AI-Powered Self-Healing and Adaptive Software Testing Framework

Grishma Lad[1], Priyanka Sharma[2]
[1]Computer Engineering Department, Bhagwan Mahavir University (BMU), India
[2]Professor Computer Engineering Department, Bhagwan Mahavir University (BMU), India

Abstract - Automated test execution has become a fundamental part of modern software development, especially with the growing adoption of spry methodologies and continuous integration practices. While automation works as a bridge in improving efficiency and reducing manual effort, it also introduces new obstacles. One of the most common issues is the fragility of test scripts, which often fail due to small and frequent changes in the application, for example, modifications in user interface elements or underlying structure. These failures are not always due to the actual defects, but rather by the minor inconveniences, which leads to debugging and maintenance efforts that are not fundamentally necessary. In recent times, researchers have explored the usage of artificial intelligence (AI) and machine learning (ML) to address some of these limitations in software testing execution. Building on these ideas, this paper presents an AI-powered self-healing and adaptive software testing framework that aims to make machine-driven testing stronger and more efficient. The proposed framework is designed to automatically predict test failures, analyze their root causes, and update test cases without requiring manual intervention. It uses a combination of techniques, including similarity-based matching for identifying changes in application elements, natural language processing for generating test scenarios, and machine learning models that learn from past executions.

Another important aspect of the framework is its ability to adapt over time. Instead of repeatedly failing in similar situations, the system improves by learning from previous test runs and applying more accurate fixes in future executions. This makes the testing process more stable and reduces the overall maintenance burden on developers and testers.

All in all, the goal of the approach is to minimize incorrect test failures, improve reliability, and make automated testing more practical in fast-changing development environments. While the framework is presented at a conceptual level, it highlights the potential of AI in transforming traditional testing practices into more intelligent and adaptive systems

I. INTRODUCTION

The pressure on software testing is at an all time high as development teams have started releasing updates more frequently. With CI and CD pipelines running on almost every commit and releases happening weekly or even daily, testing is no longer something that can take its own time. If a testing infrastructure needs days of manual fixing every time there is a minimal UI change, it quickly becomes a pipeline instead of helping the process.

Automation solved the problem of speed to some extent, but it introduced another issue which is brittleness. Many automated tests depend heavily on specific locators like X-Path or element IDs. Even a small refactor or UI change can cause these tests to fail. Over time, developers start noticing this pattern and may begin to trust the test results less and less, which defeats the purpose of having automations in the first place.

Research also supports this observation. Hamza and Jonsson [4] found that in agile environments, a significant amount of time is spent maintaining existing test cases rather than creating new ones. This is quite ironic, because automation is supposed to reduce effort, not shift it into another form of repetitive work.

This is where machine learning starts to feel useful. Instead of treating every failure the same way, a system can learn to create differences between an actual defect and a failure caused by a minor change, such as a locator modification. If it can handle these minor issues on its own, it can save a lot of manual effort, since these kinds of failures are very common.

This paper builds on that idea and proposes a system that focuses on handling such situations more mindfully. The goal is to detect failures quickly,

understand what caused them, and improve over time by learning from previous test runs.

II. LITERATURE SURVEY

The idea of applying machine learning to testing problems is not new. Zhang and Harman from Ref: [1] showed that ML could usefully rank defects by severity modest in hindsight, but it established that learned models could contribute something real to the QA pipeline beyond simply running assertions.

Pan from Ref: [2] tackled test generation, demonstrating NLP can parse user stories and produce executable test scenarios. Useful in embedded teams where the source of truth is often an informal ticket. The limitation which anyone in this space will recognize is that natural language requirements are under specified, and generated tests reflect that confusion.

Sharma and Kumar from Ref: [3] used historical execution data to assume where testing effort would be needed in future sprints. Useful for planning, but it doesn't address the problem knowing 3 days of maintenance are coming isn't the same as not needing those 3 days.

Leotta et al. From Ref: [6] did careful designated work on Selenium-based test suites, comparing how different locator strategies affects stability over time. Their conclusion that X-Path heavy tests degrade significantly faster than attribute-based ones has direct implications for the self-healing module. The brittleness isn't random; it concentrates around specific patterns.

Several tools have taken a repair-first approach. WATERFALL from Ref: [9] patches the record and replay scripts incrementally when they break effective for stable applications, less for larger structural shifts. PESTO from Ref:[12] is more interesting, it identifies UI elements by visual characteristics rather than DOM properties, sidestepping the locator's brittleness problem but introducing new challenges when layouts change.

Biagiola et al. From Ref: [8] investigated how implicit dependencies between tests exemplifies failures a single broken precondition can invalidate 20 down cases. That insight shaped how we think about failure isolation in the detection module. Feng et al. From Ref: [7] focused on requirement test traceability, side-by-side to what we're doing, but operating at the specification layer rather than the execution layer.

Mnih et al.'s deep RL work from Ref: [10] is relevant not in its specifics but the original DQN context was game-playing but, in its formulation, an agent learning actions that maximize long-term reward. Translated to testing, that reward is whether the patched test passes without introducing regressions. Levenshtein distance from Ref: [11] is older and simpler, but effective at catching element-renaming failures.

What the existing literature lacks is integration. Each piece NLP generation, similarity-based repair, RL-driven optimization has been validated individually. Nobody has built a system combining them into a coherent feedback loop. That's what this paper is trying to design.

III. PROPOSED FRAMEWORK

The framework we propose is composed of 5 modules. They're designed to hand off to each other cleanly, but each can also be understood and evaluated independently.

Failure Detection sits at the front of the pipeline. It runs concurrently with test execution and captures everything that might be relevant to diagnostic logs, DOM snapshots, screenshots, timing data. The goal is to record enough context that the analysis module doesn't have to reconstruct what happened.

Root Cause Analysis compares the captured failure context against stored baselines from previous successful runs. Levenshtein distance is the primary tool for element-level diffing it handles the common case of incremental identifier changes well. For structural changes, the module falls back to a broader DOM comparison.

Self-Healing takes the root-cause output and applies a patch. In the simple case a locator changed from "login-btn" to "signin-btn" the update is deterministic. More ambiguous cases receive a confidence score; low-confidence patches are queued for human review rather than applied silently.

Adaptive Learning is what separates this from a static repair script. Each failure-and-fix cycle feeds back into the model, shifting its prior toward more accurate diagnoses. RL drives the patch-selection step, with the reward signal being whether the patched test passed on its next run without regression.

Test Generation handles the coverage side. Using NLP, it processes requirement documents and user stories to produce test case skeletons, which the rest of

the framework can then monitor and maintain. Coverage gaps identified during failure analysis can also trigger the generation of additional tests for under-tested areas.

The 5 modules form a closed loop: execution feeds detection, detection feeds analysis, analysis feeds healing, healing feeds learning, learning feeds the next cycle. The system degrades gracefully if any module fails in the worst case, it logs the failure without attempting a patch.

$S(e, t) = \alpha \cdot f\_text(e, t) + \beta \cdot f\_attr(e, t) + \gamma \cdot f\_context(e, t)$

IV. RESULTS AND DISCUSSIONS

The most direct value is noise reduction. A significant fraction of CI test failures are false positives scripts that haven't caught up with UI changes, not real bugs. Filtering those automatically means developers stop reflexively dismissing red builds as "probably just a locator thing" and start treating failures as meaningful again.

Reduced maintenance overhead is the second major benefit, and arguably the one with clearer ROI. The hours a team spends updating tests after each sprint's UI changes are hours not spent on new feature coverage or exploratory testing. Self-healing doesn't eliminate that cost entirely, but it should concentrate it on the cases that genuinely need human judgment.

The contrast with conventional test runners is instructive. Selenium WebDriver is deterministic by design a test either matches the expected state or fails, with no middle ground. Our framework introduces exactly that middle ground: a confidence-weighted judgment about whether a discrepancy is a regression or an expected change, improving as data accumulates.

The constraints are real and worth naming. Cold-start is a genuine problem limited test history doesn't give the ML components much to learn from early on. The system would lean on similarity-based probabilistic initially and shift weight toward the learned model as history grows. Compute overhead from running inference alongside execution is non-trivial, though modern CI infrastructure typically has headroom.

For projects where test maintenance is already a recognized drain most non-trivial applications past a certain age the trade-off looks favorable. Setup costs are front-loaded; savings compound as the model accumulates variation and becomes more autonomous. The goal isn't to make broken tests pass. It's to make the category of "test broke because the UI changed" as rare and low-effort as possible so that when a test fails, it actually means something.

V. CONCLUSION

We've described a framework that treats test brittleness as a solvable engineering problem rather than an unavoidable cost. The core claim is that combining similarity-based repair, NLP-assisted generation, and reinforcement-learned patch selection can substantially reduce the human time spent keeping test suites current.

Whether that claim still holds or will hold in practice depends on implementation details that this paper does not settle that's what a prototype and a real evaluation would need to address. But the conceptual architecture is, we believe, sound, and the individual components it draws on are well-validated.

A system that learns from its own failures has a quite different relationship to change than one that simply breaks and waits. That difference is worth building toward.

VI. FUTURE WORK

The obvious next step is building a prototype. Even a minimal implementation integrated into one real CI pipeline would let us validate cold-start behavior, measure inference overhead, and stress-test the confidence-scoring logic for patch selection.

On the ML side, the NLP component in particular has room to grow. We used relatively simple models in the conceptual design; transformer-based architectures trained on domain-specific corpora could handle more complex requirement language and produce more accurate test skeletons.

Mobile testing is the other obvious extension. The locator-brittleness problem is just as acute on mobile arguably more so, given how frequently mobile UIs are redesigned but the tooling and the DOM structure are different enough that the self-healing module would need substantial adaptation.

A rigorous empirical evaluation against a controlled baseline is what would make or break the claims here. The key metrics false-failure interception rate, patch accuracy, maintenance time per sprint are measurable. We just need a deployment to measure them against.

Aspect	Traditional Approach	AI-Driven (Proposed)
Accuracy	Rule-based; may miss subtle bugs	Higher accuracy; learns complex patterns
Automation	Semi-automated; manual maintenance	Full automation including self-healing
Scalability	Difficult to scale with large codebases	Scales effectively with updates
Adaptability	Requires manual rule updates	Continuously learns from new defects

ACKNOWLEDGMENT

I would like to express my sincere gratitude to the Computer Engineering Department at Bhagwan Mahavir University (BMU), India, for providing the necessary resources, academic environment, and foundational support required to conduct this research.

I am deeply thankful to my professors, project guides, and faculty members for their invaluable guidance, constructive feedback, and continuous encouragement throughout the conceptualization and development of this framework. Their expertise and insights were instrumental in shaping the ideas presented in this paper.

Furthermore, I would like to extend my appreciation to my peers and colleagues for their engaging discussions and collaborative spirit, which helped refine the technical aspects of this study. Finally, a special thanks to my family for their unwavering moral support and motivation during this research.

REFERENCES

[1] J. Zhang and M. Harman, "Machine learning models for defect severity classification in software systems," 2019.
[2] L. Pan et al., "NLP techniques for automated test scenario generation," 2020.
[3] R. Sharma and S. Kumar, "Predictive analytics in testing effort estimation," 2021.
[4] A. Hamza and T. Jonsson, "Test maintenance costs in agile teams," 2019.
[5] M. Beller et al., "Developer testing behavior," 2019.
[6] M. Leotta et al., "Maintainability of Selenium test suites," 2013.
[7] X. Feng et al., "Traceability in software engineering," 2018.
[8] M. Biagiola et al., "Web test dependency detection," 2019.
[9] M. Hammoudi et al., "WATERFALL approach," 2016.
[10] V. Mnih et al., "Deep reinforcement learning," 2015.
[11] V. Levenshtein, "String similarity techniques," 1966.
[12] A. Stocco et al., "PESTO framework," 2017.
[13] S. Thummalapenta et al., "Automating test automation," 2012.