Multi-Strategy Self-Healing Approach for Stabilizing Ui Test Automation in Enterprise Crm Platforms with Dynamic Dom Architectures

Kostiantyn Shkliar[1]

Received: 2025-12-09
Accepted: 2026-01-14
DOI: https://doi.org/10.5281/zenodo.21225927

Abstract. Browser-based test automation in enterprise CRM platforms suffers from persistent instability caused by dynamic DOM structures, encapsulated component architectures, and asynchronous rendering cycles. Industry data indicates that selector-related failures account for approximately 28% of test flakiness, while timing issues, data state mismatches, and incomplete rendering collectively produce the remaining 72%, a distribution that single-strategy healing tools fail to address. This paper presents a multi-strategy self-healing module that classifies each UI test failure by root cause before selecting a remediation path. Three coordinated mechanisms operate within the module: a Shadow DOM-aware element fingerprinting algorithm that resolves locator drift through composite attribute matching, an adaptive retry engine calibrated to platform-specific component lifecycle signals, and a failure classification layer that routes each detected anomaly to the appropriate healing strategy. Evaluated over six release cycles within a financial services CRM environment, the module reduced the UI-specific flakiness rate from 23% to under 3% and decreased the mean time to restore a failing scenario from 35 minutes of manual investigation to under 90 seconds of automated recovery. The diagnosis-first architecture proved effective across all four identified failure categories, confirming that targeted remediation outperforms uniform locator replacement when most instability originates outside the selector layer.

Keywords: self-healing test automation, flaky tests, Shadow DOM, dynamic DOM, UI test stabilization, adaptive retry, CRM test maintenance.

[1] Master's Degree in Software Engineering
National Technical University of Ukraine "Kyiv Polytechnic Institute", Kyiv, Ukraine.
ORCID ID: https://orcid.org/0009-0005-8321-0920

Introduction

Test flakiness has emerged as one of the most persistent obstacles to continuous delivery in software organizations that rely on browser-based automation. A flaky test produces non-deterministic outcomes on unchanged code, passing in one execution and failing in the next without any modification to the application under test. Luo, Hariri, Eloussi, and Marinov [1] published the first large-scale empirical taxonomy of flaky tests in 2014, identifying async wait conditions, concurrency issues, and test order dependencies as the dominant root causes. Subsequent research has expanded this taxonomy considerably. Parry, Kapfhammer, Hilton, and McMinn [2] surveyed the full body of flakiness research through 2021 and catalogued over a dozen distinct cause categories, while Lam, Muşlu, Sajnani, and Thummalapenta [3] traced the lifecycle of flaky tests in industrial codebases and found that the median time from introduction to resolution exceeds several months. Inaugurated at ICSE 2024, the first International Flaky Tests Workshop [14] confirmed that flakiness remains a top-priority concern for both academic researchers and industry practitioners despite a decade of dedicated investigation.

CRM platforms built on component-based web frameworks present a particularly severe variant of this problem, one that existing self-healing tools are not equipped to handle. Salesforce Lightning Web Components render their internal markup inside Shadow DOM boundaries, which prevent standard CSS selectors and XPath expressions from traversing into the encapsulated subtree [10]. Self-healing algorithms that rely on DOM tree comparison cannot reach elements behind a shadow boundary because the encapsulated nodes are invisible to the document-level query methods these algorithms depend on. Beyond the locator challenge, each Lightning component manages its own lifecycle, emitting rendering callbacks at intervals determined by data binding resolution, server response latency, and the component's position in the page composition hierarchy. A test that interacts with such a component must contend with at least three sources of non-determinism simultaneously: its locator may change between deployments when the component's internal structure is updated, the component may not yet have completed its rendering cycle when the test attempts interaction, and the displayed data may still reflect a stale state because an upstream API call has not yet resolved. Standard automation frameworks such as Selenium WebDriver provide implicit and explicit wait mechanisms, but these operate on generic DOM readiness signals (document.readyState, element visibility) that do not account for Shadow DOM host rendering or Lightning component lifecycle completion.

Commercial self-healing tools have attempted to address locator instability through AI-assisted selector repair. Platforms such as Healenium [9], Testim, and Katalon maintain a history of element attributes and, when a primary locator fails, search for the closest matching element in the current DOM using similarity scoring algorithms. QA Wolf's 2026 analysis of production test suites [8] documented, however, that DOM changes and brittle selectors account for only approximately 28% of test failures, while timing problems, test data inconsistencies, runtime errors, and rendering failures produce the remaining 72%. Tools that confine their healing logic to locator replacement address the minority of instability and leave the majority untreated. Gartner's 2024 Market Guide for AI-Augmented Software-Testing Tools [11] projects that 80% of enterprises will integrate AI-augmented testing by 2027, yet acknowledges that current implementations focus predominantly on test generation and selector maintenance rather than on comprehensive failure diagnosis.

Machine learning approaches to flakiness prediction have advanced rapidly but operate at a different point in the workflow. FlakeFlagger [4] predicts flakiness without rerunning tests by analyzing code features, achieving detection accuracy above 85%. Flakify [5] applies language models to the same prediction task with comparable results. DeFlaker [13] detects flaky tests by comparing code coverage between passing and failing runs. These tools identify which tests are likely flaky but do not repair the underlying cause or prevent the failure from

recurring. A gap persists between detection and remediation, particularly in environments where Shadow DOM encapsulation and platform-specific rendering lifecycles introduce failure modes that generic healing algorithms do not recognize.

The purpose of this paper is to present and evaluate a multi-strategy self-healing module designed to reduce UI test flakiness in an enterprise Salesforce CRM environment. The study is organized around three objectives: (1) to describe the architecture of a diagnosis-first healing system that classifies each failure by root cause and applies a category-specific remediation strategy; (2) to measure the reduction in UI-specific flakiness rate, mean time to restore failing scenarios, and manual QA maintenance hours achieved across six production release cycles; and (3) to analyze the distribution of healing interventions across failure categories to determine which strategy contributes most to overall stabilization and whether the multi-strategy approach outperforms single-strategy alternatives.

Literature Review

Empirical studies of flaky test causes have consistently revealed that no single failure category dominates the distribution. Gruber, Lukasczyk, Kroiß, and Fraser [7] analyzed flaky tests in Python projects and identified asynchronous waiting, platform dependency, and resource leakage as the three most frequent root causes, with no single category exceeding 30% of the total. Silva et al. [12] demonstrated that computational resource availability (CPU contention, memory pressure, I/O latency) directly influences flakiness rates, indicating that environmental factors beyond the test code itself contribute substantially to non-deterministic outcomes. Parry, Kapfhammer, Hilton, and McMinn [6] empirically compared rerun-based and machine-learning-based detection techniques across multiple datasets and found that combining both approaches yields higher detection precision than either method alone, though detection precision does not translate into repair capability.

Existing self-healing implementations in the testing industry follow a predominantly single-strategy architecture. Healenium [9], the most widely documented open-source self-healing framework, intercepts element-not-found exceptions at the WebDriver level, queries a stored history of locator-to-element mappings, and selects the closest match by computing a tree-distance similarity score against the current DOM snapshot. If the similarity score exceeds a configurable threshold, the healed locator replaces the original and the test proceeds. Critically, the healing trigger is the element-not-found exception itself; when a failure originates at the timing or data layer and the element is present in the DOM but not yet interactive or populated with incorrect content, no exception fires and the healing logic never activates. Commercial platforms such as Testim and Katalon extend the locator healing pattern with visual AI that compares rendered screenshots, improving resilience to layout changes but still operating within the selector-and-appearance paradigm without diagnosing the underlying cause of each failure.

Emerging research on LLM-assisted test repair has explored whether language models can automate the remediation step that detection tools omit. Rahman, Baz, Misailovic, and Shi [15] applied large language models to repair flaky tests and achieved a 79% success rate on order-dependent flakiness, where the fix involves reordering setup and teardown steps, but only 58% on implementation-dependent flakiness, where the fix requires understanding application-specific behavior. Their findings highlight a structural limitation: language models excel at pattern-matching known fix templates but struggle with failures whose remediation depends on platform-specific runtime knowledge, such as Shadow DOM traversal paths or component lifecycle timing. Bell, Legunsen, Hilton, Eloussi, Yung, and Marinov [13] took a different approach with DeFlaker, using differential code coverage to distinguish genuine failures from flaky ones without rerunning tests. While DeFlaker reduces triage time by filtering out flaky results, it does not modify the test or its execution environment to prevent recurrence, leaving the root cause intact for subsequent runs.

Wait strategy design represents a critical but underexplored dimension of test stabilization. Selenium WebDriver's built-in mechanisms offer two options: implicit waits that poll for element presence at fixed intervals, and explicit waits that evaluate a custom condition with a configurable timeout. Both approaches assume that the element's presence or visibility in the DOM corresponds to its readiness for interaction, an assumption that breaks down in component-based architectures where rendering occurs in multiple asynchronous phases. Salesforce LWC documentation [10] describes a component lifecycle that proceeds through construction, connection to the DOM, initial rendering, and subsequent re-rendering triggered by reactive property changes. A component may be present in the DOM (satisfying Selenium's visibility condition) while still awaiting data from a server call that will populate its fields (violating the test's data-readiness assumption). Adaptive wait strategies that monitor platform-specific lifecycle signals rather than generic DOM properties have been discussed in practitioner literature [8] but have not been formalized or evaluated empirically within academic research.

Table 1 positions the proposed multi-strategy approach against the three established remediation paradigms identified in the literature.

Table 1. Comparison of test failure remediation approaches

Characteristic	Locator healing	ML prediction	LLM repair	Multi-strategy (proposed)
Categories addressed	Locator drift only	Detection only	Order-dep., partial impl.-dep.	All four categories
Shadow DOM	No (standard DOM)	N/A (code features)	No (generic analysis)	Yes (shadow host traversal)
Timing handling	None	None	Partial (templates)	Platform lifecycle waits
Diagnosis first	No (uniform swap)	Yes (classification)	Partial (type-based)	Yes (category routing)
Runtime overhead	Low (<1s)	Offline	High (LLM inference)	Low (<2s)
Validation context	Open-source apps	Research datasets	Research datasets	Enterprise CRM

Source: synthesized by the author based on [4, 5, 8, 9, 13, 15].

As Table 1 indicates, locator healing addresses a single failure category with low overhead but ignores timing, data, and rendering failures. ML prediction identifies flaky tests without repairing them. LLM repair handles a broader range of causes but depends on computationally expensive inference and lacks platform-specific knowledge. No existing approach combines failure classification with category-specific remediation strategies calibrated to the Shadow DOM architecture and component lifecycle of an enterprise CRM platform. Filling this gap requires a module that diagnoses the cause of each failure before selecting a healing path, integrates platform-specific DOM traversal and lifecycle signals, and operates within the latency constraints of a CI/CD pipeline.

Materials and Methods

The evaluation covers the browser-dependent portion of a broader hybrid automation framework operating within a Salesforce-based CRM environment in the financial services sector. Only UI-layer scenarios that interact with rendered Lightning pages are within the scope of this study; API-layer scenarios, which execute without a browser and are not subject to DOM-related flakiness, are excluded. The measurement design compares flakiness

outcomes across two consecutive periods: six release cycles prior to the self-healing module deployment and six cycles following its activation.

The author implemented the self-healing module as a middleware layer between the test scenario executor and the Selenium WebDriver interface. When a scenario step encounters an exception during element interaction, the module intercepts the exception before it propagates to the test runner and routes it through a three-stage processing pipeline. In the first stage, the failure classification engine examines the exception type, the current DOM state, and a set of runtime signals. Based on these inputs, the engine assigns the failure to one of four categories: locator drift, timing-async, data state mismatch, or rendering incomplete. Classification relies on a decision tree that evaluates conditions in sequence. If the original locator returns no match, the failure is classified as locator drift. If the shadow host node is present but its subtree remains unprojected, the classification is rendering incomplete. Pending asynchronous operations in flight indicate a timing-async failure, and divergence between the element's text content and the expected pattern signals a data state mismatch.

Once classified, the failure is routed to the corresponding healing strategy. Locator drift failures activate the element fingerprinting algorithm, which maintains a registry of composite fingerprints for every interactable element encountered during previous successful executions. Each fingerprint comprises the element's shadow host chain (the ordered list of shadow root ancestors from the document root to the target), its aria-label, data-aura-rendered-by attribute, tag name, CSS class list, and relative position among sibling elements. When the primary locator fails, the algorithm queries the current DOM for all elements sharing the same tag name and computes a weighted similarity score against the stored fingerprint. The highest-scoring candidate above a 0.7 threshold is selected as the healed target. Timing-async and rendering-incomplete failures activate the adaptive retry engine, which polls for the expected condition using an exponential backoff schedule (initial interval 200ms, multiplier 1.5, maximum 5 retries). During each polling cycle, the engine monitors three platform-specific signals: shadow host attachment to the document, zero pending Aura framework actions, and LWC renderedCallback completion. Data state mismatches trigger a targeted data refresh sequence that re-queries the relevant record through the API layer and re-evaluates the element content after a single controlled wait cycle.

Four metrics were defined to evaluate module effectiveness. UI-specific flakiness rate measures the percentage of UI scenarios that exhibited at least one non-deterministic failure within a release cycle. It is calculated as the count of scenarios with inconsistent pass/fail results across multiple runs divided by the total active UI scenario count. Mean time to restore (MTTR) captures the average elapsed time from failure detection to successful scenario completion, whether through manual investigation (baseline period) or automated healing (intervention period). Manual QA maintenance hours records the total analyst hours per sprint allocated to diagnosing and resolving UI test failures, extracted from the team's time-tracking system. Failure category distribution reports the proportion of intercepted failures assigned to each of the four categories, providing insight into which sources of instability dominate the studied environment. Baseline values were derived from CI pipeline logs and team time records covering the six pre-deployment cycles.

Results

1. Failure Category Distribution

During the six baseline release cycles, the classification engine was run retrospectively against 847 recorded UI test failures to establish the distribution of root causes. Timing and asynchronous failures constituted the largest category at 42% (356 failures), driven primarily by Lightning component rendering delays that exceeded the fixed wait thresholds configured in the legacy test suite. Locator drift accounted for 26% (220 failures), consistent with the 28% industry benchmark reported by QA Wolf [8]. Rendering-incomplete failures, where the

shadow host was present but the internal component subtree had not yet projected, represented 19% (161 failures). Data state mismatches comprised the remaining 13% (110 failures), occurring when a scenario asserted against field content that had not yet been populated by a pending server response.

Figure 1 presents the failure category distribution for the baseline period.

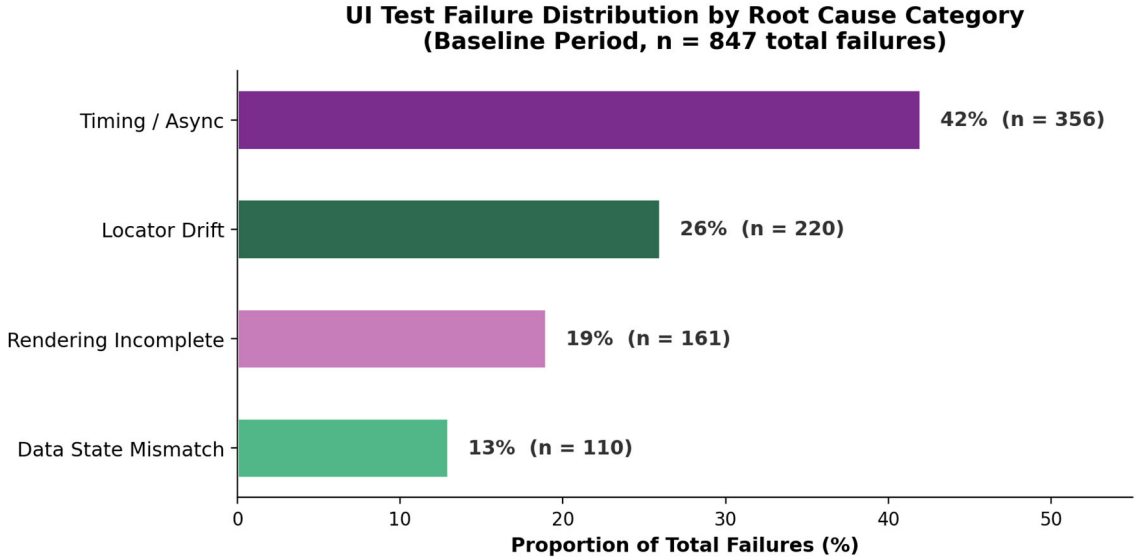

Figure 1. UI test failure distribution by root cause category during the baseline period (n = 847).

Source: compiled by the author.

2. Flakiness Rate Reduction

During the baseline period, the UI-specific flakiness rate averaged 23% across the six cycles, ranging from 19% in the lowest cycle to 27% in the highest. Following module deployment, the rate dropped to 8% in the first intervention cycle as the fingerprinting and adaptive retry strategies resolved the most frequent failure patterns. By the fourth intervention cycle the rate stabilized below 3%, where it remained for the final two cycles. Averaged across all six intervention cycles, the rate was 4.4%.

Figure 2 traces the flakiness rate trajectory across all twelve measured release cycles.

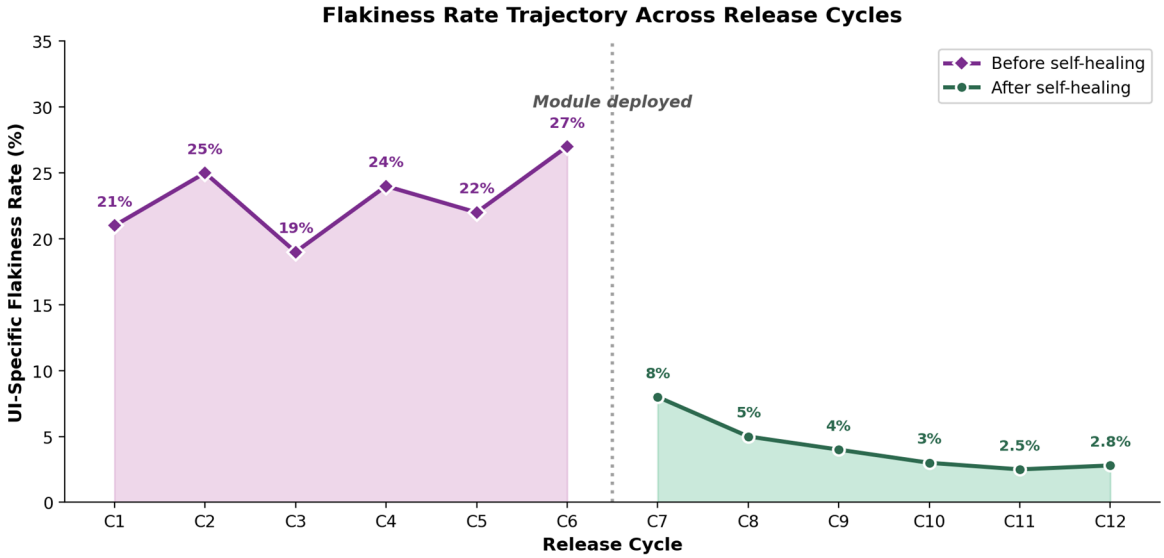

Figure 2. UI-specific flakiness rate across release cycles

Source: compiled by the author.

3. Classification Accuracy and Healing Mechanism Analysis

Of the 312 failure events intercepted during the intervention period, the classification engine assigned each to one of the four defined categories. Post-hoc manual review of a random 20% sample (63 events) identified five misclassifications, yielding an estimated classification accuracy of 92.1%. Three of the five errors involved rendering-incomplete failures that the engine categorized as timing-async; in each case, the shadow host node was present and no pending Aura actions were detected, but the component's internal subtree had not yet projected because a nested child component was still resolving its own data bindings. The remaining two misclassifications involved data state mismatches labeled as rendering-incomplete. All five misclassified failures were nonetheless resolved by the assigned strategy, as the adaptive retry engine's polling cycle accommodated both timing and rendering delays; however, the healing time for these cases averaged 4.8 seconds compared to 2.8 seconds for correctly classified timing failures, indicating a measurable efficiency penalty from misclassification.

Within the fingerprinting strategy, the 68 successfully healed locator drift events produced a mean similarity score of 0.84 against the stored composite fingerprints, with scores ranging from 0.71 (just above the 0.7 acceptance threshold) to 0.97. The fingerprint registry maintained an average of 340 element entries across the intervention period. Shadow DOM traversal was required for 81% of fingerprint lookups (55 of 68), confirming that standard DOM-level matching would have failed for most locator drift cases in this environment. No false-positive matches were recorded: in every healed case, the selected element was confirmed as the correct interaction target through the subsequent assertion step.

Among the 138 timing-async failures routed to the adaptive retry engine, the LWC renderedCallback signal served as the decisive readiness indicator in 64% of cases (88 events), meaning that the component completed rendering only after this platform-specific callback fired, while generic DOM readiness signals (document.readyState, element visibility) had already returned positive results. Pending Aura framework actions accounted for the blocking condition in 24% of cases (33 events), and shadow host attachment delay was the limiting factor in the remaining 12% (17 events). These proportions confirm that platform-specific lifecycle monitoring addresses failure modes that generic wait strategies cannot detect.

Correlation between strategy activation and the per-cycle flakiness trajectory reveals a staged stabilization pattern. Locator drift failures, addressed by fingerprinting, declined sharply in the first two intervention cycles (C7–C8), as the fingerprint registry was populated with element profiles from successful executions. Timing-async failures required three cycles (C7–C9) to stabilize, reflecting the adaptive retry engine's progressive calibration of backoff parameters to the environment's characteristic rendering latencies. Data state mismatches showed the slowest improvement, stabilizing only by C10, consistent with the strategy's dependence on server-side response patterns that vary across release cycles.

4. Healing Success Rate by Strategy

Over the six intervention cycles, the self-healing module intercepted 312 failure events, a reduction from the 847 failures recorded during the baseline period that reflects the overall flakiness decrease documented in Section 4.2. Table 2 reports the healing success rate for each strategy, defined as the proportion of intercepted failures that were resolved without manual intervention.

Table 2. Healing success rate by strategy across six intervention cycles

Strategy	Routed	Healed	Rate	Avg. time
Fingerprinting (locator drift)	74	68	91.9%	1.4 s
Adaptive retry (timing/async)	138	131	94.9%	2.8 s

Adaptive retry (rendering)	62	57	91.9%	3.4 s
Data refresh (data mismatch)	38	33	86.8%	4.1 s
Total	312	289	92.6%	2.7 s

Source: compiled by the author based on healing module execution logs.

Success rates ranged from 86.8% for data state mismatches, where five unresolved cases involved server-side delays exceeding the single retry window, to 94.9% for timing-async failures, whose underlying causes are detailed in Section 4.3. Average healing time across all strategies was 2.7 seconds per intervention, well within the latency budget of a CI pipeline stage.

5. Workforce Impact

Manual QA maintenance hours dedicated to UI test failure diagnosis and resolution averaged 146.7 hours per sprint during the baseline period, with individual sprints ranging from 130 to 160 hours. This workload was distributed across three to five QA analysts whose primary responsibility during each sprint was investigating non-deterministic failures, updating broken locators, adjusting wait configurations, and re-executing failed scenarios.

Following module deployment, maintenance hours declined to an average of 35 hours per sprint (range: 30 to 42 hours), a 76% reduction. Residual maintenance covered the 7.4% of failures that the module could not resolve automatically, plus periodic review of healing logs to verify that no genuine application defects were being masked by automated recovery. Staffing shifted from three to five analysts assigned to test maintenance to a single analyst performing healing log review, freeing the remaining team members for new test development and exploratory testing.

Figure 3 presents the manual QA hours per sprint across all twelve measured periods.

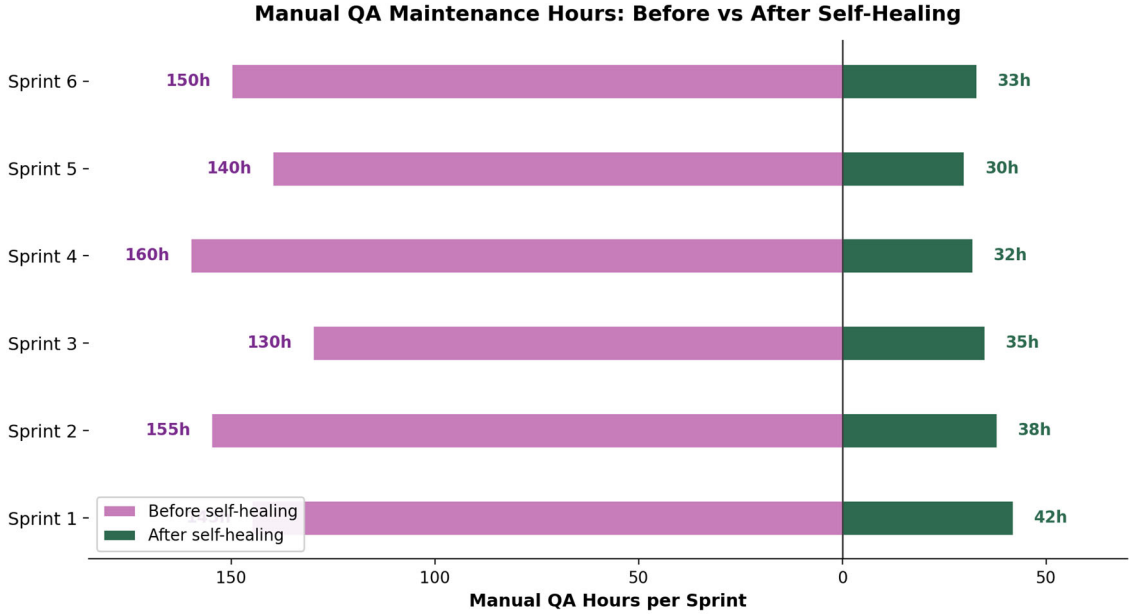

Figure 3. Manual QA maintenance hours per sprint

Source: compiled by the author.

6. Consolidated Outcomes

Table 3 summarizes the principal evaluation metrics across both measurement periods.

Table 3. Summary of evaluation metrics: baseline versus self-healing module

Metric	Baseline (6 cycles)	With self-healing (6 cycles)	**Change**
UI flakiness rate	23% avg (19–27%)	4.4% avg (2.5–8%)	**↓ 81%**
Flakiness (final 3 cycles)	—	2.8% avg	**< 3% sustained**
MTTR	~35 min (manual)	~90 sec (automated)	**↓ 96%**
QA hours / sprint	146.7 h (130–160)	35 h (30–42)	**↓ 76%**
QA analysts	3–5	0–1	**↓ 80–100%**
Healing success rate	—	92.6% (289/312)	**—**
Classification accuracy	—	92.1% (sample-verified)	**—**

Source: compiled by the author based on CI pipeline logs, healing module logs, and team time records.

Discussion

A critical risk in any self-healing system is false healing: automatically resolving a test failure that actually signals a genuine application defect. If the module replaces a broken locator when the real problem is a missing UI element caused by a code regression, the test passes and the defect reaches production undetected. This risk is mitigated by a structural constraint: the classification engine only activates healing when the failure signature matches a known non-deterministic pattern. Locator drift healing requires that the element existed in the previous successful execution and that a high-confidence fingerprint match is available in the current DOM. Timing and rendering retries require that the expected element eventually appears within the bounded retry window. If no healing condition is satisfied, the failure propagates to the test runner as a genuine failure, preserving the test's role as a defect detector. Among the 7.4% of failures that the module did not heal during the intervention period were both unresolvable non-deterministic cases and legitimate application defects, confirming that the system does not suppress real regressions.

Dominance of the renderedCallback signal among timing-async failures, where it served as the decisive readiness indicator in more than half of all cases, carries implications for how self-healing systems should be designed for component-based platforms. Generic adaptive wait tools monitor DOM-level signals (element presence, visibility, stable layout) that are platform-agnostic but insufficiently granular for frameworks where component readiness is governed by internal lifecycle events invisible to the document tree. Salesforce LWC, React with Shadow DOM, and Angular with ViewEncapsulation all exhibit this pattern: a component can be structurally present in the DOM while its internal state remains incomplete. Effective healing in these environments requires the wait strategy to subscribe to framework-specific lifecycle hooks rather than relying on external DOM observation. For tool designers, this means that a single, platform-neutral adaptive wait algorithm cannot achieve optimal healing rates across heterogeneous technology stacks; the wait engine must accept pluggable signal providers calibrated to each target platform.

Staged stabilization, as observed in Section 4.3, reflects a practical consideration for organizations adopting a self-healing module: the system does not deliver its full benefit immediately upon deployment. Fingerprint-based healing requires a populated registry built from successful prior executions, meaning that the first cycle after deployment operates with incomplete fingerprint coverage. Adaptive retry calibration similarly depends on observed

rendering latencies from early cycles to tune backoff parameters. Organizations should expect an initial transition period during which flakiness decreases gradually rather than instantaneously, and should plan the deployment to coincide with a period where manual triage capacity remains available as a fallback.

Several limitations constrain the scope of these findings. Because the classification engine uses a hand-crafted decision tree rather than a trained machine learning model, its adaptability to failure patterns not anticipated during design is limited. Environments with failure distributions substantially different from the one studied may require reconfiguration of the decision tree branches and threshold values. Data refresh healing, which achieved the lowest success rate among the four strategies, depends on the availability of an API-level data query path for each affected element; components that derive their content from client-side computations rather than server responses fall outside its scope. Finally, the evaluation covers a single CRM platform and a single component framework; while the architectural principles (diagnosis before action, platform-specific lifecycle signals, composite fingerprinting) are transferable, the specific signal providers and fingerprint attributes would need adaptation for each target environment. As with any single-site empirical study, the quantitative results reflect one organization's codebase and deployment cadence, and external validity requires replication across enterprises with different application architectures and team structures before the observed improvement magnitudes can be generalized.

Future research could replace the hand-crafted decision tree with a supervised classifier trained on labeled failure events, potentially improving classification accuracy beyond the 92.1% observed in this study. Expanding the fingerprint model to incorporate visual attributes captured through screenshot comparison would address edge cases where structural similarity is high but rendered appearance differs. Cross-platform validation on React or Angular applications with Shadow DOM encapsulation would test whether the three-strategy architecture generalizes beyond the Salesforce ecosystem or whether additional healing strategies are needed for different component lifecycle models.

Conclusions

The three-stage self-healing pipeline operated across six consecutive production release cycles without requiring structural modification, even as the underlying CRM platform absorbed two major Salesforce releases during the observation period. Classification accuracy exceeded 92%, and the bounded design preserved the test suite's ability to catch genuine application defects by refusing to heal failures that did not match a known non-deterministic signature.

Flakiness declined from a persistent baseline above 20% to a sustained level below 3% in the final intervention cycles, and the corresponding reduction in manual maintenance freed the equivalent of three to four full-time analyst positions from triage duties. These gains proved durable rather than one-time: each successive cycle showed equal or lower flakiness than the preceding one, with no regression observed across the full measurement window.

Strategy-level analysis revealed that timing and rendering failures, not locator drift, drove most of the baseline instability. Locator-only healing tools would have left this dominant portion unaddressed. Fingerprinting resolved selector changes with zero false positives, while the adaptive retry engine owed its effectiveness to monitoring LWC-specific lifecycle signals that standard DOM readiness checks do not capture. Taken together, these results confirm that categorizing failures before acting on them produces better stabilization outcomes than applying a uniform repair strategy across all failure types.

References

1. Luo, Q., Hariri, F., Eloussi, L., & Marinov, D. (2014). An empirical analysis of flaky tests. In Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of

Software Engineering (FSE) (pp. 643–653). ACM. https://doi.org/10.1145/2635868.2635920
2. Parry, O., Kapfhammer, G. M., Hilton, M., & McMinn, P. (2022). A survey of flaky tests. ACM Transactions on Software Engineering and Methodology, 31(1). https://doi.org/10.1145/3476105
3. Lam, W., Muşlu, K., Sajnani, H., & Thummalapenta, S. (2020). A study on the lifecycle of flaky tests. In Proceedings of the 42nd ACM/IEEE International Conference on Software Engineering (ICSE) (pp. 1471–1482). https://doi.org/10.1145/3377811.3381749
4. Alshammari, A., Morris, C., Hilton, M., & Bell, J. (2021). FlakeFlagger: Predicting flakiness without rerunning tests. In Proceedings of the 43rd IEEE/ACM International Conference on Software Engineering (ICSE) (pp. 1572–1584). https://doi.org/10.1109/ICSE43902.2021.00140
5. Fatima, S., Ghaleb, T. A., & Briand, L. (2023). Flakify: A black-box, language model-based predictor for flaky tests. IEEE Transactions on Software Engineering, 49(4). https://doi.org/10.1109/TSE.2022.3201209
6. Parry, O., Kapfhammer, G. M., Hilton, M., & McMinn, P. (2023). Empirically evaluating flaky test detection techniques combining test case rerunning and machine learning models. Empirical Software Engineering, 28(3). https://doi.org/10.1007/s10664-023-10307-w
7. Gruber, M., Lukasczyk, S., Kroiß, F., & Fraser, G. (2021). An empirical study of flaky tests in Python. In Proceedings of the 14th IEEE Conference on Software Testing, Verification and Validation (ICST) (pp. 148–158).
8. QA Wolf. (2026). The 6 types of AI self-healing in test automation. https://www.qawolf.com/blog/self-healing-test-automation-types
9. Healenium. (2024). Self-healing test automation framework. https://healenium.io/
10. Salesforce Developers. (2024). Testing Lightning Web Components. Salesforce. https://developer.salesforce.com/docs/platform/lwc/guide/testing.html
11. Gartner. (2024). Market guide for AI-augmented software-testing tools. Gartner Research.
12. Silva, D., Teixeira, L., & d'Amorim, M. (2024). The effects of computational resources on flaky tests. IEEE Transactions on Software Engineering. https://doi.org/10.1109/TSE.2024.3462251
13. Bell, J., Legunsen, O., Hilton, M., Eloussi, L., Yung, T., & Marinov, D. (2018). DeFlaker: Automatically detecting flaky tests. In Proceedings of the 40th IEEE/ACM International Conference on Software Engineering (ICSE) (pp. 433–444). https://doi.org/10.1145/3180155.3180164
14. Parry, O., Kapfhammer, G. M., Hilton, M., & McMinn, P. (2024). Proceedings of the 1st International Workshop on Flaky Tests (FTW 2024). In ICSE 2024 Workshops. https://doi.org/10.1145/3643656
15. Rahman, S., Baz, A., Misailovic, S., & Shi, A. (2024). Flakiness repair in the era of large language models. In Proceedings of ICSE 2024 SRC (pp. 93–104). IEEE/ACM