www.ijecs.in
International Journal Of Engineering And Computer Science
Volume 14 Issue 07 July 2025, Page No. 27547-27554
ISSN: 2319-7242 DOI: 10.18535/ijecs/v14i07.5186

Quality Metrics of Automated Tests in the Application of Machine Learning

Vladyslav Korol

Software Developer Engineer In Test (Penn Entertainment)
USA, Florida, Miami

Abstract
The article examines how machine learning methods are being increasingly integrated into automated testing processes; however, the absence of a unified system of quality metrics complicates the assessment of their effectiveness. In light of rising costs for defect resolution and the growing proportion of unstable (flaky) tests, investigating quality metrics for automated tests, taking machine learning (ML) approaches into account, becomes highly relevant. The goals of this work are to classify existing metrics and apply them to the particular requirements of testing ML systems; further, an economic model is proposed for decision justification in CI processes. Research novelty lies in unifying classical, machine learning (ML)-oriented, and financial metrics, and evaluating their impacts on practical costs for infrastructure, as demonstrated through industrial cases from Facebook/Meta, Netflix, and Slack. It was found that machine learning (ML) approaches to automated testing via Predictive Test Selection led to significant savings in CPU hours (Gradle, Netflix) and reductions in the percentage of flaky tests—data from Facebook/Meta and Slack also confirms this. It is shown here that PR-AUC and other precision–recall–based metrics are more faithful to real-world imbalanced defect classes than ROC curves. The economic model, with coefficients C_FP and C_FN, enables the computation of the optimal tradeoff between test execution speed and defect detection rate, allowing decisions to be made during Continuous Integration/Continuous Deployment. Quantitative data give rise to integral metrics that combine coverage type, build stability, and economic components. The article will be of particular interest to quality assurance engineers, project managers, and researchers in the fields of software testing and machine learning.

Keywords: automated tests, machine learning, quality metrics, Predictive Test Selection, code coverage, PR-AUC, C_FP, C_FN, CI/CD, build stability.

Introduction

Modern software systems are increasingly rarely released without a dense layer of automated tests, yet scenarios of the type «it ran green» are no longer sufficient. The CISQ report highlights the economic scale of the problem: direct losses from low-quality software in the USA reached 2.41 trillion USD in 2022, and a significant portion of this amount is associated with defects detected late [1]. The classic study [2] demonstrates that fixing a bug after release is more expensive than during the design phase, and as technical debt accumulates, the cost increases. Therefore, every engineering dollar invested in tests must be justified by a measurable effect: how many defects were caught earlier, how many CI downtimes were eliminated, and how quickly the team receives feedback.

One must focus not only on the cost of defects but also on defects in the testing process itself. The use of machine learning for prioritizing a regression suite, predicting failures, or detecting flaky tests makes the measurement task even more multilayered. Now, evaluations are needed at two levels: the model and the process. However, such gains are possible only with strict control of metrics, such as Recall for identifying critical failures and Precision for resource allocation. A data drift factor emerges: code and infrastructure

changes cause validation metrics to «float», so classical coverage and pass-rate are augmented by ROC-AUC, probability calibration, and the cost of erroneous predictions.

Materials and Methodology

Another 16 sources—in reports from the industry, academic research, practical case studies, and technical reviews—offer quality metrics for automated tests in the context of machine learning. Data for this study were drawn from three sources. First is the CISQ report on the cost of low-quality software in the USA [1]. Next is a classic survey reported in Forbes on the price of fixing bugs [2]. The other one is from the CircleCI 2023 State of Software Report, which considers economic and contextual factors along with the impacts that defects have on cost development [10]. The conceptual underpinning was built from a review of test methodologies for ML systems[3] and studies on error metrics in ML, such as the Brier Score and Calibration Error[4], which help show the basic ideas behind model validation and the control needs for probabilistic predictions. Practical case studies include Predictive Test Selection at Facebook/Meta [5, 15], as well as a survey of the cost of flaky tests in an industrial example [6] and an analysis of system-wide flakiness load [7], providing quantitative assessment of the impact of ML approaches on the CI process. Data on code coverage and tool usage were taken from the study [8] and from the practical experience of Slack engineers in suppressing unstable tests [9]. For methods of assessing the impact of business costs of errors, publications on the limitations of ROC curves under imbalanced data conditions [11, 12, 14] and materials from Gradle on real figures for CPU-hour savings with Predictive Test Selection [13] and Netflix examples [16] were utilized. Expert opinions on the need to balance speed and quality metrics, presented in works [7, 12], were also considered.

The research methodology includes three interrelated stages. The first stage is a systematic review of the literature and reports. Data on class imbalance and limitations of the ROC curve were obtained from studies [11, 12, 14], which justified the transition to Precision–Recall pairs and PR-AUC for evaluating machine learning models. The second stage involves content analysis of practical case studies, based on works [5, 15] and the Netflix case from Gradle [16], to assess fundamental changes in infrastructure costs and defect coverage levels after the implementation of Predictive Test Selection. The analysis of the cost of flaky tests [6] and empirical statistics from Parry et al. [7] were used to calculate the coefficients C_FP and C_FN, which reflect the costs of false positives and missed defects. The third stage involves the quantitative analysis of reports and process metrics. Data on code coverage was obtained from [8], while build stability indicators and the distribution of failure causes were derived from Slack Engineering [9] and CircleCI [10].

Results and Discussion

A two-level approach to quality assessment becomes necessary as soon as machine learning models are introduced into the automated testing pipeline. At the first level, the model itself is evaluated: its statistical correctness determines which tests will be executed, skipped, or prioritized. At the second level, the effect of these decisions on the entire CI/CD process is measured—that is, how much faster and cheaper the team receives reliable feedback. A review of 144 studies on testing machine learning (ML) systems emphasizes that a separate but coordinated system of metrics is the primary condition for managing such solutions [3].

For the model, classical classification and regression metrics are used. Accuracy is convenient for rough diagnostics, but in tasks of selecting essential tests, the emphasis shifts to Recall, since a missed defect is more expensive than an extra run. Precision is vital for saving computational minutes, and the F1-score helps maintain balance in the presence of class imbalance. Validation of confidence probabilities via Brier Score or Expected Calibration Error shows how closely the model's values correspond to the real likelihood of a test failure [4].

At the process level, production indicators become key: CPU-hour savings, acceleration of feedback, and the proportion of defects caught before merge. In an industrial experiment at Facebook (Predictive Test Selection), the transition to ML prioritization halved infrastructure costs while preserving 95% of individual failures and 99.9% of defective changes reported to developers [5]. Similar evaluations are complemented by Flakiness Index and Fault Detection Effectiveness, allowing one to understand which areas of the regression suite contribute the most to quality for the minimum time budget.

The cost of an incorrect model decision can be conveniently expressed as $C = FP \times C\_FP + FN \times C\_FN$. Field analytics show a gap of several orders of magnitude: an automatic rerun of an unstable test

costs about 0.02¢ [6], whereas manual investigation of a false failure costs $5.67; in total, teams spend up to 1.28% of working time on fixing flaky tests, which is equivalent to approximately $2250 per month for an average product team [7]. Thus, in setting up the model, we assign the coefficients C_FP and C_FN according to business priorities. For instance, in critical subsystems, we may increase the weight of a false negative so that the model detects potential failures more aggressively.

Bringing together these two levels of metrics provides the team with a holistic picture: how well the model performs in terms of statistical validity and what benefits accrue from the process in production. Only in this way can one ensure that intelligent prioritization truly enhances product quality, rather than simply adding another black box to the delivery chain.

Classical indicators provide a starting point, without which it is impossible to reliably judge whether ML prioritization of the test suite improves delivery quality. The oldest indicator remains code coverage. Line coverage records the fact of executing a line, while branch coverage checks all outcomes of conditional operators, so branch coverage is fundamentally more accurate when searching for dead branches. In a large sample of 100 open-source projects, line coverage averaged 53%, with quartiles at 28% and 77% - illustrating how far real code can stray from formal standards [8]. Thus, it is essential to record both types of coverage when comparing ML suggestions with the baseline suite and observe how the branch component evolves.

Next comes Pass Rate — the proportion of green runs in continuous integration. The metric is trivial by formula, but critical for the team's morale: as long as the branch is red, developers cannot merge code and are forced to identify the cause. Slack engineers reported that, before implementing automatic filtering of flake tests, the average stability of the main branch was only 20% successful builds; after suppressing noisy tests, the indicator sharply increased [9]. A detailed analysis of 80% of failed builds revealed that 57% of them failed due to test-task failures, including unstable and failing automated tests. Additionally, 13% of builds were unsuccessful due to developer errors, CI problems, or testing infrastructure issues, and another 10% were due to merge conflicts, as shown in Fig. 1.

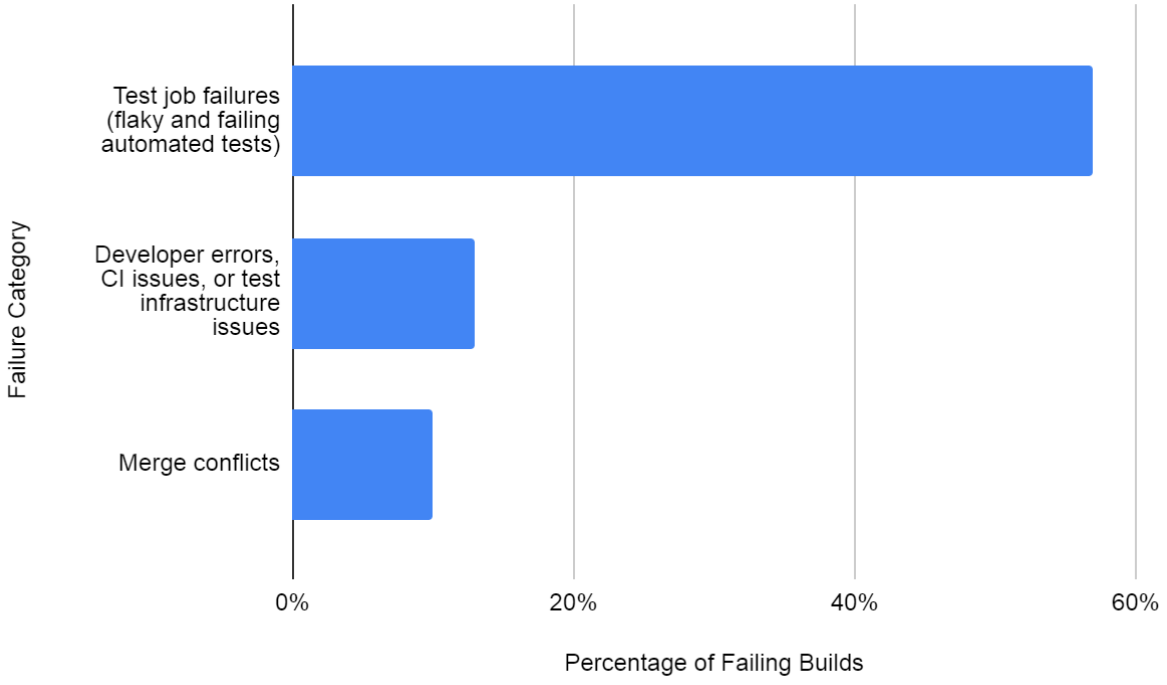

Fig. 1. Distribution of reasons for failed CI builds [9]

Average runtime directly converts into infrastructure costs and cognitive overhead. Study [10] shows that only half of projects recover from a failed workflow in 64 minutes or less, indicating that most teams have significant room to reduce their feedback cycle time [10]. Fifty percent of workflows complete in 3.3 minutes or less (the fastest 25% complete in less than 1 minute), and 75% finish within 9 minutes, while the average duration is about 11 minutes and in the 95th percentile exceeds 27 minutes, as shown in Fig. 2.

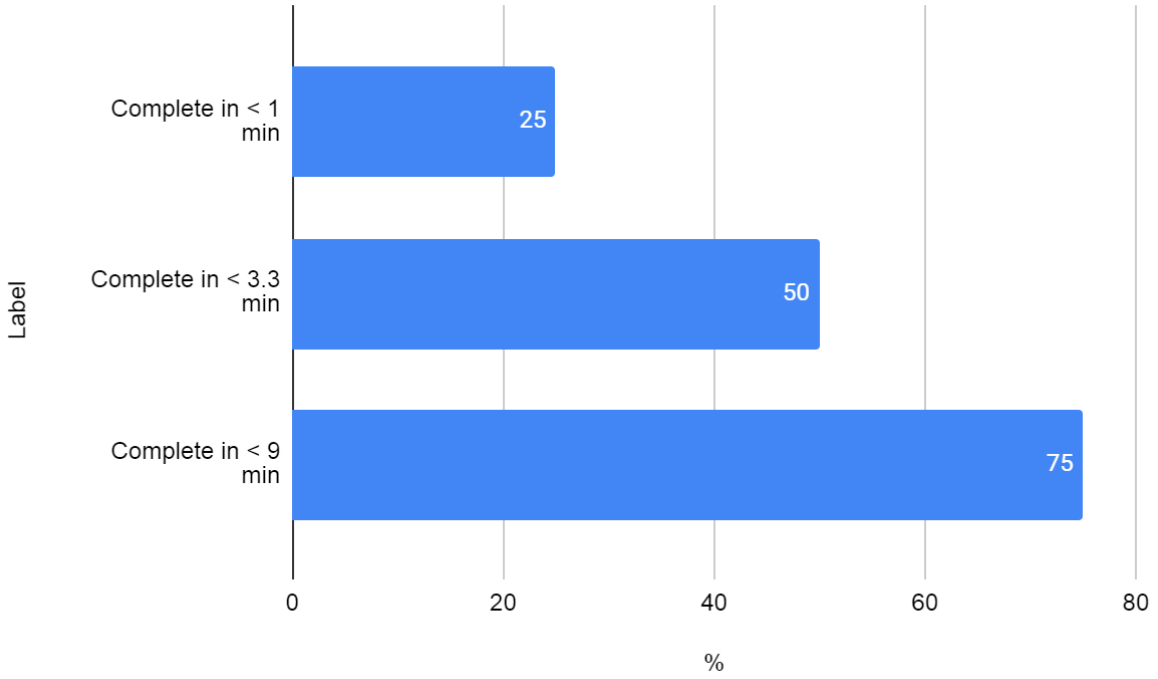

Fig. 2. Percentage of Workflows Completed Within Time Thresholds [10]

Finally, Defect Detection Rate links testing metrics to the discovery of real bugs. The formula (defects found by automation)/(all defects) yields the percentage of defects intercepted before manual testing or production. This coefficient is convenient because it remains interpretable even after integrating ML ranking: one can directly measure whether intelligent ordering improved the proportion of early findings with the same or fewer runs.

Collectively, four fundamental indicators—coverage, Pass Rate, average cycle time, and Defect Detection Rate—form the control group for all subsequent ML experiments. Suppose a new model demonstrates a statistically significant increase in at least one of these without degradation of the others. In that case, one can talk of true progress rather than the illusion of betterment created by complex analytics. The move from classical regression suites to intelligent ranking has made the measurement system way more complicated and multi-layered. Before running a thousand tests, we first assess the model to determine which tests are necessary and then proceed with the confidence interval (CI) process itself. This shift has necessitated introducing a new upper group of indicators while leaving the basic metrics as a control group on which one can rely during any A/B experiments. At the classification level, the best-known limitation is the unreliability of Accuracy on imbalanced samples. Engineers examine the Precision–Recall pair, which indicates how many extra tests were conducted and how many critical failures were missed. When classes are very imbalanced, computing the area under the ROC curve starts to give an advantage to the method, whereas PR-AUC stays sensitive to rare failures. A study by Klarna engineers showed that at a ratio of 1:1000, ROC-AUC overestimates real quality by 0,07 on average [11], whereas PR-AUC deviates by no more than 0,01; a similar conclusion about ROC metrics' over-optimism is confirmed by a medical meta-analysis of imbalanced samples [12]. For regression tasks—for example, predicting execution time or defect probability as a continuous variable—MAE, RMSE, and MAPE are used. However, high values of Precision or low RMSE do not guarantee that model probabilities reflect reality. To check calibration, the Brier Score and Expected Calibration Error are used.

The choice of the primary metric depends on the business cost of errors. Therefore, in the task of detecting flaky tests, Recall becomes paramount, while Precision is constrained by the minimum acceptable threshold that prevents CI budget inflation. Moving to the automated test suite level, the first indicator of the ML era became the Flakiness Index—the percentage of non-deterministic runs for each test.

The Mutation Testing Score complements the picture by measuring how effectively the suite detects artificially injected defects. To understand how quickly tests catch real bugs, Fault Detection Effectiveness is used. Coverage Differential captures the gain in branch coverage explicitly achieved through the model's

recommendations. The TPS approach, based on the latest code changes, increased hot coverage without increasing total runtime.

Finally, Feature-Importance Coverage compares the code areas the model deems most risky with the actual coverage of those areas by tests. All the listed indicators together form a closed loop: the model makes a prediction, the test suite records the result, and metrics at both levels provide transparent feedback, allowing for the timely retraining of the algorithm or redesign of the scenarios themselves.

The fundamental linking element is the Cost of Wrong Prediction. The formula C = FP × C_FP + FN × C_FN converts classification errors into currency: an extra run (false positive) is counted according to the CI meter, and a missed defect (false negative) is evaluated by average investigation time and potential downtime cost. The next layer is a direct measurement of saved computational resources. The typical range of benefit, confirmed by Gradle customers, is 35–70% of each build's duration without degrading result reliability [13].

The higher the MTES, the more tempting it is to trim the test suite further, but this is precisely where balance is required. Practice shows that beyond a threshold of about 70% savings, each additional minute gained from CI is purchased at an exponential increase in the risk of missing a critical defect; this is visible in the CWP curve, which steepens due to the rapidly growing FN × C_FN component. Therefore, the strategic goal is not merely a minimal build time, but rather the point at which the MTES gain no longer compensates for the rise in overall risk. The five sprints methodology recommends empirically fixing this compromise: run A/B experiments with different fractions of selected tests, measure MTES and CWP in parallel, and keep the configuration in which the MTES increment per unit increase in error cost becomes negligibly small.

By introducing machine learning into the test pipeline, the team inevitably converts technical metrics into economic ones. This is why, before any optimizations, one should design composite indicators such as CWP and MTES and agree on how they will translate deviations in Pass Rate or Recall into monetary and reputational risks. Prioritizing metric implementation is easiest in an iterative manner: first the control group of fundamental indicators is recorded (coverage, Pass Rate, average run time, Defect Detection Rate), then CI logs are extended with model predictions fields, allowing construction of the confusion matrix and daily calculation of Recall/Precision, and only after these numbers stabilize are CWP and MTES computed. This sequence prevents a situation in which the team optimizes a complex index without noticing that actual code coverage has dropped.

The most common pitfall is relying on Accuracy when positive examples are imbalanced. Theoretical and practical reviews emphasize that when failures are rare, this metric systematically overstates model quality and masks missed defects. For a reliable picture, at least precision–recall or PR-AUC are needed [14]. Another mistake is forgetting about data drift: if a model was trained on six-month-old statistics, quick tests may become obsolete and cease to cover risky code areas, while the report appears green. Ignoring unstable tests affects all layers simultaneously. Therefore, quarantining flaky tests and moving their metrics to a separate dashboard is the first mandatory step before building any machine learning (ML) models.

Significant cases confirm the economic potential of properly configured metrics. At Meta, the transition to Predictive Test Selection doubled testing infrastructure efficiency by reducing the executable suite to one-third while retaining 99.9% of caught regressions [15]. Develocity at Netflix demonstrates that this approach can save over 280,000 CI-hours per year and, in individual projects, reduce test phase duration by up to 90% without compromising quality [16].

All the mentioned examples converge on one conclusion: the most substantial effect is achieved when build time remains below six to seven minutes—precisely at this threshold, developers do not have time to switch to another task, and the team reacts instantly to a red build, preserving cognitive context. Therefore, testing metric optimization must stay within the triangle of quality, speed, and cost: as soon as the MTES gain from further suite trimming becomes smaller than the CWP increase, the system should be retrained rather than further extending the cycle. The pros and cons of ML-based test prioritization metrics are summarized in Table 1.

Table 1. Pros and Cons of ML-Based Test Prioritization Metrics
(compiled by the author)

Pros	Cons

Early defect detection (increased Recall) by prioritizing critical tests	High metric complexity: need to track not only coverage but also ROC-AUC, PR-AUC, and probability calibration
Significant CI resource savings (reduction in CPU-hours and build times)	Risk of missing a critical defect (false negative) when optimizing aggressively—high cost of FN × C_FN
Faster feedback loop for developers (reduced CI/CD cycle time)	Need for continuous monitoring of data drift and retraining the model after code or infrastructure changes
Reduction of flaky-test noise: improved branch stability and higher Pass Rate	Additional overhead for collecting and storing model prediction statistics (logging, dashboards)
Flexibility to adjust business coefficients (C_FP, C_FN) based on project priorities	Potential black box effect: Developers may struggle to understand how tests are selected and ranked
Ability to measure economic impact (composite metrics CWP, MTES) in monetary and time equivalents	Labor-intensive A/B experiments are required to find the optimal balance between MTES and CWP
Improved branch coverage through Feature-Importance Coverage	Implicit costs for training and maintaining ML engineers
Easy integration with existing CI/CD pipelines (adding prediction fields to logs)	Increased overall complexity of the test pipeline (new dependencies, additional validation steps)

Thus, the introduction of machine learning into automated testing requires a transition from static indicators to a multi-level system of metrics that combines both classical indicators (coverage, Pass Rate, average cycle time, and Defect Detection Rate) and indicators of model effectiveness (Recall, Precision, ROC-AUC/PR-AUC, probability calibration) and economic coefficients (C_FP, C_FN, CWP, MTES). Such a complex makes it possible, at the same time, to ensure high reliability of predictions, minimize resource costs on CIs, and detect data drift promptly, which is critical for maintaining feedback stability and speed. Every new ML model must show statistically significant improvement in at least one basic or process metric without deterioration in others. A/B experiments with different test proportions must further determine an optimum trade-off between defect-finding time and risk of missing defects. This practically means business coefficients should be calibrated regularly, Flakiness Index and Feature-Importance Coverage need to be monitored continuously, and models must be retrained flexibly as soon as there are changes to code or infrastructure. It is this balanced, iterative approach to quality metrics development and monitoring that ultimately makes it possible to attain the quality-speed-cost triad while raising the test strategy to long-term efficiency.

Conclusion

Quality metrics of automated tests in the era of machine learning require a comprehensive, multi-level approach that unites classical and model indicators with economic coefficients. First, it is necessary to maintain the basic control group of metrics—code coverage, Pass Rate, average cycle time, and Defect Detection Rate—to establish an objective baseline for any modernization of the test pipeline.

The next important component is measuring the economic efficiency of machine learning (ML) ranking decisions. Using the formula for the cost of wrong prediction, $C = FP \times C_{FP} + FN \times C_{FN}$, allows converting technical errors into monetary and time losses. Practical cases demonstrate that with properly chosen coefficients, the cost of false positives is hundreds of times lower than the cost of a missed defect: an automatic rerun of a flaky test costs on the order of a hundredth of a cent, whereas manual investigation of a false failure costs several dollars. Thus, prioritizing business priorities (raising the FN weight for vital

subsystems) ensures a more aggressive selection of potential problematic tests and risk minimization. Simultaneously, savings in CPU hours and a reduction in build duration (MTES) serve as a direct indicator of benefit, allowing teams to redirect released resources toward more valuable tasks.

Organizationally and methodically, the key lies in the iterative implementation of metrics: first, classical indicators are logged; then, model-predicted fields are added to CI logs, on which a confusion matrix is built and Recall/Precision is aggregated daily. Only after these numbers stabilize are combined indicators, such as CWP (Cost of Wrong Prediction) and MTES, introduced, linking model accuracy with resource and business risks. This approach prevents a situation where the team optimizes a complex composite index without noticing that actual code coverage or Pass Rate has declined. One of the most essential practical conclusions is the necessity of regular A/B testing with different proportions of selected tests in order to empirically identify the balance point at which additional CI resource savings are no longer compensated by the exponentially increasing risk of missing a critical defect.

Data drift must be monitored carefully, as changes occur to the codebase and infrastructure. They influence fluctuations in the validation metrics; therefore, the model is revaluated periodically and retuned as needed. Unstable (flaky) tests being ignored and not accounted for separately by their metrics leads to a grossly distorted view of the branch's stability and the effectiveness of prioritization. Moving both the Moving Flakiness Index and Feature-Importance Coverage to separate dashboards gives an undiluted view into how well the model can correspond risk areas with existing test coverage, which in turn helps respond quickly to quality degradation.

A balanced and iterative approach to selecting and monitoring automated test quality metrics ultimately ensures both the technical accuracy of ML models and the economic benefits of the entire CI/CD process. A classical measure of coverage and stability, detailed ML metrics, and economic coefficients together guarantee high reliability of predictions, minimize the costs of CI resources, and keep feedback speed at a level that retains developers' cognitive context. Regular calibration of business coefficients, continuous monitoring of Flakiness Index and Feature-Importance Coverage, and flexible retraining of models when code or infrastructure changes—all these practices serve as the foundation for the long-term effectiveness of a test strategy in an automated environment where machine learning plays a key role.

References

1. H. Krasner, "Cost of Poor Software Quality in the U.S.: A 2022 Report," CISQ, Dec. 16, 2022. https://www.it-cisq.org/the-cost-of-poor-quality-software-in-the-us-a-2022-report/ (accessed May 02, 2025).
2. M. Simonova, "Costly Code: The Price Of Software Errors," Forbes, Dec. 26, 2023. https://www.forbes.com/councils/forbestechcouncil/2023/12/26/costly-code-the-price-of-software-errors/ (accessed May 03, 2025).
3. J. M. Zhang, M. Harman, and Y. Liu, "Machine Learning Testing: Survey, Landscapes and Horizons," arXiv, Jun. 2019, doi: https://doi.org/10.48550/arxiv.1906.10742.
4. M. Z. Naser and A. H. Alavi, "Error Metrics and Performance Fitness Indicators for Artificial Intelligence and Machine Learning in Engineering and Sciences," Architecture, Structures and Construction, Nov. 2021, doi: https://doi.org/10.1007/s44150-021-00015-8.
5. M. Machalica, A. Samylkin, M. Porth, and S. Chandra, "Predictive Test Selection," arXiv, Jan. 2018, doi: https://doi.org/10.48550/arxiv.1810.05286.
6. F. Leinen, D. Elsner, A. Pretschner, A. Stahlbauer, M. Sailer, and Elmar Jürgens, "Cost of Flaky Tests in Continuous Integration: An Industrial Case Study," Proceedings of 2024 IEEE Conference on Software Testing, Verification and Validation (ICST), pp. 329–340, May 2024, doi: https://doi.org/10.1109/icst60714.2024.00037.
7. O. Parry, G. Kapfhammer, M. Hilton, and P. McMinn, "Systemic Flakiness: An Empirical Analysis of Co-Occurring Flaky Test Failures," arXiv, 2025. https://arxiv.org/abs/2504.16777 (accessed May 08, 2025).
8. R. Featherman, K. J. Yang, B. Roberts, and M. D. Ernst, "Evaluation of Version Control Merge Tools," arXiv, pp. 831–83, Oct. 2024, doi: https://doi.org/10.1145/3691620.3695075.

9. A. Patel, “Handling Flaky Tests at Scale: Auto Detection & Suppression,” Slack Engineering, Apr. 05, 2022. https://slack.engineering/handling-flaky-tests-at-scale-auto-detection-suppression/ (accessed May 08, 2025).
10. J. Rose, “The 2023 State of Software Delivery,” Circleci, 2024. Accessed: May 10, 2025. [Online]. Available: https://circleci.com/landing-pages/assets/CircleCI-The-2023-State-of-Software-Delivery.pdf
11. A. Igareta, “Navigating Imbalanced Datasets: Beyond ROC & GINI,” Klarna Engineering, Nov. 09, 2023. https://engineering.klarna.com/stop-misusing-roc-curve-and-gini-navigate-imbalanced-datasets-with-confidence-5edec4c187d7 (accessed May 13, 2025).
12. F. Movahedi, R. Padman, and J. F. Antaki, “Limitations of receiver operating characteristic curve on imbalanced data: Assist device mortality risk scores,” The Journal of Thoracic and Cardiovascular Surgery, vol. 165, no. 4, pp. 1433-1442.e2, Apr. 2023, doi: https://doi.org/10.1016/j.jtcvs.2021.07.041.
13. “Predictive Test Selection,” Gradle, 2025. https://gradle.com/develocity/product-tour/accelerate/predictive-test-selection/ (accessed May 17, 2025).
14. M. Ghanem et al., “Limitations in Evaluating Machine Learning Models for Imbalanced Binary Outcome Classification in Spine Surgery: A Systematic Review,” Brain Sciences, vol. 13, no. 12, pp. 1723–1723, Dec. 2023, doi: https://doi.org/10.3390/brainsci13121723.
15. M. Machalica, “Predictive test selection: A more efficient way to ensure reliability of code changes,” Engineering at Meta, Nov. 21, 2018. https://engineering.fb.com/2018/11/21/developer-tools/predictive-test-selection/ (accessed May 22, 2025).
16. “Netflix - Case Study,” Gradle. https://gradle.com/customers/story/netflix/ (accessed May 24, 2025).