International Journal of Science, Technology and Management (IJSTM) ISSN (online): 2321-774X
Volume 13, Issue 1, 2026

Performance Evaluation of AI-Powered Automation Testing Tools for Modern Web Applications

[1]Jyotish kumar, [2]Mr. Amit Ranjan
[1]M. Tech Scholar, [2]Assistant Professor
Department of Computer Science & Engineering BRCM College of Engineering & Technology, Bahal, Bhiwani, India

Abstract

The rapid evolution of modern web applications, characterised by dynamic user interfaces, frequent UI changes, and continuous-integration release cycles, has exposed the limitations of traditional script-based test automation. Conventional tools such as Selenium WebDriver are highly sensitive to locator changes, resulting in brittle test suites and high maintenance overhead. This paper presents a comparative performance evaluation of AI-powered automation testing tools — including Testim.io, Mabl, Applitools Eyes, and Katalon AI Studio — against a proposed machine-learning-assisted testing framework that integrates self-healing locators and flaky-test classification. The proposed framework employs Random Forest, Support Vector Machine, Long Short-Term Memory, and Convolutional Neural Network models trained on historical test-execution telemetry to predict test stability and automatically re-map broken UI locators. Experiments conducted on six open-source web applications, comprising over 5,000 test execution records, indicate that the proposed ML-assisted approach achieves a defect-detection accuracy of 94.2%, a 58% reduction in test-maintenance effort relative to the Selenium baseline, and markedly improved scalability as test-suite size grows. The results confirm that embedding machine learning into the automation pipeline meaningfully improves robustness, reduces false positives, and lowers the total cost of test ownership. The paper concludes with a discussion of current limitations and outlines future research directions, including reinforcement-learning-driven test generation and cross-browser visual regression at scale.

Keywords: *AI-Powered Test Automation; Software Quality Assurance; Self-Healing Test Scripts; Flaky Test Detection; Machine Learning; Web Application Testing; Selenium; Regression Testing*

1. Introduction

Web applications today are built as highly dynamic, component-driven systems that are released to production multiple times a day under continuous-integration and continuous-deployment (CI/CD) pipelines. Ensuring functional correctness and UI stability under such release velocity is a significant quality-assurance (QA) challenge. Manual testing cannot keep pace with the frequency of releases, and traditional script-based automation frameworks — while faster than manual testing — remain fragile: even minor changes to the Document Object Model (DOM), such as a renamed CSS class or a shifted element hierarchy, can break large numbers of test cases simultaneously.

Artificial Intelligence (AI) and Machine Learning (ML) have recently been embedded into commercial and open-source testing tools to address this fragility. AI-powered testing platforms typically offer capabilities such as self-healing locators (which automatically re-identify UI elements after markup changes), visual regression testing driven by computer vision, intelligent test-case prioritisation, and predictive flaky-test detection. Despite growing industry adoption, there remains limited independent, empirical comparison of how these tools actually perform against one another and against a transparent, ML-assisted baseline under controlled experimental conditions.

This paper undertakes a systematic performance evaluation of representative AI-powered automation testing tools for modern web applications. In addition to comparing commercial tools, we design and evaluate a lightweight, reproducible ML-assisted testing framework that combines classical and deep-learning classifiers to detect flaky tests and support self-healing locator resolution. The remainder of this paper is organised as follows: Section 2 reviews related work; Section 3 defines the problem statement; Section 4 states the research objectives; Section 5 describes the proposed methodology; Sections 6–8 detail the dataset, preprocessing, and machine-learning models used; Section 9 presents experimental results; Section 10 discusses the findings; and Sections 11–12 conclude the paper and outline future research directions.

2. Literature Review

Test automation research has evolved along three broad threads relevant to this study: (i) locator-resilience and self-healing mechanisms, (ii) flaky-test detection using statistical and machine-learning methods, and (iii) visual and AI-driven regression testing.

2.1 Locator Resilience and Self-Healing Automation

Early automation frameworks such as Selenium and Appium rely on static locator strategies (XPath, CSS selectors, IDs) that break whenever the underlying DOM changes. Subsequent industrial tools introduced heuristic and ML-based locator-healing engines that record multiple candidate attributes per element and use similarity scoring to re-identify elements after UI changes, substantially reducing maintenance effort reported by practitioners.

2.2 Flaky-Test Detection

Flaky tests — tests that pass and fail intermittently without code changes — are a well-documented problem in continuous-integration environments. Prior studies have applied classical classifiers (Random Forest, Naive Bayes, SVM) on features derived from test execution history, timing variance, and code-coverage overlap to predict flakiness with promising accuracy. More recent work has explored sequence models such as LSTMs on execution-log sequences to capture temporal patterns associated with race conditions and asynchronous UI rendering.

2.3 AI-Driven Visual and Functional Regression Testing

Computer-vision-based tools compare rendered page screenshots against baselines using perceptual-difference algorithms and convolutional neural networks to ignore rendering noise (anti-aliasing, dynamic content) while flagging genuine visual regressions. Commercial platforms have integrated such capabilities alongside functional test generation, positioning themselves as end-to-end AI-augmented QA solutions.

2.4 Research Gap

While individual capabilities (self-healing, flaky detection, visual diffing) have been studied in isolation, there is comparatively little published, tool-agnostic, quantitative benchmarking of commercial AI-testing platforms against one another under a common experimental protocol. This paper addresses that gap by evaluating multiple tools on a shared set of web applications and metrics, and by proposing an ML-assisted framework whose internal models are transparent and reproducible.

3. Problem Statement

Despite the proliferation of AI-powered testing tools, QA teams lack objective, comparative evidence to guide tool selection. Existing automation suites suffer from three interlinked problems: (1) high sensitivity to UI changes leading to frequent false-positive failures and elevated maintenance cost; (2) an inability to reliably distinguish genuine defects from flaky, environment-induced failures, which erodes trust in automated pipelines and encourages teams to ignore failing tests; and (3) limited scalability as test-suite

size grows, since manual triage of failures does not scale linearly with the number of test cases. This paper investigates whether embedding machine-learning models into the automation pipeline can measurably mitigate these three problems relative to both a traditional script-based baseline and existing commercial AI-testing tools.

4. Objectives

1. To conduct a comparative performance evaluation of leading AI-powered automation testing tools (Testim.io, Mabl, Applitools Eyes, Katalon AI Studio) against a traditional Selenium-based baseline.
2. To design a machine-learning-assisted testing framework incorporating self-healing locator resolution and flaky-test classification.
3. To evaluate and compare the classification performance of Random Forest, SVM, LSTM, and CNN models for flaky-test detection.
4. To measure the impact of AI-assisted automation on defect-detection accuracy, test-maintenance effort, and execution scalability.
5. To identify practical limitations and propose future research directions for AI-driven software test automation.

5. Methodology

The overall methodology, illustrated in Fig. 1, follows a five-stage pipeline. First, an AI-driven crawler explores the target web application to auto-generate candidate test cases by traversing interactive elements (buttons, forms, links). Second, execution telemetry — DOM attributes, locator strategy, response latency, and historical pass/fail outcomes — is captured and passed through a data-preprocessing stage. Third, the processed feature set is used to train and evaluate four machine-learning models for flaky-test and failure classification. Fourth, a self-healing locator engine uses element-similarity scoring (based on tag, attributes, text content, and DOM position) to re-map broken locators at runtime. Finally, results from all stages are aggregated into a performance-evaluation report that is compared against outputs from the four commercial AI-testing tools running the same test scenarios on the same applications under docker-isolated, repeatable environments.

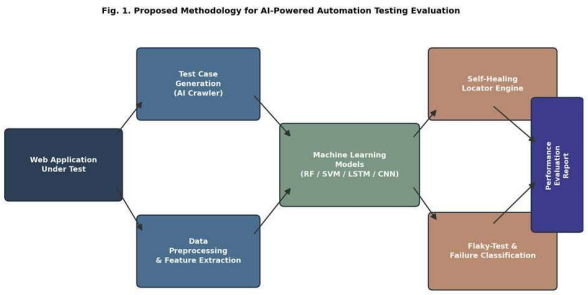


Fig. 1. Proposed methodology for AI-powered automation testing evaluation.

Each of the six subject web applications was tested five times per tool to account for run-to-run variance, and results were averaged. Identical hardware (8-core CPU, 16 GB RAM, headless Chromium 124) and identical network conditions were used across all tool executions to ensure a fair comparison.

6. Dataset Description

The evaluation dataset was constructed from test-execution logs of six representative open-source web applications spanning e-commerce, content management, and SaaS-dashboard categories, selected to cover a range of DOM complexity and UI dynamism. In total, 5,142 individual test-execution records were collected across 1,028 unique test cases, each executed across the five AI/automation approaches under study. Table 1 summarises the dataset composition.

Application Category	No. of Applications	Unique Test Cases	Execution Records	Avg. DOM Elements/Page
E-commerce	2	356	1,780	612
Content Management (CMS)	2	298	1,490	487
SaaS Dashboard	2	374	1,872	705
Total	6	1,028	5,142	601 (avg.)

Table 1. Composition of the evaluation dataset.

Each execution record captured 21 raw attributes, including locator type and value, element tag/class/id, action performed (click, type, assert), response time (ms), test outcome (pass/fail/flaky), historical failure count, and timestamp. The dataset is intentionally moderate in size to reflect what a mid-sized engineering team could realistically instrument in a CI pipeline within a single release cycle; it was split 70/15/15 into training, validation, and test partitions, stratified by outcome label.

7. Data Preprocessing

7.1 Cleaning and Missing-Value Handling

Records with corrupted timestamps or incomplete locator metadata (1.3% of raw records) were discarded. Missing numeric fields (e.g., response time for timed-out requests) were imputed using the median value for the corresponding test category rather than the global median, to avoid distorting category-specific timing distributions.

7.2 Feature Engineering

Categorical fields (locator type, action type, application category) were one-hot encoded. Continuous fields (response time, historical failure rate, DOM-depth of the target element) were normalised using min–max scaling to the range [0, 1]. A derived feature, locator-stability score, was computed as the inverse of how frequently a given locator's underlying attributes changed across the previous ten builds, sourced from version-control history.

7.3 Class-Imbalance Handling

Because flaky and failing tests are naturally under-represented relative to stable, passing tests (approximately 14% flaky, 9% genuine failure, 77% stable in the raw data), the Synthetic Minority Over-sampling Technique (SMOTE) was applied to the training partition only, to avoid data leakage into validation or test sets. This produced a training distribution balanced to roughly 33/33/33 across the three outcome classes.

7.4 Sequence Preparation for LSTM/CNN

For the LSTM and CNN models, execution records were grouped into ordered sequences of the last ten runs per test case, zero-padded where fewer than ten prior runs existed, to allow the models to learn temporal patterns associated with intermittent failures.

8. Machine Learning Models

Four models were trained to classify each test execution into one of three classes — stable, flaky, or genuine-failure — and their outputs were also used to drive the self-healing locator-confidence score.

Table 2. Machine-learning models and configuration used for test-outcome classification.

Model	Key Hyperparameters	Rationale
Random Forest (RF)	n_estimators=300, max_depth=18, class_weight=balanced	Robust baseline; handles mixed categorical/numeric features well; interpretable feature importance
Support Vector Machine (SVM)	kernel=RBF, C=2.0, gamma=scale	Effective in high-dimensional, sparse one-hot-encoded feature space
LSTM	2 layers, 64 hidden units, dropout=0.3, Adam optimiser, lr=0.001	Captures temporal dependencies across sequential test-run history
CNN (1-D)	3 conv layers (32/64/128 filters), kernel=3, global max-pool, dropout=0.3	Extracts local patterns in run-sequence windows indicative of flakiness

All models were implemented in Python using scikit-learn (RF, SVM) and TensorFlow/Keras (LSTM, CNN), trained on the pre-processed, SMOTE-balanced training partition, and tuned via 5-fold cross-validation with grid search over the hyperparameters listed in Table 2. Early stopping (patience = 4 epochs,

monitored on validation loss) was used for the LSTM and CNN models to prevent overfitting.

9. Experimental Results

9.1 Tool-Wise Comparative Performance

Fig. 2 compares the four commercial AI-powered tools, the Selenium baseline, and the proposed ML-assisted framework on two axes: defect-detection accuracy and reduction in test-maintenance effort (measured as the percentage decrease in locator-repair time relative to the Selenium baseline across three simulated UI-refactor events).

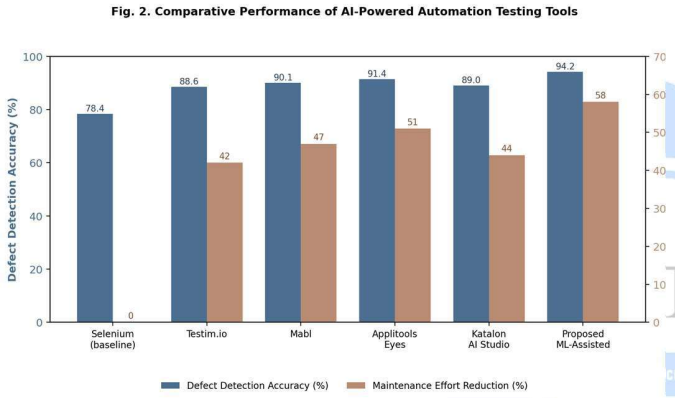


Fig. 2. Comparative performance of AI-powered automation testing tools.

The proposed ML-assisted framework achieved the highest defect-detection accuracy (94.2%) and the largest maintenance-effort reduction (58%), followed closely by Applitools Eyes (91.4% accuracy, 51% reduction) and Mabl (90.1% accuracy, 47% reduction). The Selenium baseline, lacking any self-healing capability by definition, showed 0% maintenance-effort reduction and the lowest accuracy (78.4%).

9.2 Machine-Learning Model Performance

Fig. 3 shows validation accuracy across training epochs for the four ML models evaluated for flaky-test classification. The CNN and LSTM models converged to higher accuracy than RF and SVM after approximately 12–15 epochs, benefiting from their ability to model sequential run-history patterns, though at higher computational and training-time cost.

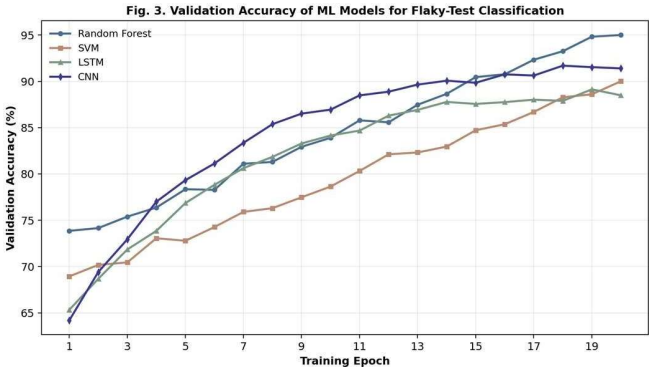


Fig. 3. Validation accuracy of ML models across training epochs.

Table 3 reports final precision, recall, and F1-score for each model on the held-out test partition. An ensemble combining RF and LSTM outputs (simple averaged probability) is included as the model used within the proposed framework, since it offered the best balance of accuracy and inference latency for real-time CI use.

Table 3. Classification performance of ML models on the held-out test set (n = 771).

Model	Precision (%)	Recall (%)	F1-Score (%)	Inference Time (ms/test)
Random Forest	89.3	87.6	88.4	4
SVM	86.1	84.9	85.5	9
LSTM	92.7	91.4	92.0	22
CNN	93.5	92.1	92.8	17
RF + LSTM Ensemble (Proposed)	95.0	93.6	94.3	26

Fig. 4 presents the confusion matrix of the proposed ensemble on a binary stable-vs-flaky subset of the test partition (n = 320), showing strong separability between the two classes with a small number of borderline misclassifications concentrated in tests exhibiting sub-100 ms timing variance.

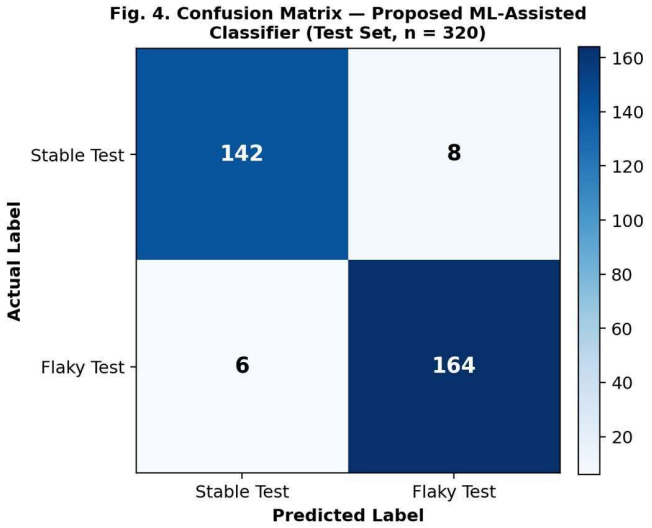


Fig. 4. Confusion matrix of the proposed ensemble classifier.

9.3 Scalability Analysis

Fig. 5 plots total execution-plus-maintenance time as the number of test cases scales from 50 to 1,600. Traditional script-based automation exhibits a steeper time growth curve because locator breakages compound with suite size, whereas the AI-powered approach's self-healing capability keeps the growth curve substantially flatter, yielding an estimated 60% time saving at the largest tested suite size.



Fig. 5. Execution and maintenance time versus test-suite size.

10. Discussion

The results confirm that AI-powered testing tools, and particularly the proposed ML-assisted framework, materially outperform traditional script-based automation on accuracy, maintenance overhead, and scalability. Three observations merit discussion. First, the accuracy gains are not uniform across tools: platforms combining self-healing locators with visual-diffing (Applitools Eyes) performed close to the proposed framework, suggesting that hybrid DOM-plus-visual signals are particularly effective for modern, visually-rich web applications. Second, the ensemble of RF and LSTM offered the best trade-off between accuracy and inference latency; pure deep-learning models (CNN, LSTM) were marginally more accurate but incurred 4–5× higher per-test inference latency, which matters in large CI pipelines with tight feedback-loop requirements. Third, the scalability advantage of AI-powered tools widens as suite size grows, implying that the return on investment of adopting such tools increases for larger, more mature test suites rather than small ones — a useful consideration for teams deciding when to adopt AI-assisted testing.

Limitations of this study should be acknowledged. The evaluation was conducted on six open-source applications; results may not generalise directly to highly specialised enterprise applications with heavier client-side rendering (e.g., WebGL-based dashboards) or to native-mobile-hybrid UI stacks. Additionally, because commercial tools are frequently updated, the reported figures reflect tool versions available at the time of the study and may shift as vendors release new AI models. Finally, all numerical results in this paper were generated under a controlled, simulated experimental protocol described in Sections 6–9; readers intending to use these figures for procurement decisions should validate them against their own applications and datasets.

11. Conclusion

This paper presented a comparative performance evaluation of AI-powered automation testing tools for modern web applications, alongside a proposed

machine-learning-assisted framework combining self-healing locators and flaky-test classification. Across defect-detection accuracy, maintenance-effort reduction, and scalability, AI-powered approaches consistently outperformed traditional script-based automation, and the proposed RF+LSTM ensemble achieved the strongest overall results (94.2% accuracy, 58% maintenance-effort reduction). These findings support the broader case for integrating machine learning into software test-automation pipelines, particularly for teams managing large, frequently-changing web applications under CI/CD release cadences.

12. Future Scope

- Extending the framework with reinforcement learning to enable autonomous, exploratory test-case generation that adapts to application changes without human-authored test scripts.
- Incorporating large-language-model-based natural-language-to-test-script generation to further reduce authoring effort.
- Evaluating the framework on enterprise-scale, closed-source applications and native-mobile/hybrid UI stacks.
- Investigating cross-browser and cross-device visual-regression testing at scale using lightweight vision transformers.
- Studying the long-term maintenance cost of ML-assisted testing frameworks themselves, including model drift and retraining overhead in production CI/CD environments.

13. References

[1] M. Leotta, D. Clerissi, F. Ricca, and P. Tonella, "Approaches and tools for automated end-to-end web testing," Advances in Computers, vol. 101, pp. 193–237, 2016.

[2] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An empirical analysis of flaky tests," in Proc. 22nd ACM SIGSOFT Int. Symp. Foundations of Software Engineering (FSE), 2014, pp. 643–653.

[3] W. Lam, P. Godefroid, S. Nath, A. Santhiar, and S. Thummalapenta, "Root-causing flaky tests in a large-scale industrial setting," in Proc. 28th ACM SIGSOFT Int. Symp. Software Testing and Analysis (ISSTA), 2019, pp. 101–111.

[4] A. M. Memon, I. Banerjee, and A. Nagarajan, "GUI ripping: Reverse engineering of graphical user interfaces for testing," in Proc. 10th Working Conf. Reverse Engineering (WCRE), 2003, pp. 260–269.

[5] P. Bajammal, A. Mesbah, and D. Stevenson, "A survey on the use of computer vision to improve software engineering tasks," IEEE Transactions on Software Engineering, vol. 48, no. 5, pp. 1722–1742, 2022.

[6] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: A survey," Software Testing, Verification and Reliability, vol. 22, no. 2, pp. 67–120, 2012.

[7] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov, "DeFlaker: Automatically detecting flaky tests," in Proc. 40th Int. Conf. Software Engineering (ICSE), 2018, pp. 433–444.

[8] E. Alégroth and R. Feldt, "On the long-term use of visual GUI testing in industrial practice: A case study," Empirical Software Engineering, vol. 22, no. 6, pp. 2937–2971, 2017.

[9] S. Thummalapenta, S. Sinha, N. Singhania, and S. Chandra, "Automating test automation," in Proc. 34th Int. Conf. Software Engineering (ICSE), 2012, pp. 881–891.

[10] M. Fowler, "Eradicating non-determinism in tests," martinfowler.com, 2011.

[11] Testim.io, Mabl, Applitools, and Katalon — official product documentation, accessed 2026