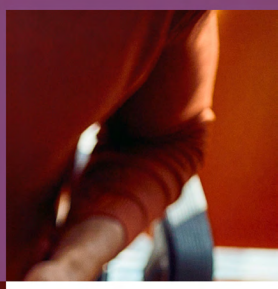
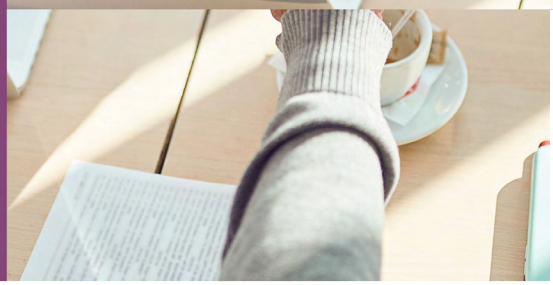





Self-Healing Test Scripts

AI-Augmented Quality Assurance with Smarter Element Change Detection

Executive Summary

Frequent user interface (UI) changes in modern applications often break automated test scripts, leading to delays, increased maintenance effort, and higher costs. Traditional approaches require manual updates and reruns, which can take hours for even minor changes and significantly slow down release cycles. This white paper introduces self-healing test scripts, a solution that combines heuristic logic and identification driven by artificial intelligence (AI) to automatically adapt to UI changes. The approach uses a layered strategy: first attempting the original locator, then applying heuristics, and finally leveraging an AI model when needed. A detailed report of healed elements simplifies maintenance and reduces turnaround time.

CADMUS LOGIC.AI℠

By eliminating the need for manual locator updates and reruns, this solution can save several hours per test cycle and reduce the risk of undetected bugs. It improves Continuous Integration/Continuous Deployment (CI/CD) stability, accelerates delivery timelines, and minimizes human intervention, resulting in both time and cost efficiencies. Future enhancements will include support for multiple AI models, improved reporting, and automated script updates. Self-healing powered by Generative AI (GenAI) offers a practical path to resilient, efficient test automation that delivers measurable productivity gains.

Introduction

Modern software applications are increasingly focused on delivering rich and dynamic UI and seamless user experiences (UX). While this evolution enhances usability, it introduces significant challenges for automated testing. Test automation frameworks rely heavily on stable UI element identifiers to validate functionality. However, frequent changes in application code—especially updates to UI elements—often break previously working test scripts. This results in additional effort to maintain scripts and delays in the testing cycle.

Current Challenges

Testing modern UI/UX applications introduces several pain points that impact the reliability and efficiency of automated scripts. These challenges often lead to increased maintenance effort and delayed feedback in the development cycle:

- **Fragile Test Scripts:** Automated scripts are tightly coupled with UI element identifiers. Any modification to these identifiers causes test failures.
- **Delayed Bug Detection:** When scripts fail due to identifier changes, subsequent application updates may introduce bugs that remain undetected until scripts are repaired.
- **Manual Maintenance Overhead:** Updating broken scripts is time-consuming and requires detailed analysis of failure reports.

> “Instead of halting execution when a locator becomes invalid, the script intelligently searches for alternatives.

As-Is Process

The current approach to handling UI changes in test automation involves several manual steps:

1. **Script Development:** Test scripts are created using the UI element identifiers present in the application at the time of development.
2. **Failure Analysis:** When identifiers change, scenarios using those identifiers fail. Engineers review test automation reports to identify the root cause.
3. **Script Update:** The relevant identifiers in the script are updated to match the new application state.
4. **Validation:** Updated scripts are rerun to confirm successful execution.
5. **Version Control:** Changes are committed and pushed to the repository (e.g., Git).

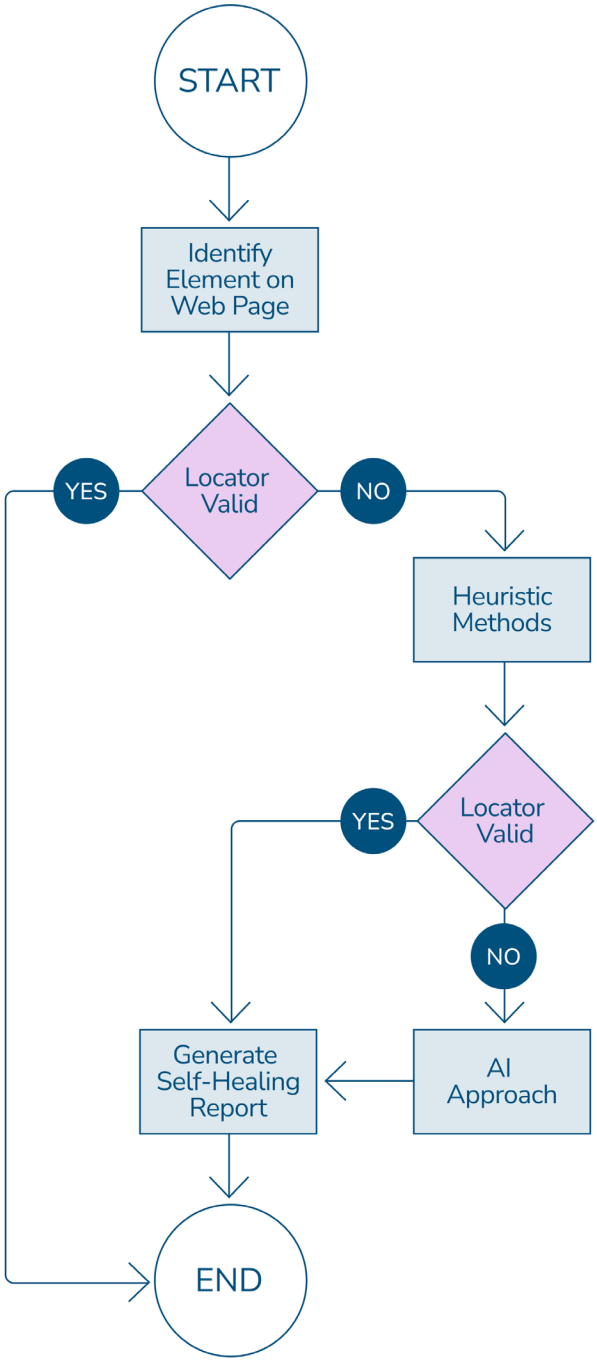
Time and Effort

On average, analyzing failures and updating scripts manually takes one to two hours per scenario. In cases where multiple elements are affected, the effort can increase significantly, impacting overall testing timelines and delivery schedules.

Solution Approach

Understanding Self-Healing Scripts

In traditional test automation, scripts fail when UI element locators change, leading to time-consuming manual fixes.



Self-Healing Test Script Workflow

Self-healing scripts address this challenge by introducing resilience into the automation process. Instead of halting execution when a locator becomes invalid, the script intelligently searches for alternatives—first through heuristic methods and then, if necessary, using AI-driven techniques. This layered approach ensures tests continue running smoothly, even in dynamic UI environments.

How the Solution Works

The implementation follows a structured yet adaptive process.

The journey begins with creating a self-healing function—a reusable component designed to identify elements on a web page. When a test runs, this function first attempts to locate the element using the original locator. If successful, the script proceeds without interruption.

However, when the locator fails, the script doesn't stop. Instead, it handles the *NoSuchElementException* gracefully and invokes a heuristic method. This method uses alternate attributes or patterns derived from the original locator to find the element. The alternate locators are ordered according to priority in which the element needs to be identified based on the given locator value, so the most probable locator will be used first to identify the element and only if that fails, it moves to the next most probable locator to identify the element. If the heuristic approach succeeds, the test continues seamlessly.

If both the original locator and heuristic method fail, the solution escalates to the AI-based approach. Leveraging the Mistral model, the script predicts and identifies the correct element based on historical patterns and contextual clues. This AI step is intentionally reserved as a last resort because it is more computationally intensive than heuristic checks.

Finally, the process concludes with **report generation**. The report captures:

- Elements that failed with the original locator
- Elements successfully identified through heuristic or AI methods
- Old and new identifiers used during healing

This transparency simplifies future maintenance and provides valuable insights into UI changes.

Why This Approach Matters

The benefits of this solution extend beyond reducing failures:

- Scripts automatically adapt to minor or moderate UI changes, minimizing manual intervention.
- Engineers receive detailed reports, making updates straightforward.
- Tests pass with updated locators during execution, eliminating the need for reruns.
- XPath or attribute changes no longer derail automation, thanks to self-healing intelligence.

Tools and Technologies Behind the Solution

The framework was built using **Python** and **Behave**, developed in **PyCharm Community Edition**. For browser interaction, **Selenium** with **ChromeDriver** was employed. The AI capability relies on **Mistral** (mistral.mistral-large-2407-v1:0), which powers the intelligent element identification process.

How to Implement Self-Healing

For New Applications

Implementing self-healing in a new application starts with designing a reusable function that becomes the backbone of element identification. This function should be modular and structured for flexibility:

- **Create a reusable function** to locate UI elements dynamically.
- **Break down the logic** into two supporting functions:
  - One for heuristic-based identification.
  - Another for AI-driven identification.

The main function should:

1. Attempt to find the element using the default locator.
2. If that fails, call the heuristic function and return the element if found.
3. If heuristics fail, escalate to the AI-based function and return the element if found.
4. If all attempts fail, raise a NoSuchElementException.

This layered approach ensures that the script exhausts all options before failing, making it robust against UI changes.

For Existing Applications

Enhancing existing scripts to become self-healing requires minimal but strategic changes:

- Introduce the same reusable function described above.
- Modify existing scripts so that every element lookup calls this new function.
- In Java-based frameworks, the @FindBy annotation can be overridden to integrate this logic seamlessly, ensuring backward compatibility without rewriting entire scripts.

How is the AI/Heuristics Approach Validated?

The element locator identified by the heuristic or AI model is used to substitute the existing failed identifier used in the script to complete the

> “ Script maintenance issues can be reduced drastically if the development team works hand in hand with the test automation team, which can ensure minimum failures in the automation scripts.

test scenario run. If the scenario has passed the validation using the identifier provided by the heuristic approach or the AI model, then it has proved to be a successful identifier to be used. Also, a report is created on the new self-healed identifiers used, so the user can verify again before updating the script with the new identifier.

Lessons Learned

Setting up the framework was one of the biggest hurdles, as it required integrating multiple components and ensuring they worked seamlessly together. Another challenge was choosing the right self-healing strategy from several available options. Each approach had trade-offs, so finding a balance between performance and adaptability was critical.

Self-healing is not a one-size-fits-all solution. There are numerous ways to implement it, and the chosen approach should reflect the project's requirements and constraints. The method described here is one implementation and can be adapted or extended to suit different environments.

Next Steps

The current implementation lays a strong foundation, but there are several enhancements planned to make the framework more robust and user-friendly:

- **Expand AI Capabilities:** Integrate additional LLM (Large Language Model) models so the framework can support multiple AI options, improving flexibility and accuracy.
- **Improve Reporting:** Enhance the self-healing report to include the file name and function where updates are required, making maintenance faster and more intuitive.
- **Introduce Scoring for Heuristics:** Implement a score-based system to prioritize the most likely alternative locators, reducing unnecessary retries and improving efficiency.
- **Unified HTML Report:** Develop a consolidated HTML report that combines overall test results with self-healing logs, providing a single source of truth for execution details.
- **Automated Script Updates:** Explore the possibility of automatically updating test scripts with new locators and pushing changes back to Git for complete end-to-end self-healing.

Impact of Self-Healing Test Scripts

This depends on the stage of the project, what technology and tools are used for development, and the member of the team responsible for the development. Script maintenance issues can be reduced drastically if the development team works hand in hand with the test automation team, which can ensure minimum failures in the automation scripts. If proper measures are adopted by the development and test team working together, there will only be minimal element locator identifier issues arising from regular project maintenance, and the self-healing approach used here will be able to handle these and thus save considerable time. This self-healing approach helps teams develop independently in parallel while minimizing test failures and thus contributing to significantly faster development and testing iterations.

Existing Alternatives:

There are tools that make use of AI and images for self-healing in the industry, but most of these are paid tools. For open source, there is healenium, but this is a java based library. At the moment, our solution with custom built functions with open-source libraries described here is the only self-healing automation testing solution.

Conclusion

Self-healing test scripts represent a significant advancement in test automation. When implemented correctly, they can dramatically reduce maintenance effort and turnaround time, allowing teams to focus on higher-value tasks. However, success depends on a thorough understanding of the application under test and careful design of heuristic and AI strategies tailored to its complexity.

In practice, this approach not only resolves minor to moderate locator issues but also ensures smoother CI/CD pipelines by preventing failures caused by UI changes. The inclusion of detailed reports further simplifies script maintenance, making the process almost effortless.

The use of Mistral Large as an AI model has shown promising results in identifying elements accurately within the limited scope tested. While early outcomes are encouraging, broader testing in complex environments is essential to validate its effectiveness. As confidence grows, the integration of GenAI into test automation frameworks can evolve from a simple assistive feature to a core capability, driving efficiency and resilience across testing processes.