www.ijecs.in
International Journal Of Engineering And Computer Science
Volume 14 Issue 11 November 2025, Page No. 27819-27829
ISSN: 2319-7242 DOI: 10.18535/ijecs/v14i11.5299

Machine Learning Approaches for Auto-Repairing Test Steps in UI Automation: A Review

Alex Thomas Thomas
Saransh Inc, USA

Abstract
In modern software development, UI automation testing is one of the most important aspects of application quality assurance. Excessive UI changes make automated test scripts fragile, hence resulting in flaky tests and exorbitant maintenance costs that expose QA teams to unbearable stress. To overcome such issues, recent progress in Machine Learning (ML) and Artificial Intelligence (AI) has been leveraged to develop auto-repair mechanisms for automatically detecting and repairing broken test steps, making tests more reliable and reducing human effort. This survey presents an overview of current ML-based solutions to self-healing UI test automation, including element locator repair, flaky test stabilization, and wait automation. We address methods from heuristic to large language model-based repair and dynamic web and mobile test script modification. Leading frameworks describe how the integration of ML can make test maintenance easier and the tests themselves more resilient. Our presentation recognizes that while locator repair approaches have matured and offer immediate practical advantages, newer LLM-based approaches offer improved semantic understanding but at high costs of explainability and computational complexity. We talk about current constraints on the generalizability and explainability of test repair and provide guidelines for future research on advancing intelligent, adaptive QA automation solutions that reduce downtime and enhance software delivery quality.

Keywords: UI Test Automation, Test Script Maintenance, Automated Test Repair, Machine Learning, Self-Healing Test Automation, Flaky Test Mitigation

1. Introduction

UI testing is an important phase of the software development life cycle so that applications work as desired and are acceptable to users. Among numerous types of testing, UI testing is essential to verify how users interact with applications. Automated UI testing has been trendy because it can perform repetitive tasks extremely fast. However, UI tests are extremely sensitive—they fail because of slight layout variations, element attributes, or application flow even when the underlying action does not alter. It generates high test maintenance overhead, decreased reliability, and increased human effort in script recovery. As a result, researchers and practitioners began exploring self-healing or auto-repairing mechanisms with the ability to automatically locate and fix erroneous test steps. Machine Learning (ML) developments allow for the development of intelligent methods to enhance test resilience and maintainability. For example, regression element matching techniques have been proposed to repair faulty locators by correlating elements between UI versions [1], and tree-based structural matching enables dynamic structural comparison of dynamic web pages [2]. The remaining approaches use Large Language Models (LLMs) to improve the accuracy of web element localization [3] or learn waits automatically to reduce flaky test failure [4]. Test oracles have also been integrated with ML models for facilitating reasoning about element behavior at repair time [5]. Deep learning [7] and exploration-based methods [6] are being investigated to be applied to test script repair in web and mobile domains. As the solutions become mature, there is a growing need to critically evaluate the

state of ML-based UI test repair techniques to be aware of existing trends, challenges, and research directions on test automation.

This paper provides a comprehensive review of machine learning techniques employed in automated UI test repair. We categorize current methods into five general categories: locator repair techniques, flakiness management, LLM-based techniques, mobile and cross-platform techniques, and self-healing frameworks. We identify the strengths and weaknesses of each method, highlight the research gap, and provide directions for future research to improve the area.

This survey systematically examines ten prominent papers from 2017 to 2024, which are among the most significant contributions to test repair in UI based on ML. We include approaches that: (1) apply machine learning for repair, (2) concern web or mobile test automation at the UI level, and (3) provide empirical evidence of their effectiveness. Our analysis reveals a picture of scattered methodologies with varying objectives and performance metrics, causing a knowledge gap addressed by this review through integration of current approaches, contrast of their merits and demerits, and identification of directions for future research.

2. Overview of UI Automation and Challenges

Automated User Interface (UI) testing is of the utmost significance in software program verification by imitating user interaction to ascertain anticipated functionality [8]. Though of the utmost significance, automated UI tests are well-known for being flaky and prone to failure when UI elements undergo frequent changes, dynamic content is updated, or timing mismatches take place. The ensuing impact, popularly referred to as flaky or broken tests, significantly escalates maintenance costs and reduces confidence in automated test suites [1], [8].

2.1 Common Causes of Test Failures

- **Broken Locators**: UI elements are generally identified by locators such as XPath, CSS selectors, or element IDs. Developers changing the UI structure may cause these locators to no longer point to the same elements, and test failures are seen.
- **Timing Issues**: Asynchronous calls, network latency, and dynamic content loading can cause timing mismatches between test execution and the state of the application, rendering the tests flaky and failing intermittently.
- **UI Structural Changes**: Drastic redesigns or incremental modifications to DOM structure can render existing element identification methods invalid, requiring automatic test script modifications manually.
- **Dynamic Content**: Elements that appear conditionally or transform based on user interactions or external data sources are difficult for static test scripts.

2.2 Traditional Approaches and Limitations

Traditional approaches to repairing such tests are generally by manual intervention, which is both time-consuming and error prone. Static fallback approaches, such as trying multiple locator strategies sequentially, provide limited resilience and are unable to adapt to emerging UI patterns. Manual script maintenance increasingly becomes unwieldy as application complexity grows, and release cycles compact.

2.3 The Self-Healing Paradigm

To address these issues, the concept of auto-repair or self-healing test automation has been proposed. Self-healing test frameworks automatically detect test failures caused by UI changes and attempt to repair test scripts without human involvement. These systems typically have a common workflow:

- **Detection**: Detect test failure and determine the cause (e.g., broken locator, timing issue)
- **Analysis**: Apply ML models to understand the UI change and identify the target element
- **Repair**: Develop and apply a repair (e.g., update locator, change wait time)

- **Validation**: Verify the fixed test produces correct results

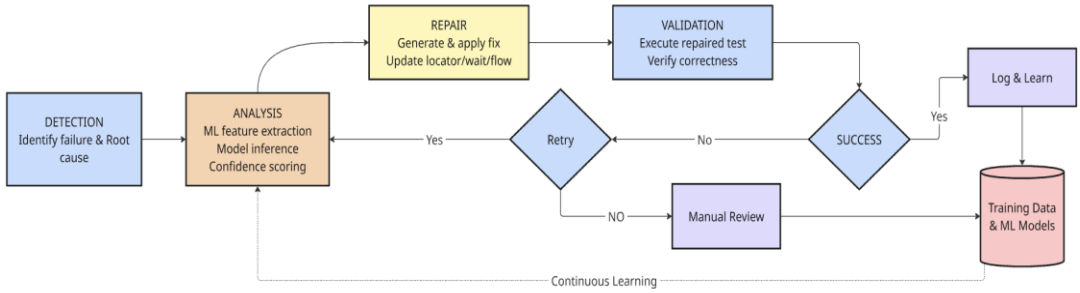

Figure 1. Self-Healing Architecture

3. Categories of ML-Based Auto-Repair Approaches

This part categorizes the examination of ML-based auto-fixing approaches into five main classes based on their problem focus and approach.

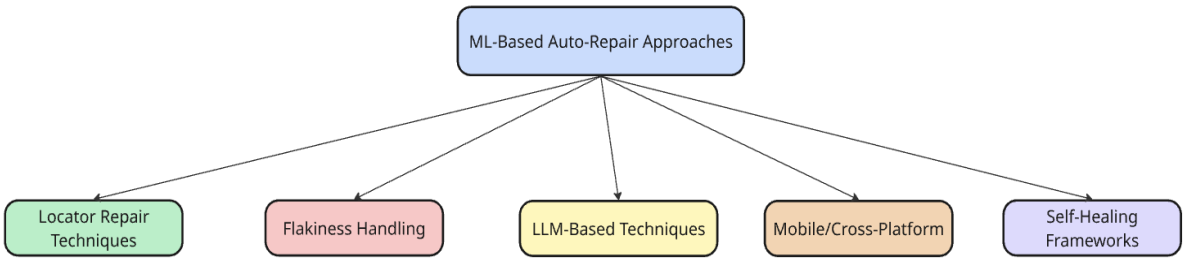

Figure 2. Taxonomy of ML-Based Auto-Repair Approaches

3.1 Locator Fixing Techniques

Locator fix is a simple form of auto-fixing UI tests, wherein element locators are identified and updated by machine learning models when they become broken or incorrect.

- **Iterative Element Matching**: Lin et al. [1] proposed an automated repair method for web UI tests using iterative element matching. The approach matches elements between UI versions by considering multiple attributes (tag name, text, location) and progressively improving matches. The approach caused significant minimization of human repair effort but requires multiple cycles of execution before converging to accurate repairs.
- **Flexible Tree Matching**: Brisset et al. [2] introduced SFTM (Similarity-based Flexible Tree Matching), which facilitates efficient web page matching by tree structure matching in the DOM tree based on similarity measurements. It can accommodate more robust dynamic web page form handling than strict structural matching but incurs computation overhead for complex DOMs.
- **Context-Aware Locator Suggestion**: **Leotta et al. [9]** proposed a multi-locator approach that uses multiple contextual cues and voting procedures to select robust locators for web test scripts. The approach aggregates results from different locator generation algorithms, each considering different element properties, context, and attributes. COLOR examines element properties, context surrounding, and visual attributes to recommend alternative locators. The approach was found to be highly accurate but is limited to certain web technologies.

These methods reduce the need for manual effort by catching UI change patterns, yet they may struggle with radical redesigns or entirely new UI patterns.

3.2 Handling Test Flakiness

Flaky testing, frequently caused by timing issues or dynamic content, is among the largest challenges addressed by ML-driven auto-repair.

- **Intelligent Wait Generation**: Liu et al. [4] proposed WEFix, which produces explicit waits strategically for efficient web end-to-end flaky tests. Their machine learning model is trained on real latency patterns to generate the appropriate wait conditions, stabilizing more tests that fail every now and then due to timing-related issues. WEFix suppressed flaky test failures significantly without affecting test execution efficiency.
- **Empirical Study of Flakiness**: Romano et al. [8] empirically studied UI-based flaky tests and revealed common patterns and root causes. The findings showed that problems related to timing cause an extremely high percentage of flaky failures, which led ML-based solutions to stabilize tests by using wait strategies and retry logic trained with ML.

These avoidance methods of flakiness enable more reliable end-to-end test executions by addressing time-related issues of UI testing that traditional static approaches cannot handle effectively.

3.3 Large Language Model (LLM) Usage

The most recent innovations employ Large Language Models to enhance the efficiency of test script fixing, heralding a new era for self-healing test automation.

- **LLM-Improved Element Localization**: Nass et al. [3] explored improving web element localization through large language models. The approach utilizes natural language explanations and context semantics of UI elements to improve locator robustness. LLMs provide semantic awareness over syntactic pattern matching but at the expense of tremendous computational resources.
- **Explained Repair with Explanation Checking**: Xu et al. [5] proposed directing ChatGPT to fix web UI tests via explanation-consistency checking. Their orchestration technique employs LLMs to produce repair proposals and then verifies the proposals by using consistency checking techniques. This process combines LLMs' reasoning capability with verification techniques to ensure the correctness of repair, though explainability remains an issue.

LLM-based approaches can address complex repair scenarios that require semantic awareness but are constrained by resource consumption and repair choice explainability.

3.4 Mobile and Cross-Platform Repairs

Machine learning approaches also address mobile and cross-platform UI test repairs with an understanding of the unique issues in mobile environments.

- **Semantic matching**: Liu et al. [6] proposed an adaptive semantic matching-based test reuse approach for Android mobile applications. The approach provides reusability of test cases across different apps via semantic matching of UI elements and adaptation of test scripts to analogous UI patterns, reducing the effort in creating new test scripts for similar-functionality applications.
- **ATOM for Mobile Maintenance**: Li et al. [10] introduced ATOM (Automatic Maintenance of GUI Test Scripts), which uses deep learning to automate GUI test scripts maintenance for evolving mobile applications. ATOM tracks the change in UI elements and automatically updates test scripts, though it faces scalability challenges in maintaining diversified mobile platforms.

These mobile-oriented solutions address platform-specific challenges such as diverse screen sizes, interaction types, and rapid UI transformation common in mobile application development.

3.5 Self-Healing Frameworks

Self-healing test automation frameworks use several ML techniques to detect, diagnose, and heal the test failure independently, which are end-to-end solutions and not specialized techniques. These frameworks typically combine a couple of techniques from the above classes—element matching, wait

generation, and adaptive locator strategies—into combined systems. They leverage AI and ML for end-to-end test repair with minimal effort in manual debugging. Commercial and research frameworks in this class are a paradigm shift in quality engineering from reactive manual repair to proactive automated maintenance. Self-healing frameworks demonstrate greater maintainability and resilience of scale UI automation yet require careful setup and ongoing tuning to continue to be effective as applications are created.

- **AI/ML-Based Self-Healing Framework**: Saarathy et al. [7] proposed a self-healing test automation framework with AI and ML techniques to automatically diagnose, detect, and repair test failures. Their framework combines a number of ML techniques including pattern recognition and adaptive learning how to create solid test automation frameworks that minimize manual intervention without sacrificing test reliability across varying applications.

3.6 Supporting Techniques and Strategies

Aside from the base ML-based techniques, several supplementary techniques also enhance self-healing capabilities:

- **Element Discovery Methods**: There are several methods to discover UI elements even when primary identifiers change, e.g., switching between XPath and CSS selectors, text-based identification with visible labels, and visual recognition.
- **Machine Learning Algorithms**: Pattern learning and anomaly detection algorithms form the basis of most self-healing systems, learning UI change patterns and detecting failures that trigger adaptive fixes.
- **Dynamic Object Mapping**: Version-controlled updating and maintaining element repositories make script updates easier. Dynamic object maps store the history of how elements change with time.
- **Fallback Mechanisms**: Retry mechanisms and graceful degradation make test continuation possible even when main repair methods fail, with retrying element identification by other strategies or running tests with reduced capability.

Visual Testing Methods: Screenshot comparison and image recognition provide other validation methods where attribute matching fails, visually recognizing items and detecting UI differences.

Table 1: Supporting Techniques for Self-Healing Test Automation

Technique	Description	Details
Element Identification Strategies	Strategies to locate UI elements even when identifiers change	XPath and CSS Selectors: Switches between available selectors. Text-Based Identification: Uses visible text/labels if attributes change [2]
Machine Learning Algorithms	ML-powered methods to detect patterns and anomalies in UI changes, triggering adaptive repairs	Pattern Recognition: Learns and recognizes UI changes. Anomaly Detection: Detects failures due to UI changes, triggers or suggests automated fix [1]
Dynamic Object Mapping	Maintaining and updating element repositories and tracking versions to handle UI updates	Object Repositories: Dynamic, auto-updated element maps. Version Control: Tracks element versions for streamlined script updates [3]
Fallback Mechanisms	Redundant adaptation steps to maintain test continuity if primary strategies fail	Retry Logic: Re-attempts element identification with alternates. Graceful Degradation: Test runs with reduced capability when key elements are missing [1], [2]
Visual Testing Techniques	Uses visual feedback for element identification and UI validation	Image Recognition: Detects elements visually. Screenshot Comparison:

	when attribute matching is insufficient	Validates UI by comparing before/after screenshots to pinpoint discrepancies [1]

4. Comparative Analysis of ML-Based Repair Methods

Our evaluation demonstrates that different ML-based repair techniques exhibit different strengths, limitations, and best use cases. Locator repair techniques [1,2,9] are the most mature, achieving high accuracy for structure changes with low computation overhead, making them suitable for CI/CD pipelines processing frequent incremental UI changes, though they break under large redesigns. Flakiness handling approaches [4,8] effectively reduce timing-related failures, though wait-based solutions may moderately increase execution time. LLM-based approaches [3,5] provide improved semantic understanding for complex scenarios at the cost of heavy computational overhead with inference times orders of magnitude slower than traditional methods due to model complexity [3,5], are plagued by explainability problems and high resource demands, and are therefore restricted to practical deployment on high-value test cases. Mobile repair approaches [6,7,10] address platform-specific problems. ATOM [10] requires substantial training data from diverse applications to generalize effectively, while deep learning approaches [7] depend on extensive labeled datasets for accurate script maintenance. Self-healing systems integrate multiple approaches for end-to-end solutions but require substantial initial implementation effort and ongoing maintenance. Key innovation challenges in the use of such methods are integration challenge, test team lack of ML experience, explainability limits affecting trust and debuggability, and resource consumption concerns. Practitioners can start with mature locator repair for early ROI, add flakiness treatment when timing issues dominate, experiment with LLM methods for hard cases with human oversight, use platform-specific methods for mobile apps, and carefully balance automation level against team capability and organizational constraints.

Table 2: Comparative Analysis of ML-Based Repair Methods

Method Type	**ML Technique / Approach**	**Platform Focus**	**Evaluation Metrics Used**	**Key Strengths**	**Key Limitations**
Locator Repair	Iterative Element Matching [1]	Web	Success rate, accuracy, manual effort reduction	Handles structural changes, reduces manual fixes	Fails with large UI redesigns, needs multiple runs
Locator Repair	Flexible Tree Matching [2]	Web	Success rate, accuracy, FP/FN rate, time overhead	Fast and accurate matching	Limited to structural issues only
LLM-Based Repair	LLM for Element Localization [3]	Web	Time overhead, computational cost	Semantic understanding, adaptable	Resource-heavy, inconsistent outputs
Flakiness Handling	ML-generated Explicit Waits [4]	Web	Manual effort reduction, flakiness reduction, stability	Improves timing-based test stability	Increases test execution time
LLM-Based Repair	Explanation-Consistency via ChatGPT [5]	Cross-Platform	Computational cost	Handles novel scenarios, semantic reasoning	Low explainability, heavy infrastructure required
Mobile Repair	Adaptive Semantic Matching [6]	Mobile (Android)	Manual effort reduction, test reuse rate	Enables test reuse across similar apps, reduces script	Limited to similar UI patterns, generalization challenges

				creation effort	
Self-Healing Framework	AI/ML-Based Self-Healing [7]	Cross-Platform	Computational cost, automation coverage	Integrated approach, reduces manual intervention	Implementation complexity, requires ongoing tuning
Flakiness Analysis	Empirical Study + Heuristics [8]	Web	Flakiness reduction, test stability	Reveals flaky patterns, improves test design	No repair mechanism, limited to timing issues
Locator Repair	Context-Aware Locator Recommender (COLOR) [9]	Web	Success rate	Multi-source clues for locator repair	Complex logic, needs diverse input clues
Mobile Repair	Heuristic Script Maintenance (ATOM) [10]	Mobile	Pass rate improvement	Automated script updates, robust for mobile changes	Generalization challenges, integration complexity

5. Research Gaps and Challenges

Despite remarkable advancements in ML-based UI test auto-repair, several critical research gaps and real-world challenges persist, limiting the scalability and adoption of existing solutions.

5.1 Absence of Standardized Benchmarks

The field lacks publicly available, standardized benchmark test sets for training and evaluating repair processes. Experiments are found to source draws from different application domains, test sets, and metrics, making reproducibility and fair comparison difficult. Effort should be delivered in the future through the preparation of curated sets of actual-world test failures, standardized UI evolution cases, shared evaluation procedures, and open-test suites to enable consistent benchmarking across experiments.

5.2 Limited Explainability

Most ML-based repair systems, particularly those using large language models (LLMs), are opaque in their reasoning, restricting their use in safety-critical or regulated applications [3], [5]. These need to be made interpretable, rationale generation for repairs, confidence scoring mechanisms, and human-in-the-loop validation processes to enhance user responsibility and confidence.

5.3 Limited Generalization Across Platforms

Most recent solutions are platform-specific—web or mobile-oriented—and don't scale well across heterogeneous environments such as desktop applications [6, 8]. Cross-platform repair techniques must contend with variations in UI frameworks, DOM layouts, rendering engines, and user interaction patterns [5].

5.4 Challenges in CI/CD Integration

Integrating fix into CI/CD pipelines is difficult due to the computational overhead and latency introduced by most ML-based approaches [2], [3], [7]. Real-time or near-real-time repair, low resource utilization, integration with existing test framework support, and conformance with automated verification gates are required for efficient integration.

5.5 Inefficient Handling of Multi-Step Test Failures

Current methods primarily address individual test step failures but are not sufficient when dealing with complex, multi-step UI changes affecting entire test flows [1, 9]. There is a need for techniques that can infer test purpose and dynamically adapt entire test scenarios according to UI evolution.

5.6 High Training Data Requirements

Deep learning and LLM-based methods both need ample quantities of labeled training data, which in the case of proprietary or domain-specific applications are often unavailable [3]. While few-shot and zero-shot learning offer potential alternatives, it remains to be seen that they can be used in the context of UI test repair.

6. Disadvantages and Limitations of Self-Healing UI Tests

While self-healing UI tests are highly beneficial with the added strengths of minimal human intervention and better resilience against UI changes, several inherent limitations must be balanced for total implementation in test automation programs.

6.1 Complexity of Implementation

Applying self-healing solutions might be technically challenging, typically requiring careful setup and considerable initial setup effort. Integrating these solutions with current automation frameworks would mean nontrivial tool and process modification, complicating the project [2]. The application of self-healing frameworks usually calls for advanced expertise in machine learning algorithms and pattern recognition techniques, which may not be found within traditional testing organizations. Moreover, coupling locator suggestion systems with element matching algorithms brings architectural complexity that must be taken seriously [9].

6.2 Over-Reliance on Automation

Excessive dependency on self-healing mechanism can result in less manual intervention, and teams may overlook fine-grained test cases, only which can be identified by human testers [1]. This over-reliance could provide false assurance of protection since not all faults or test failures are recoverable through an automated mechanism. Automated repair tools have been known to conceal underlying application issues that can only be correctly identified by human inspection [8]. Moreover, automated self-healing can lead to reduced interaction with test maintenance, and teams might not pick up on significant changes in application behavior that call for test redesign rather than repair.

6.3 Maintenance Overhead

Self-healing techniques will still decrease the number of manual fixes, but there is always a requirement for continuing maintenance to maintain repair logic effective as applications evolve and expand. The algorithms themselves can also introduce computational overhead, which can reduce the efficiency of large test suites [3]. For instance, techniques employing large language models for element localizations can require a lot of processing time and resources [3]. Similarly, iterative element matching algorithms require several rounds of execution in order to identify converging correct repairs, which are expensive in terms of total test execution time [1]. The cost of maintenance also extends to the training data and models underpinning self-healing systems, which must be regularly updated to maintain accuracy [7].

6.4 Inaccurate Adaptation Risk

Self-healing techniques can incorrectly classify or not correctly adapt UI elements, resulting in false positives or negatives in the test outcomes [1]. Failure to comprehend context or follow application state can render repairs that deviate from anticipated user flows, impacting overall test dependability.

Experiments have indicated that automatic repair mechanisms have a tendency to exert erroneous explicit waits or timing changes that alter the semantics of tests [4]. Moreover, similarity-based matching algorithms can get wrong matches between elements in complex UI structures and render passing tests that test erroneous application functionality [2]. The issue is particularly acute in mobile settings where UI evolution could entail vast structural changes that confuse repair algorithms [6], [10].

6.5 Limited Scope of Healing

Success of self-healing solutions is narrow, particularly where handling large-scale UI redesign or architectural changes involving manual updates of test scripts [10]. Some methodologies are streamlined for specific locator mechanisms or technologies, restricting their applicability to other platforms and domains. For example, methodologies fine-tuned for web applications would not work well when transferred to mobile environments without drastic retooling [6]. Also, self-healing approaches based on historical test run data or specific UI patterns may fail when applications undergo radical redesigns that render past learning redundant [9].

These limitations highlight that efficient adoption of self-healing UI testing paradigms must be an equilibrium game, combining robust automation with judicious manual verification and ongoing adjustment of logic as well as test resources. Organizations must carefully balance trade-offs between automation benefits and risks of over-engineering, ensuring human expertise is never sacrificed to the game [1], [8].

7. Future Research Directions

7.1 LLM and Visual AI Integration

The integration of Large Language Models (LLMs) with visual perception of UI elements is a potential way to enhance semantic repair capabilities [3], [5]. Latest advancements in multimodal AI that embed textual and visual reasoning have a lot of potential for UI context and user intent inference. Combining these approaches could offer intelligent, adaptive self-healing architectures that reason semantically when handling UI changes rather than structural heuristics.

7.2 Benchmark Datasets and Evaluation Frameworks

Benchmark datasets standardized across varied application domains and UI technologies have to be created urgently. They should have representative real-world failure tests, cross-application patterns of UI evolution, repair scenarios with established ground truth, and cross-platform test cases. Standardized test protocols will be created to enable reproducible comparisons between methods and accelerate progress in the field.

7.3 Model Explainability and Trust

Improving the explainability of ML and LLM-based repair options is crucial in safe and controlled environments. The research should focus on approaches that provide transparent explanation of repair recommendations, thereby building practitioner trust and facilitating debugging if automated repairs fail. Promising areas include attention-based visualizations, counterfactuals, confidence scores-based uncertainty quantification, and interactive refinement processes.

7.4 Cross-Platform and CI/CD Integration

Effective repair methods that are general across web, mobile, and desktop platforms [6], [7], [10], and can integrate naturally into CI/CD pipelines, is essential for practical adoption. Light-weight and low-overhead repair mechanisms meeting CI/CD time constraints are especially relevant. Research areas include incremental repair to minimize overhead, parallel analysis to accelerate results, cache-based optimizations for frequent patterns, and APIs to ease integration with mainstream CI/CD platforms.

7.5 Hybrid Approaches

Combining symbolic reasoning with ML-based methods can address current problems in handling advanced, multi-step UI adjustments [1], [9]. Hybrid systems are capable of leveraging rule-based systems for processing well-documented repair patterns and using ML models for obscure or unexpected situations, making them both reliable and flexible. Some potential approaches include combining expert systems and ML models, constraint-based repairs with added learned heuristics, formal verification of patches written through the use of ML, and human-in-the-loop active learning for continuous improvement.

7.6 Few-Shot and Zero-Shot Learning

To counteract the heavy training data demand of deep learning as well as LLM-based approaches, few-shot and zero-shot learning techniques offer viable alternatives. They could enable efficient self-healing test automation in technical or proprietary environments where large, labeled training data are unavailable [3], [7].

7.7 Test Intent Understanding

Future research will need to look beyond having just the locators fixed to having an understanding and hold on underlying intent of tests. That involves having a system that can reason over purpose and meaning of test cases and rewriting entire test scenarios when UI changes affect more than one step or alter user workflows.

8. Conclusion

This review paper reviewed machine learning techniques for auto-fixing test steps in UI automation. Our evaluation indicates that ML-based strategies—such as iterative element comparison [1], tree comparison based on similarity [2], localization enhanced by LLM [3], [5], intelligent wait generation [4], [8], and semantic matching-based test reuse [6], automated mobile script maintenance [10] and integrated self-healing frameworks [7]—have greatly enhanced test robustness and maintenance overhead against manual traditional methods. The analyzed techniques are grouped into five large categories: locator repair mechanisms, flakiness avoidance methods, LLM-based methods, cross-platform and mobile solutions, and self-healing solutions. Each category's strength appears in some features and the respective weaknesses are as important. Locator repair mechanisms suit UI structural modification best but are not capable of dealing with substantial redesigns. Flakiness handling mechanisms increase test stability but work only with timing-related issues. LLM-based approaches provide semantic understanding but at the costs of computation and explainability. Platform-agnostic solutions eliminate platform fragmentation but suffer from scalability. But there are significant challenges in implementation complexity, cross-platform generalization, scalability, and adaptation precision. Imperative research gaps are the lack of standardized benchmarks, weak explainability of repair options, difficulties in cross-platform generalization, and CI/CD integration problems. The fusion of LLMs with vision-based AI promises more contextual awareness, though upcoming research must contend with model explainability, benchmark dataset development, and native CI/CD pipeline to realize the full potential of automated self-healing. Auto-repair techniques through ML are full of promise to reduce costs and test quality in sophisticated software environments, provided release achieves the ideal balance of automation potential with timely right human intervention as well as ongoing improvement. The region is growing rapidly, and LLM solutions are leading the charge, but adoption in real-world scenarios will entail bridging the cited research gaps and designing solutions that can be easily integrated into existing development processes. For those implementers who wish to take advantage of these approaches, we recommend starting with established locator repair methods [1,2,9] for short-term ROI, addressing flakiness through intelligent wait generation [4,8], and pioneering LLM-based approaches cautiously [3,5] on high-risk test cases under human monitoring. Choice of method should balance automation benefits against team expertise, machine resources, and app limitations. As the software development market continues to increase release cycles and embrace

continuous delivery practices, intelligent, adaptive test automation becomes ever more important. The meeting point of machine learning, large language models, and visual AI technologies opens up a path to truly autonomous test maintenance systems able to support high-speed application evolution without compromising on test quality and reliability.

References

1. Lin, Y., Wen, G. and Gao, X., 2023, September. Automated fixing of web ui tests via iterative element matching. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (pp. 1188-1199). IEEE.
2. Brisset, S., Rouvoy, R., Seinturier, L. and Pawlak, R., 2023. Sftm: Fast matching of web pages using similarity-based flexible tree matching. *Information Systems*, *112*, p.102126.
3. Nass, M., Alégroth, E. and Feldt, R., 2024. Improving web element localization by using a large language model. *Software Testing, Verification and Reliability*, *34*(7), p.e1893.
4. Liu, X., Song, Z., Fang, W., Yang, W. and Wang, W., 2024, May. Wefix: Intelligent automatic generation of explicit waits for efficient web end-to-end flaky tests. In *Proceedings of the ACM Web Conference 2024* (pp. 3043-3052).
5. Xu, Z., Li, Q. and Tan, S.H., 2023. Guiding chatgpt to fix web ui tests via explanation-consistency checking. *arXiv preprint arXiv:2312.05778*.
6. Liu, S., Zhou, Y., Han, T. and Chen, T., 2022, December. Test reuse based on adaptive semantic matching across android mobile applications. In *2022 IEEE 22nd International Conference on Software Quality, Reliability and Security (QRS)* (pp. 703-709). IEEE.
7. Saarathy, S.C.P., Bathrachalam, S. and Rajendran, B.K., 2024. Self-Healing Test Automation Framework using AI and ML. *International Journal of Strategic Management*, *3*(3), pp.45-77.
8. Romano, A., Song, Z., Grandhi, S., Yang, W. and Wang, W., 2021, May. An empirical analysis of UI-based flaky tests. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)* (pp. 1585-1597). IEEE.
9. Leotta, M., Stocco, A., Ricca, F. and Tonella, P., 2015, April. Using multi-locators to increase the robustness of web test cases. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)* (pp. 1-10). IEEE.
10. Li, X., Chang, N., Wang, Y., Huang, H., Pei, Y., Wang, L. and Li, X., 2017, March. ATOM: Automatic maintenance of GUI test scripts for evolving mobile applications. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)* (pp. 161-171). IEEE