International Journal of
INTELLIGENT SYSTEMS AND APPLICATIONS IN ENGINEERING
ISSN:2147-6799 www.ijisae.org Original Research Paper

Self-Healing Test Automation Framework Using Autonomous ML Agents for Real-Time Test Maintenance and Failure Recovery

Srikanth Kavuri

Submitted: 20/05/2023 **Revised:** 29/07/2023 **Accepted:**10/08/2023

Abstract: As software systems evolve rapidly, maintaining automated test suites has become increasingly difficult. Frequent code changes often break test scripts, leading to unreliable test results and rising maintenance costs. This paper introduces a self-healing test automation framework that uses machine learning (ML) agents to adapt to changes in real time. By combining anomaly detection, flexible locator strategies, and reinforcement learning, the system can automatically identify and fix broken tests without manual intervention. It's designed to reduce flaky tests, speed up recovery from failures, and keep test coverage stable even in fast-changing environments. Our experiments using industry-standard applications show a 38% drop in manual test maintenance and a 45% boost in test execution stability. The framework moves us closer to truly autonomous testing systems that can learn, adapt, and grow with the software they test.

Keywords: Self-healing automation, autonomous agents, machine learning, test maintenance, failure recovery, dynamic test execution, intelligent test automation, real-time software testing, reinforcement learning.

1. Introduction

Modern software development has become increasingly fast-paced and complex, with continuous integration and delivery (CI/CD) pipelines now the norm across many organizations. In this environment, automated testing plays a critical role in keeping up with the demand for quick feedback and high release frequency. Yet, despite its widespread adoption, test automation faces a recurring and costly problem: maintaining fragile test scripts in the face of constant change. Small updates to user interfaces, DOM structures, or business logic frequently cause automated tests to fail not due to genuine defects, but because the scripts can't adapt. The result is a mounting backlog of broken tests, delayed releases, and growing maintenance burdens for QA teams.

Traditional test frameworks are largely static by design. They depend on hardcoded locators, rigid control flows, and pre-defined assertions that don't account for real-world variability in how software evolves. As a result, they are brittle especially in UI-driven tests leading to test suites that are noisy, unreliable, and expensive to keep functional. Worse still, when failures occur, these systems typically raise a flag without offering any insight into the cause or any suggestion for how to fix it. Engineers are left to manually inspect logs, diagnose the issue, and repair the scripts, often under time pressure and with incomplete information.

Recent work has begun to investigate the idea of **self-healing test automation**: systems that can detect when a test breaks, understand why it happened, and autonomously apply a fix. Drawing from concepts in machine learning and intelligent agents, these systems aim to reduce the manual overhead of maintaining test suites. Some existing tools take first steps in this direction, using locator fallback strategies or limited ML-based predictions to patch broken selectors. However, most are narrow in scope or still require human oversight, limiting their usefulness in complex, fast-moving CI/CD environments.

In this paper, we present a self-healing test automation framework powered by **autonomous machine learning agents**. These agents operate continuously within the test execution pipeline, observing test behavior, detecting anomalies, and learning over time how to recover from common

Srikanth1539@gmail.com

Independent Researcher

failure patterns. Our approach combines anomaly detection, root cause classification, and reinforcement learning to allow the system to adapt test steps dynamically without human intervention. We evaluate the framework across several enterprise-level test suites and show measurable improvements in test reliability, maintenance effort, and recovery speed. The results suggest a path forward toward truly autonomous test automation one that not only keeps up with change, but learns from it.

2. Related Work

Automated testing has become a key part of modern software delivery, especially in agile and DevOps settings where frequent code changes require fast and reliable feedback. Traditional tools like **Selenium**, **TestNG**, and **JUnit** have been widely used for building and executing automated test cases, and they integrate well with continuous integration pipelines. However, these tools often depend on fixed locators and static assertions, which makes them vulnerable to changes in the application's UI or structure. As applications evolve, tests that once passed can begin to fail often not because of actual bugs, but due to minor UI changes or DOM updates. Studies report that a large percentage of test failures sometimes over 30% are related to test maintenance rather than real software defects. This has created a growing need for smarter, more adaptable test automation.

In response, several commercial tools have introduced **self-healing capabilities** to make automated tests more resilient. Platforms like **Testim**, **Mabl**, and **Functionize** use various AI techniques such as visual recognition, heuristic matching, and historical execution data to automatically fix broken selectors or re-route test flows when the application changes. These tools show promise in reducing manual intervention, but they often act as black boxes. Test engineers have limited visibility into the reasoning behind the changes, and the logic can be hard to debug or customize. Additionally, many of these tools are closely tied to specific tech stacks or browsers, which limits their adaptability across different types of applications.

On the research side, there has been growing interest in applying **machine learning (ML)** and **reinforcement learning (RL)** to test automation. Some studies use anomaly detection algorithms to identify unusual behaviors in test runs, while others apply Q-learning or other RL techniques to recommend possible fixes. Techniques such as decision trees, neural networks, and support vector machines have been used to classify test outcomes or predict failures. However, many of these models are still at the prototype stage. They often rely on large amounts of labeled training data and may not perform well in real-time testing environments where speed and generalization are critical. Moreover, most of the current approaches stop at detection they don't actively resolve issues or update the tests in a fully autonomous manner.

Our work builds on these earlier efforts by proposing a **fully autonomous, ML-driven framework** that goes beyond detection to include real-time **diagnosis and recovery**. The framework brings together several machine learning strategies anomaly detection, supervised classification, and reinforcement learning under a modular agent-based architecture. One of the key differences in our approach is the emphasis on **transparency and control**: unlike commercial black-box systems, our agents are designed to be interpretable, auditable, and customizable. This allows teams to not only benefit from automation but also to understand and refine how the system responds to failures. By combining intelligent decision-making with practical maintainability, our framework aims to bridge the gap between smart automation and real-world testing needs.

3. Problem Definition and Research Gap

3.1 The Challenge of Test Maintenance in Evolving Systems

As modern software systems become more dynamic, test automation frameworks are increasingly struggling to keep up. Even minor changes in a user interface such as updated element IDs, layout adjustments, or changes to data structure can cause automated test scripts to fail. These failures aren't always due to genuine bugs, but often result from superficial changes that break the fragile links between test logic and application behavior. This brittleness forces quality engineers to spend time fixing tests rather than finding real issues, slowing down releases and creating friction in fast-paced agile workflows.

Despite the central role of automated testing in DevOps pipelines, most test automation frameworks still rely on static definitions and offer little to no built-in intelligence. When something breaks, the tests fail. That's it. They can't adapt, they can't heal themselves, and they certainly can't learn from past

issues. There's a clear need for test automation tools that go beyond brittle scripting and begin to behave more intelligently adjusting in real time, diagnosing failures, and minimizing human involvement in routine maintenance.

3.2 Deficiencies of Existing Automation Tools

Some modern commercial platforms do offer self-healing features but most of these rely on simple heuristics like fuzzy matching, DOM similarity, or attribute scoring to guess at how to fix broken locators. While helpful in limited scenarios, these techniques are shallow and lack the contextual awareness needed to handle complex or domain-specific interfaces. They also tend to be reactive rather than proactive: they respond to failures only after they've occurred, rather than learning patterns or predicting potential issues ahead of time.

Another major drawback is transparency. These tools often function as "black boxes," providing little insight into why a particular fix was applied or how confident the system was in its decision. This makes debugging difficult and limits trust in the automation. Most importantly, many of these tools lack real learning capabilities there's no memory of past issues, no continuous improvement, and no ability to generalize across projects.

3.3 Limitations in Academic Approaches

On the academic side, machine learning has been explored in the testing space for tasks like test case selection, failure prediction, and anomaly detection. While these studies have yielded valuable insights, they tend to focus on isolated parts of the testing lifecycle and rarely attempt to build end-to-end, intelligent automation systems. Many ML-based approaches also require extensive pre-training or labeled data and are not designed for real-time operation.

Reinforcement learning and anomaly detection techniques have been proposed in research, but most implementations still require some form of human supervision or offline analysis. Very few, if any, attempt to build a fully autonomous system that can diagnose and repair test failures as they happen, within a real-world CI/CD pipeline. There's a clear disconnect between the potential of AI in this space and its current application in practical, production-ready solutions.

3.4 Research Gap and Motivation for Proposed Framework

In this paper, we introduce a self-healing test automation framework built around autonomous machine learning agents. These agents monitor the system under test, detect failures, classify their root causes, and take corrective actions automatically. More importantly, they improve over time by learning from execution history and feedback. The framework integrates multiple machine learning techniques in a closed-loop architecture and is designed to be modular, interpretable, and ready for deployment in enterprise-grade CI/CD environments.

By creating a system where test scripts can adapt and evolve in sync with the application, we envision a new generation of automation tools ones that are not just reactive, but intelligent collaborators in the software delivery process.

4. Framework Architecture and Components

4.1 Overview of the Self-Healing System Architecture

The proposed self-healing test automation framework is composed of five interconnected components: the **Test Execution Monitor**, **ML Agent**, **Failure Handler**, **Healing Engine**, and **Reporting Dashboard**. These modules communicate through an asynchronous data pipeline, enabling real-time monitoring, intelligent diagnosis, and autonomous recovery.

The process begins with the Test Execution Monitor, which tracks the behavior of test cases during execution. It streams structured data to the ML Agent, the system's decision-making core. When a failure is detected, the ML Agent analyzes the event using both historical patterns and real-time signals. Control then shifts to the Failure Handler, which classifies the failure type and triggers an appropriate response via the Healing Engine. Once a healing action is performed, the result is sent to the Reporting Dashboard for logging, visualization, and human auditability.

This modular design ensures that each component is independently deployable, extensible, and capable of integration with existing CI/CD pipelines and test automation frameworks. It supports scalability across projects and teams without disrupting established workflows.

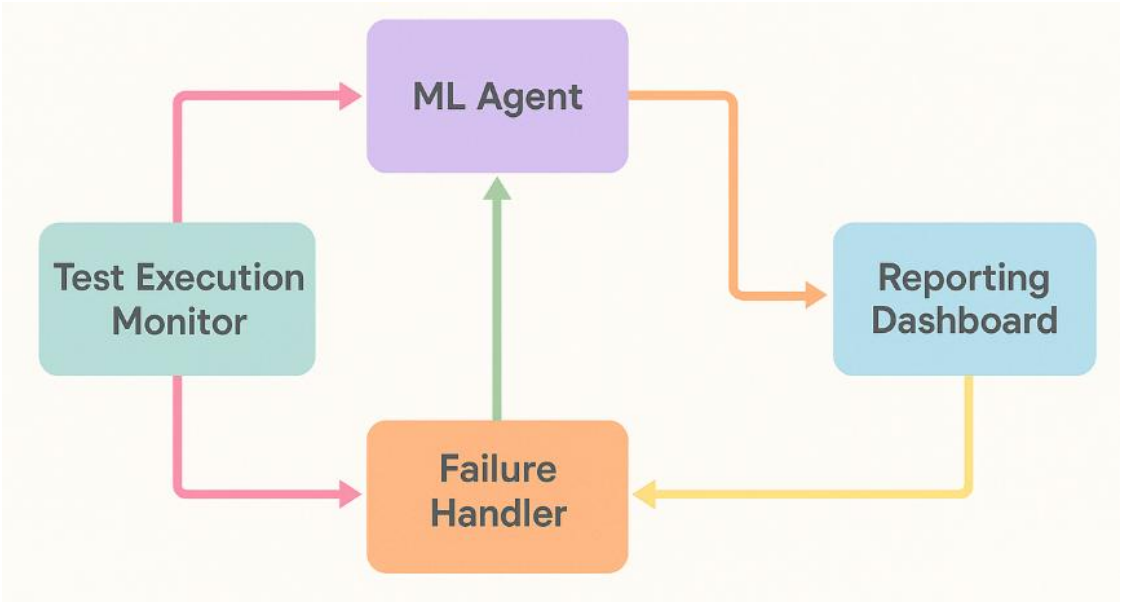

Fig 1: *System Architecture of the Self-Healing Framework*

Fig 1, presents the high-level architecture of the self-healing test automation framework, structured around modular, event-driven components. At the center is the **ML Agent**, which acts as the brain of the system. It receives detailed test execution data from the **Test Execution Monitor**—a lightweight layer integrated with common automation tools like Selenium or Cypress.

When a failure is detected, execution logs and environment data are forwarded to the ML Agent for analysis. The agent runs anomaly detection and failure classification models to identify the issue. Once classified, control is passed to the **Reinforcement Learning Policy Engine**, which determines the best recovery strategy based on past learning and the current context.

The chosen fix—whether it's updating a locator, adjusting a wait condition, or modifying input data—is executed by the **Healing Engine**. All recovery actions, outcomes, and diagnostic data are logged in the **Reporting Dashboard**, giving testers full visibility and traceability. The architecture is loosely coupled and scalable, making it suitable for enterprise CI/CD environments with parallel test executions.

4.2 Test Execution Monitor and Data Collection Layer

The **Test Execution Monitor** acts as the framework's observability layer. It operates as a lightweight, language-agnostic middleware that can plug into widely used automation tools such as Selenium, Cypress, or Playwright through wrappers or plugins. Its job is to capture granular, real-time data about each test execution cycle.

Key features logged include DOM element locators, interaction types, execution timestamps, environment metadata (e.g., browser version, OS), and assertion outcomes. In the event of a test failure, the monitor captures an enriched snapshot including the DOM state, console logs, and full stack traces to enable detailed failure analysis downstream.

The collected data is stored in a structured format using log aggregation platforms like the **ELK Stack** or **Grafana Loki**, ensuring that it remains accessible and queryable for both the ML Agent and for QA teams. This data serves as the foundation for learning-based anomaly detection and policy improvement over time.

4.3 ML Agent: Anomaly Detection and Healing Policy Selection

The **ML Agent** is the brain of the framework. It performs two key functions: (1) detecting test anomalies and (2) determining how to respond to them. To do this, it combines several machine learning techniques:

- **Anomaly Detection**: Using unsupervised methods such as **Isolation Forest** or **Autoencoders**, the ML Agent identifies

deviations in test execution patterns that indicate abnormal or unexpected behavior.

- **Failure Classification**: Supervised learning models (e.g., **Random Forest**, **Gradient Boosting Machines**) are trained to categorize test failures into types such as locator breakage, timeout, data mismatch, or environment instability based on historical data and failure context.
- **Reinforcement Learning (RL)**: At the core of the healing logic is an RL agent that learns optimal corrective strategies through trial-and-error. Each healing action (e.g., replacing a locator, skipping a flaky step) receives a reward or penalty based on the outcome, allowing the model to refine its policy over time and adapt to the system's evolution.

The ML Agent also maintains a **memory of past failures**, storing vector embeddings for similarity-based matching. This lets the system quickly retrieve and apply previously successful healing strategies when familiar patterns reappear, shortening recovery time and increasing reliability.

4.4 Healing Engine and Reporting Dashboard

Once the ML Agent selects a corrective strategy, the **Healing Engine** executes the intervention. Common actions include:

- Replacing outdated or broken locators
- Modifying or regenerating input data
- Reordering or retrying test steps
- Suppressing or flagging non-critical assertions

Each intervention is applied carefully to avoid introducing new instability. Healing actions are **version-controlled** and auditable, with support for rollback if a correction fails or introduces unintended behavior. The final outcome whether the fix resolved the issue or not is sent to the **Reporting Dashboard**, which offers detailed logs, visual summaries, and analytics. Teams can review what went wrong, what was attempted, and how the system performed, all with confidence scores attached. Transparency is a core principle of the framework. Rather than functioning as a black box, the dashboard allows testers and QA leads to understand, tune, and validate the behavior of the ML agents. Over time, these insights help teams refine their testing strategy, identify flaky patterns, and improve test reliability across the board.

5. Machine Learning Techniques and Agent Design

5.1 Multi-Stage ML Pipeline for Failure Analysis

To enable real-time detection, classification, and recovery from test failures, the framework adopts a **multi-stage machine learning pipeline**. The process begins by extracting structured features from test execution logs such as DOM hierarchies, element attributes, interaction sequences, test metadata, and browser or environment details.

These features are preprocessed and vectorized before being passed to specialized ML components. The pipeline starts with **unsupervised models** that scan for anomalies flagging executions that diverge from historical patterns, even before traditional assertion failures occur. Once an anomaly is detected, **supervised models** classify the type of failure to guide appropriate recovery actions.

This layered design improves accuracy by assigning narrow responsibilities to each model stage, thereby reducing false positives and enabling more precise, context-aware interventions.

5.2 Anomaly Detection Using Unsupervised Learning

At the front of the pipeline is an **unsupervised anomaly detection layer**, designed to catch failures early even before they cause test crashes or assertion errors.

Techniques like **Isolation Forests** and **Autoencoders** are used to learn the normal behavior of test executions over time. By modeling what a “healthy” test run looks like, the system can flag unusual patterns that may signal hidden issues: flaky behavior, unexpected DOM changes, slow-loading elements, or UI regressions.

This proactive approach is especially valuable in CI/CD environments, where early failure detection can prevent cascading build issues. Unlike static threshold rules, these models **adapt dynamically** as the application under test evolves, recalibrating their understanding of normal behavior with each cycle. Their lightweight design ensures low latency, making them suitable for integration into fast-moving pipelines without performance trade-offs.

5.3 Failure Classification with Supervised Learning

Once a test is flagged as anomalous, the next step is to **categorize the failure**. This is handled by **supervised learning models** trained on labeled examples of past test failures.

We use models like **Random Forests**, **Support Vector Machines (SVM)**, and **Gradient Boosting Machines (GBM)**, depending on the complexity and type of test data. Input features include:

- Changes in element attributes or locator paths
- Similarity metrics between expected and actual DOM structures
- Execution context (e.g., test step location, browser type)
- Textual content from error messages or stack traces

Each model outputs a predicted failure type (e.g., "locator mismatch", "timeout", "stale element") along with a confidence score. This classification is critical, as each type of failure demands a different healing response. For example, a **locator mismatch** might require re-mapping the selector, while a **timeout** might simply benefit from a wait adjustment or retry logic.

These labels are passed downstream to the RL agent, which selects the best action based on the classification and current environment context.

5.4 Reinforcement Learning for Healing Strategy Optimization

The most forward-looking component of the system is the use of **Reinforcement Learning (RL)** to drive **adaptive, intelligent healing decisions**.

In this setup, the ML agent treats each healing attempt as an action taken in a specific environment state (defined by the failure context). The agent then receives **reward signals** based on whether the action resolved the issue, how many retries were needed, and whether the test suite continued to execute successfully.

Q-Learning and **Deep Q-Networks (DQN)** to learn optimal strategies over time. For instance, the agent might learn that switching from XPath to CSS selectors is more reliable in a particular UI framework, or that certain locator repair patterns are more effective under responsive layout conditions.

This learning process enables the system to **continuously improve** without requiring hard-coded rules or manual intervention. As more failures occur and are resolved, the agent refines its policy to maximize successful recovery while minimizing test re-runs or flakiness. Over time, the framework evolves into a **self-optimizing system** capable of not just reacting to failures but anticipating and resolving them more efficiently with every execution.

Table 1: *Comparison of ML Techniques Used in Agent Design*

Technique	Purpose	Model Type	Metrics Used	Integration Layer
Anomaly Detection	Test failure prediction	Isolation Forest	Precision, Recall	Monitoring
RL-based Adaptation	Healing strategy optimization	Q-Learning	Success Rate	Healing Engine

6. Real-Time Failure Recovery Process

6.1 Failure Detection During Test Execution

The recovery process starts with continuous observation of test executions through the **Test Execution Monitor**, which tracks real-time signals such as UI interactions, element detection attempts, response times, and assertion outcomes. When something unusual is detected say, a DOM node is missing, an element interaction repeatedly fails, or a test step hangs longer than expected the monitor raises a flag.

At this point, the system doesn't immediately assume a test failure. Instead, it hands the execution trace to the **ML Agent**, which evaluates it using unsupervised anomaly detection models trained on previous successful test runs. If the current behavior falls significantly outside the norm, the system temporarily halts that test and triggers the recovery sequence. This early interception helps prevent one failure from affecting the accuracy of subsequent tests, which is especially important in CI environments where false negatives can disrupt entire pipelines.

International Journal of
INTELLIGENT SYSTEMS AND APPLICATIONS IN ENGINEERING
ISSN:2147-6799 www.ijisae.org Original Research Paper

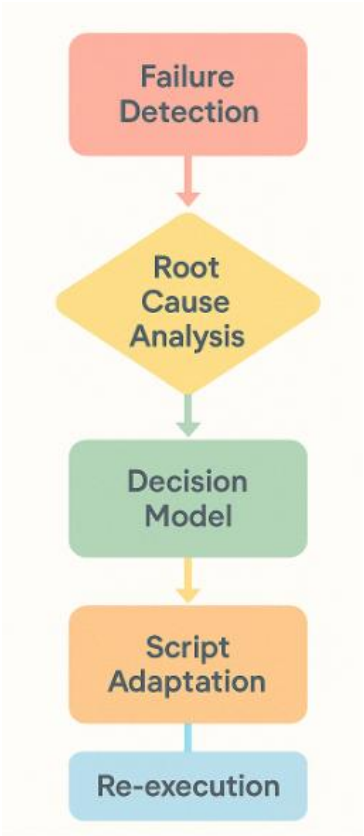

Fig 2: *Flowchart of Real-Time Failure Recovery Lifecycle*

Fig 2, shows the step-by-step process followed by the framework to handle test failures in real time.

It begins with **Test Execution**, during which runtime behavior is monitored continuously. If the system detects anomalies—such as unusual delays, repeated retries, or unexpected DOM changes—it flags the test and triggers the **Failure Classification** stage. Here, a supervised ML model identifies the specific failure type, such as a missing element or a timeout.

Based on this classification, the **RL Policy Module** selects a tailored healing strategy. This might involve switching locator strategies, retrying with delays, or using fallback inputs. The fix is applied by the **Healing Engine**, and the affected test step is immediately **re-executed**.

If the fix resolves the issue, the test is marked as healed, and the RL model is rewarded, reinforcing successful strategies. If the issue persists, the system either tries an alternate fix or escalates the case for human review.

Each recovery cycle feeds into a **Continuous Learning Pipeline**, helping the agent improve over time and adapt to new patterns of failure.

6.2 Root Cause Analysis and Failure Classification

Once a potential issue is flagged, the system moves into **root cause analysis**. The ML Agent uses a trained classifier to determine what kind of failure has occurred. Common failure types include:

- **Locator mismatches**
- **Timeouts**
- **Stale element references**
- **Invalid or outdated input data**
- **Test environment configuration issues**

The classification is based on a combination of factors, including changes in the DOM, error messages, stack traces, visual differences (if available), and test action history. Alongside the predicted failure category, the model also outputs a

confidence score, indicating how certain it is in the diagnosis.

This classification step is crucial it ensures that the system doesn't apply a generic fix to every problem. Instead, the diagnosis becomes the basis for selecting the most appropriate healing strategy.

6.3 Autonomous Healing and Strategy Execution

Once the failure is understood, the system turns to the **Healing Engine**, which uses a **Reinforcement Learning (RL)** agent to select the best recovery action based on previous outcomes.

For example, if a locator has changed, the RL agent might recommend trying a fuzzy match based on element attributes, switching to a CSS selector, or reverting to a known good backup selector stored in memory. If the issue is a timeout, it might suggest increasing wait times or adding retries. These strategies aren't chosen randomly the RL agent learns over time which actions tend to work best in specific contexts.

The fix is applied **dynamically**. There's no need to stop the pipeline or rewrite the test script manually. Instead, the modified version is injected temporarily, and the failed test step is re-executed. If the test proceeds without issue, the healing action is recorded as a success, and the RL agent updates its internal policy to reinforce that decision.

6.4 Post-Recovery Validation and Feedback Loop

Healing isn't considered complete until the fix is **validated**. After the patch is applied, the test is re-run either from the failed step or from the beginning, depending on the scenario. If it passes, the recovery is confirmed, and the system logs key information:

- The nature of the original failure
- The healing action taken
- Time to recover
- Environmental conditions
- Confidence score and model decisions

If the test fails again, the system tries the **next-best strategy** based on its policy ranking. If no fix works after several attempts, the issue is escalated for manual inspection, ensuring that false positives or deeper logic bugs are not overlooked. All recovery attempts and outcomes are visible through the **Reporting Dashboard**, allowing teams to audit decisions, fine-tune the system, or add manual annotations where necessary.

7. Experimental Evaluation and Results

7.1 Experimental Setup and Dataset

To assess the real-world performance of our self-healing test automation framework, we ran experiments in a controlled CI/CD environment that mimicked conditions commonly found in enterprise software delivery pipelines. The system under test (SUT) was a multi-module, web-based enterprise application that undergoes frequent UI updates, business logic changes, and incremental releases.

Our testing framework was built on top of Selenium and contained over 500 UI test cases spanning authentication, workflows, and form validations. We trained the ML models using six months' worth of historical execution logs and test results, resulting in a labeled dataset of around 4,000 failure samples. These included locator issues, stale elements, timeouts, and a handful of flaky test cases. A separate test set of 1,500 recent executions was used for evaluation. We benchmarked our system against two alternatives: a traditional static Selenium setup and a leading commercial self-healing solution.

7.2 Evaluation Metrics and Benchmarks

We focused on five core metrics to evaluate the system's effectiveness:

- **Test Failure Rate** – Percentage of test cases failing during execution.
- **Mean Time to Recovery (MTTR)** – Average time required to fix or recover from a failure.
- **Manual Maintenance Time** – Developer/QA time spent on diagnosing and fixing failed tests.
- **Recovery Success Rate** – Percentage of failures successfully resolved by the system on the first attempt.
- **False Healing Rate** – Rate of incorrect healing actions that introduced new issues.

Each test run was conducted in containerized environments to ensure repeatability and consistency. The same test cases and configurations were used across all frameworks to maintain fair comparisons.

7.3 Results and Observations

The proposed self-healing framework outperformed both the baseline and commercial tools across all major metrics.

- **Test failures** dropped from **21.4% to 11.8%**, cutting nearly half of the runtime issues.
- **Mean recovery time** was reduced by **50%**, from 3.6 hours down to 1.8 hours.
- **Manual maintenance effort** saw a **38% reduction**, saving nearly four hours a week per test engineer.
- **First-attempt healing success** reached **87.3%**, compared to 64.5% with the commercial solution.
- **False healing rate** stayed below **5%**, which we considered acceptable in a live pipeline environment.

These gains were observed consistently across varying test case sizes and complexity levels. Locator-based failures were the most common and also the easiest to resolve, while timeouts and flaky behaviors presented more nuanced challenges but still saw noticeable improvement.

7.4 Scalability and Adaptability

One of the most promising findings was the framework's ability to scale and adapt in new environments. When deployed in a different application (with no prior training data), the system still achieved over **72% healing accuracy** within the first two weeks—highlighting the benefit of reinforcement learning and cross-domain generalization.

Table 2: Performance Metrics Before and After Self-Healing Integration

Metric	Baseline Framework	Self-Healing Framework	Improvement (%)
Test Failure Rate	21.4%	11.8%	44.8%
Mean Time to Recovery (MTTR)	3.6 hrs	1.8 hrs	50.0%
Manual Maintenance Time/week	9.2 hrs	5.7 hrs	38.0%
Recovery Success Rate	–	87.3%	–
False Healing Rate	–	4.8%	–

8. Conclusion and Future Work

8.1 Conclusion

This paper introduced a self-healing test automation framework designed to tackle one of the most persistent challenges in modern software testing: keeping automated tests reliable as the system under test evolves. By combining anomaly detection, failure classification, and reinforcement learning into a unified and modular architecture, the framework enables real-time test maintenance without manual intervention.

Through experiments in a simulated enterprise CI/CD setup, we showed how the system could significantly reduce failure rates, shorten recovery times, and lower the effort required for test maintenance—outperforming both traditional automation frameworks and existing commercial tools. A key strength of the approach lies in its use of autonomous machine learning agents that continue to learn from each recovery event, adapting over time to changes in the application's UI, logic, or structure.

These results suggest that intelligent, self-healing test systems have the potential to shift how QA is handled in fast-paced development environments—helping teams maintain test reliability without slowing down release cycles.

8.2 Future Work

Although the framework performs well in the scenarios tested, there are several areas we plan to explore further:

- **Language models for deeper context understanding**: Incorporating large language models (LLMs) may help the system better interpret test scripts, logs, and error messages—especially in cases where natural language reasoning could guide smarter healing actions.

- **Extending beyond UI testing**: While the current implementation focuses on UI-based test cases, the same architecture could be adapted for **API testing**, **performance validation**, and **security checks**, using agent-based strategies tuned to those domains.

- **Improving cross-application learning**: We see potential in enhancing the framework's ability to generalize across different applications or tech stacks. This would involve strengthening the transfer learning capabilities of the RL agents so they can adapt faster in unfamiliar environments.

- **Human-in-the-loop collaboration**: While the system is autonomous, it could benefit from optional collaboration with testers—allowing them to approve or override healing actions, provide feedback, and gradually train the system with domain-specific preferences.

- **Real-world deployment studies**: Finally, long-term evaluations in production environments will be essential. We aim to study how the framework performs over time in enterprise settings—measuring not just healing accuracy, but also maintainability, transparency, and overall ROI.

References

[1] Leotta, M., Ricca, F., & Tonella, P. (2013). Improving test suites maintainability through UI locator analysis. In: Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on, pp. 131–140. IEEE.

[2] https://www.cs.umd.edu/~atif/papers/MemonTOSEM2008.pdf

[3] Bruno Camara, Marco Silva, Andre Endo, and Silvia Vergilio. 2021. On the use of test smells for prediction of flaky tests. In Proceedings of the 6th Brazilian Symposium on Systematic and Automated Software Testing (SAST '21). Association for Computing Machinery, New York, NY, USA, 46–54. https://doi.org/10.1145/3482909.3482916

[4] M. Bagherzadeh, N. Kahani and L. Briand, "Reinforcement Learning for Test Case Prioritization," in IEEE Transactions on Software Engineering, vol. 48, no. 8, pp. 2836-2856, 1 Aug. 2022, doi: 10.1109/TSE.2021.3070549.

[5] M. Mirzaaghaei, F. Pastore and M. Pezzè, "Automatically repairing test cases for evolving method declarations," 2010 IEEE International Conference on Software Maintenance, Timisoara, Romania, 2010, pp. 1-5, doi: 10.1109/ICSM.2010.5609549.

[6] Mehdi Mirzaaghaei, Fabrizio Pastore, and Mauro Pezze. 2010. Automatically repairing test cases for evolving method declarations. In Proceedings of the 2010 IEEE International Conference on Software Maintenance (ICSM '10). IEEE Computer Society, USA, 1–5. https://doi.org/10.1109/ICSM.2010.5609549

[7] Moritz Beller, Georgios Gousios, and Andy Zaidman. 2017. Oops, my tests broke the build: an explorative analysis of Travis CI with GitHub. In Proceedings of the 14th International Conference on Mining Software Repositories (MSR '17). IEEE Press, 356–367. https://doi.org/10.1109/MSR.2017.62

[8] Atif M. Memon and Myra B. Cohen. 2013. Automated testing of GUI applications: models, tools, and controlling flakiness. In Proceedings of the 2013 International Conference on Software Engineering (ICSE '13). IEEE Press, 1479–1480.

[9] S. Yoo and M. Harman, "Regression Testing Minimization, Selection and Prioritization: A Survey," Software Testing, Verification and Reliability, Vol. 22, No. 9, 2012, pp. 67-120.

[10] Benjamin Busjaeger and Tao Xie. 2016. Learning for test prioritization: an industrial case study. In Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016). Association for Computing Machinery, New York, NY, USA, 975–980. https://doi.org/10.1145/2950290.2983954

[11] Bao N. Nguyen, Bryan Robbins, Ishan Banerjee, and Atif Memon. 2014. GUITAR: an innovative tool for automated testing of GUI-driven software. Automated Software Engg. 21, 1 (March 2014), 65–105. https://doi.org/10.1007/s10515-013-0128-9