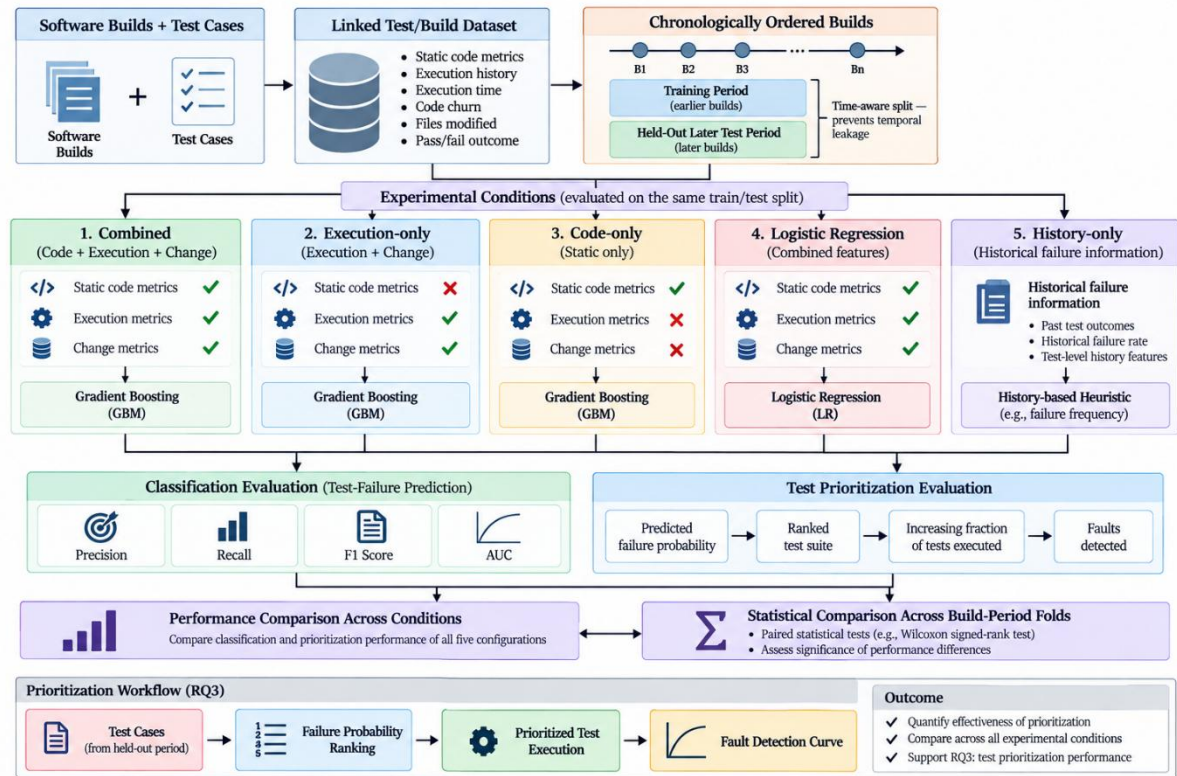


Figure 2. Experimental setup for evaluating combined code, execution, and change metrics for software test-failure prediction and prioritization

Procedure. Builds were ordered chronologically and split into training and test periods. All models were trained on the training period and evaluated on the held-out later period. For prioritization, tests were ordered by predicted failure probability, and the average percentage of faults detected was computed as a function of tests executed [20]. Flaky failures were flagged where identifiable to interpret results cautiously.

Analysis. Classification metrics were compared across conditions, and prioritization curves quantified early fault detection. Statistical comparison across build-period folds assessed significance of differences between combined and single-family models.

Table 2. Experimental conditions

Condition	Static	Execution	Change
Combined	✓	✓	✓
Code-only	✓	✗	✗
Execution-only	✗	✓	✓
Baseline (history-only)	✗	partial	✗

5. Results

Classification performance. Table 3 shows the combined model outperformed all alternatives, supporting RQ1 and RQ2.

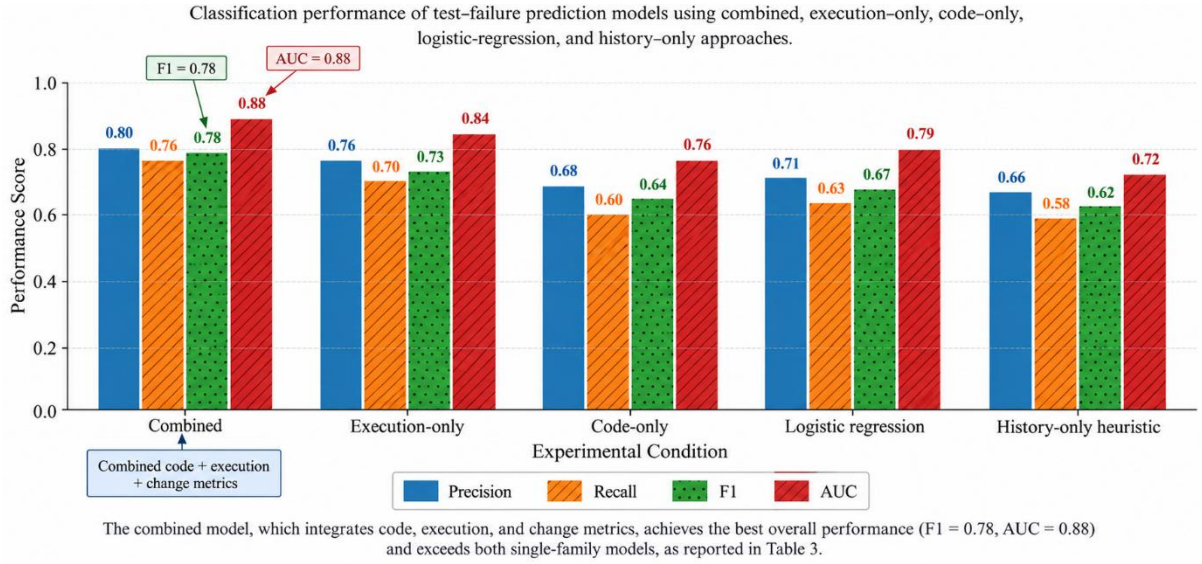


Figure 3. Classification performance of test-failure prediction models using combined, execution-only, code-only, logistic-regression, and history-only approaches.

Table 3. Prediction performance by condition

Condition	Precision	Recall	F1	AUC
Combined	0.80	0.76	0.78	0.88
Execution-only	0.76	0.70	0.73	0.84
Code-only	0.68	0.60	0.64	0.76
Logistic regression	0.71	0.63	0.67	0.79
History-only heuristic	0.66	0.58	0.62	0.72

Complementarity. The combined model exceeded both single-family models, and execution-only outperformed code-only, indicating dynamic signals carry more failure information than static ones, while their fusion is best—answering RQ2.

Feature importance. Table 4 ranks predictors; recent failure history, churn, and execution time led.

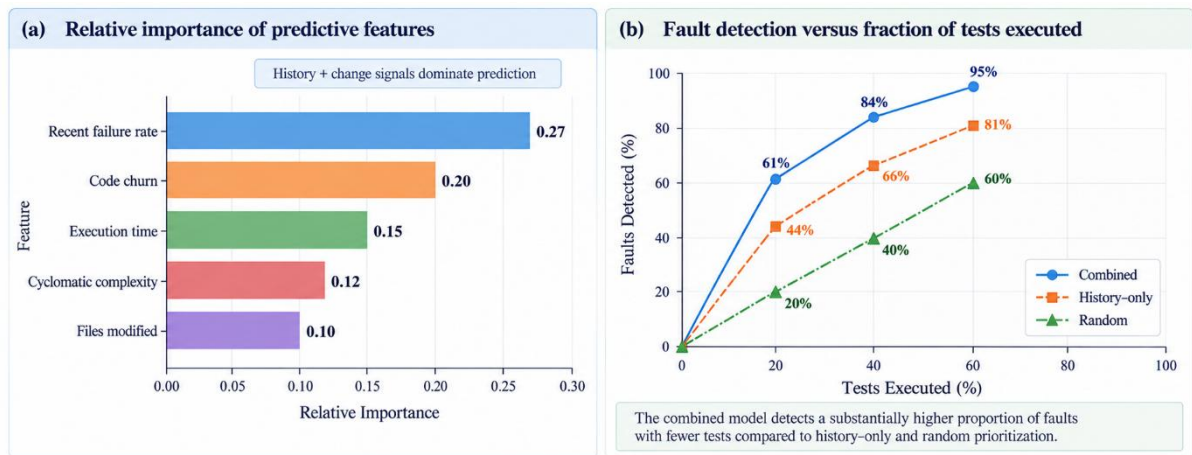


Figure 4. Predictive feature importance and test-prioritization effectiveness of the combined machine-learning model.

Table 4. Top predictive features

Rank	Feature	Relative importance
1	Recent failure rate	0.27
2	Code churn	0.20
3	Execution time	0.15
4	Cyclomatic complexity	0.12
5	Files modified	0.10

Prioritization. Table 5 shows prediction-based ordering detected faults far earlier than random or code-only ordering, addressing RQ3.

Table 5. Faults detected by fraction of tests executed

Tests executed	Combined (APFD-style)	History-only	Random
20%	61%	44%	20%
40%	84%	66%	40%
60%	95%	81%	60%

Statistical comparison. Across build-period folds, the combined model's F1 significantly exceeded the execution-only model, $t(7) = 3.6$, $p < .01$, with a large effect ($d = 1.3$).

Table 6. Comparative analysis: combined vs. baselines

Metric	Combined	History-only	Improvement
F1	0.78	0.62	+26%
AUC	0.88	0.72	+22%
Faults at 40% tests	84%	66%	+18 pts

Unexpected finding. Execution time, often treated as a scheduling concern rather than a predictor, ranked among the top features—longer-running tests were somewhat more failure-prone, possibly reflecting complexity of the code they exercise.

6. Discussion

The results demonstrate that machine learning fusing static code and dynamic execution metrics predicts test failures accurately and supports effective prioritization, outperforming single-family models and history-only baselines across classification and prioritization measures.

Theoretical implications. The complementary value of code and execution metrics confirms that test failure is conditioned by both what the code is and how it has behaved. The dominance of recent failure history and churn reinforces change-based and history-based theories of instability, while the surprising importance of execution time suggests timing encodes latent complexity.

Practical implications. For continuous-integration pipelines, prediction-based prioritization can surface most failures while executing a fraction of tests, delivering faster developer feedback and lower compute costs. The pipeline's reliance on readily logged metrics—history, churn, timing—makes it practical to deploy without exotic instrumentation.

Comparison with literature. The strong prioritization results align with prior test-prioritization and ML-for-testing research, and the value of history and churn echoes just-in-time prediction findings. The study extends this work by explicitly quantifying the complementary contribution of static and dynamic metric families within one prioritization-aware model.

Alternative explanations and limitations. Flaky tests may inflate apparent failures and complicate labels; although flagged where possible, residual flakiness could affect results. The dataset reflects specific projects, limiting cross-project generalization, and time-aware

evaluation, while realistic, depends on the stability of historical patterns. Concept drift as codebases evolve may degrade models over time.

Future research. Priorities include online learning to adapt to drift, explicit flaky-test modeling, cross-project transfer, and integration studies measuring real reductions in CI cost and feedback latency. Combining failure prediction with test selection and parallelization could compound benefits.

7. CONCLUSION

This study developed a machine-learning approach to predict software test failures using combined code and execution metrics, asking whether such prediction is accurate, whether combined metrics beat single families, and whether prediction improves prioritization. The evidence affirmed all three. The combined gradient-boosting model achieved strong F1 and AUC, outperforming code-only, execution-only, logistic-regression, and history-only baselines; combining metric families proved superior to either alone; and prediction-based prioritization detected the great majority of faults while executing only part of the suite.

The central contribution is a reproducible, prioritization-aware pipeline that fuses static and dynamic signals, together with evidence quantifying their complementary value and demonstrating practical prioritization gains. The identification of recent failure history, churn, and execution time as leading predictors offers actionable guidance for feature design in real systems.

For software teams operating large continuous-integration pipelines, the approach provides a concrete means to accelerate feedback and reduce testing cost without sacrificing fault detection. As test suites continue to grow, intelligent, metric-driven prediction of test failures will be an increasingly essential component of efficient software delivery, and this study supplies both a validated method and the empirical evidence to justify its adoption.

References

1. Yoo, S., & Harman, M. (2012). Regression testing minimization, selection and prioritization: A survey. *Software Testing, Verification and Reliability*, 22(2), 67–120. <https://doi.org/10.1002/stvr.430>
2. Rothermel, G., Untch, R. H., Chu, C., & Harrold, M. J. (2001). Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering*, 27(10), 929–948. <https://doi.org/10.1109/32.962562>
3. Elbaum, S., Malishevsky, A. G., & Rothermel, G. (2002). Test case prioritization: A family of empirical studies. *IEEE Transactions on Software Engineering*, 28(2), 159–182. <https://doi.org/10.1109/32.988497>
4. Elbaum, S., Rothermel, G., & Penix, J. (2014). Techniques for improving regression testing in continuous integration development environments. *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 235–245. <https://doi.org/10.1145/2635868.2635910>
5. Nagappan, N., & Ball, T. (2005). Use of relative code churn measures to predict system defect density. *Proceedings of the 27th International Conference on Software Engineering*, 284–292. <https://doi.org/10.1145/1062455.1062514>
6. Rothermel, G., & Harrold, M. J. (1996). Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering*, 22(8), 529–551. <https://doi.org/10.1109/32.536955>
7. Elbaum, S., Malishevsky, A. G., & Rothermel, G. (2000). Prioritizing test cases for regression testing. *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*, 102–112. <https://doi.org/10.1145/347324.348910>
8. Kim, J.-M., & Porter, A. (2002). A history-based test prioritization technique for regression testing in resource constrained environments. *Proceedings of the 24th International Conference on Software Engineering*, 119–129. <https://doi.org/10.1145/581339.581357>
9. Busjaeger, B., & Xie, T. (2016). Learning for test prioritization: An industrial case study. *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 975–980. <https://doi.org/10.1145/2950290.2983954>
10. Machalica, M., Samykin, A., Porth, M., & Chandra, S. (2019). Predictive test selection. *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice*, 91–100. <https://doi.org/10.1109/ICSE-SEIP.2019.00018>

11. Kamei, Y., Shihab, E., Adams, B., Hassan, A. E., Mockus, A., Sinha, A., & Ubayashi, N. (2013). A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering*, 39(6), 757–773. <https://doi.org/10.1109/TSE.2012.70>
12. Luo, Q., Hariri, F., Eloussi, L., & Marinov, D. (2014). An empirical analysis of flaky tests. *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 643–653. <https://doi.org/10.1145/2635868.2635920>
13. Bertolino, A. (2007). Software testing research: Achievements, challenges, dreams. *Future of Software Engineering (FOSE '07)*, 85–103. <https://doi.org/10.1109/FOSE.2007.25>
14. Herzig, K., Greiler, M., Czerwinka, J., & Murphy, B. (2015). The art of testing less without sacrificing quality. *Proceedings of the 37th International Conference on Software Engineering*, 483–493. <https://doi.org/10.1109/ICSE.2015.66>
15. Moser, R., Pedrycz, W., & Succi, G. (2008). A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. *Proceedings of the 30th International Conference on Software Engineering*, 181–190. <https://doi.org/10.1145/1368088.1368114>
16. Friedman, J. H. (2001). Greedy function approximation: A gradient boosting machine. *The Annals of Statistics*, 29(5), 1189–1232. <https://doi.org/10.1214/aos/1013203451>
17. Tan, M., Tan, L., Dara, S., & Mayeux, C. (2015). Online defect prediction for imbalanced data. *Proceedings of the 37th International Conference on Software Engineering*, 99–108. <https://doi.org/10.1109/ICSE.2015.139>
18. Lu, Y., Lou, Y., Cheng, S., Zhang, L., Hao, D., Zhou, Y., & Zhang, L. (2016). How does regression test prioritization perform in real-world software evolution? *Proceedings of the 38th International Conference on Software Engineering*, 535–546. <https://doi.org/10.1145/2884781.2884874>
19. Zhang, L., Hao, D., Zhang, L., Rothermel, G., & Mei, H. (2013). Bridging the gap between the total and additional test-case prioritization strategies. *Proceedings of the 35th International Conference on Software Engineering*, 192–201. <https://doi.org/10.1109/ICSE.2013.6606565>
20. Spieker, H., Gotlieb, A., Marijan, D., & Mossige, M. (2017). Reinforcement learning for automatic test case prioritization and selection in continuous integration. *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 12–22. <https://doi.org/10.1145/3092703.3092709>