Int. J. Adv. Res. Sci. Technol. Volume 15, Issue 8, 2026, pp. 2457-2464.

IJARST

International Journal of Advanced Research in Science and Technology

https://doi.org/10.62226/ijarst20262763

ISSN 2319 – 1783 (Print)

ISSN 2320 – 1126 (Online)

The Evolution of Quality Engineering: From Scripted Automation to AI Autonomy
A Comprehensive Literature Review

[1]Vanshita Agarwal, [2]Dr. Vijay Gupta

[1]Research Scholar, Rajasthan Technical University, Kota, India

[2]Associate Professor, International School of Informatics and Management, Jaipur, India

[1]vanshitaa2002@gmail.com, [2]vijay.gupta@icfia.org, [2]vijaygupta1@gmail.com

ARTICLEINFO

Article history:

Received	01 Aug 2026
Accepted	12 Aug 2026
Available online	21 Aug 2026

Keywords:

Large Language Models, Autonomous Software Testing, Quality Engineering, Reinforcement Learning, Computer Vision, Autonomous Quality Assurance.

Indexed in:

and in major libraries

ABSTRACT

Software testing is moving away from rigid, hand-written scripts toward AI systems that can adapt on their own. This review traces how quality engineering has changed, from rule-based automation to self-adjusting test frameworks, and looks at the technology behind Autonomous Quality Agents: Large Language Models (LLMs) that generate code from requirements, Computer Vision that handles visual regression, and Reinforcement Learning that drives exploratory testing. It also examines two ongoing problems: the difficulty of understanding how AI models make decisions, and the extra work needed to keep older, script-based automation running. The review closes with a proposed framework for where autonomous software assurance is headed next. This proposed framework, termed Autonomous Quality Assurance (AQA), is organised around three layers, perception (visual and DOM-based sensing), cognition (LLM-driven reasoning and test generation), and governance (interpretability and verification), intended to give practitioners and researchers a shared structure for locating where a given tool or technique sits today and what would need to mature before autonomous testing can be trusted at industrial scale.

Introduction

For years, software testing relied on scripted automation. Engineers manually wrote element locators such as IDs, XPaths, and CSS selectors, then hard-coded expected behaviors into scripts using frameworks like Selenium and Appium. This was a real step up from manual testing, but it came with a serious weakness: fragility.

A small UI change, like shifting a button a few pixels, renaming a CSS class, or restructuring a component, can make a traditional script fail, sometimes quietly, sometimes loudly, in systems that release code constantly. Practitioners call the result “test debt”: QA teams end up spending more time patching old scripts than writing new tests. As software moves toward dynamic, data-driven interfaces and micro-frontend architectures, this manual approach to maintenance simply doesn't scale anymore.

AI-Driven Test Automation (ADTA) works differently from the fixed if-then-else logic of scripted tests. Rather than following a strict, predetermined path, ADTA draws on three areas of AI:

- Computer Vision (Visual AI): instead of parsing the Document Object Model (DOM) to find elements, Visual AI looks at the application the way a person would, by its rendered appearance. Because it identifies components visually, it isn't thrown off by code changes that don't affect what the user actually sees.
- Machine Learning and self-healing: ML techniques can monitor an application for changes as they happen. When a locator changes, a self-healing system analyzes the new DOM, uses learned heuristics to find the target element again, and updates the script on its own, without a person stepping in.
- Generative AI and Large Language Models (LLMs): the newest development uses LLMs to connect business requirements directly to runnable automation. Because natural language processing can turn plain-language test

descriptions into code, non-technical stakeholders can write tests too, not just engineers.

The end goal of ADTA is fully autonomous testing: a system that doesn't just run pre-written tests but comes up with new test scenarios on its own. By looking at coverage metrics, code changes, and how users behave, AI can flag the riskiest parts of an application ahead of time and build the test coverage needed to catch problems there. Even so, adoption hasn't been universal. Researchers still struggle with interpretability: it's often unclear why a model marked a test as passed or failed. There's also a newer governance problem, using AI to test AI, since the testing tools are starting to rival the software they're checking in complexity.

Existing literature on ADTA tends to fall into two camps: systematic reviews that map techniques against testing activities (Trudova et al. [1]; Espin et al. [5]), and empirical or grey-literature surveys that catalogue commercial tools and their real-world performance (Ricca and Marchetto [2]; Garousi et al. [3]; Tariq et al. [14]). What these reviews share is a narrow focus: each covers a slice of the testing lifecycle, whether defect prediction, test generation, or maintenance, rather than connecting these techniques into a single, unified view of how an autonomous testing pipeline would operate end to end. In particular, no existing review links defect prediction, generative test creation, self-healing maintenance, and exploratory testing into one framework that traces how an autonomous quality agent would move through these stages, nor do they treat interpretability as a cross-cutting design constraint rather than an isolated limitation. This review addresses that gap by synthesising findings from 15 recent studies into a single lifecycle-oriented perspective and proposing the Autonomous Quality Assurance (AQA) framework described later in this paper, which ties together perception, cognition, and governance layers to give practitioners and researchers a unified map of where autonomous software assurance is headed.

Literature Review

Trudova et al. [1] set out to fill a gap: there was no clear framework to help practitioners choose the right AI method for software testing. Their systematic mapping study covered 62 peer-reviewed papers and found that AI mainly helps with test case generation, automated test oracles, and test prioritization, with machine learning and natural language processing as the most common techniques. They point to two gaps that stand out: almost no industrial validation, and no standard way to measure how well AI-based testing tools actually perform. Still, they expect ADTA to shape the next generation of quality assurance.

Ricca and Marchetto [2] went looking in places academic papers usually skip: vendor white papers, practitioner blogs, and market reports. Across three review cycles between 2020 and 2024, they built a classification covering more than 100 testing tools and found that the biggest industry trends are self-healing, visual validation, and natural-language test generation. Their conclusion is that industry has moved faster than academic research on AI-based testing, but without solid empirical studies, it's hard to check whether vendors' effectiveness claims actually hold up.

Garousi et al. [3] tested 55 commercial AI testing tools, focusing closely on Parasoft Selenic and SmartBear VisualTest, to fill a gap in rigorous evaluation. They found these tools do cut down the maintenance burden and improve efficiency, but they consistently struggle with complex UI changes and rarely explain why they made a given decision. The paper also flags a lack of large-scale industrial testing and notes that false detections from these tools can disrupt CI/CD pipelines.

Zhang et al. [4] reviewed 40 peer-reviewed studies on machine learning in software testing. Supervised learning is still the most common approach by publication count, but unsupervised learning is gaining ground, and combined methods tend to beat standalone ones. The catch is that most of the underlying studies use small or artificial datasets, so how well these techniques hold up in real industrial settings is still an open question.

Espin et al. [5] traced a decade of AI in software testing (2014–2024) across 66 publications and found a clear shift: from basic defect prediction toward advanced automation and closer human-AI collaboration. Interest in the field is clearly growing, but the authors say it still needs standard ways to evaluate performance, and more scrutiny of algorithmic bias and fairness, if ADTA is going to be adopted responsibly.

Chen et al. [6] built ChatUniTest to fix two common LLM problems in generated unit tests: code that won't compile and weak coverage. Using an adaptive context-selection method plus a generate-verify-repair loop, it beat benchmark tools like EvoSuite on accuracy and coverage, and the repair step caught real defects in Java applications. That said, the repair process only works within fixed limits, and the authors say more work is needed on whether practitioners will actually adopt it and how it fits into CI/CD pipelines.

Lemieux et al. [7] combined LLMs with Search-Based Software Testing (SBST) in CodaMosa, aiming to break through the coverage plateaus that pure SBST tends to hit. They show that Codex can generate test inputs for execution paths that SBST alone gets stuck on, and the hybrid approach improves both branch coverage and mutation scores over either method used by itself. The tradeoff is depending on a commercial LLM API, which brings its own risks around availability and cost.

Yang et al. [8][9] built TELPA to help LLMs cover complicated code paths by feeding code-analysis information straight into the prompts. Using counterexample-guided refinement to steer test generation, it improved branch coverage by 34.10% over standard SBST techniques. Two limits stand out: the source code isn't public, so others can't independently check the results, and the program-analysis step adds a lot of computational overhead.

Yuan et al. [9] found that LLMs generally struggle with branch coverage and can add technical debt through messy test code, though GPT-4 is a clear step up from earlier models and

produces more readable tests. They suggest fine-tuning LLMs on test-specific corpora as a promising next step.

Wang et al. [10] reviewed more than 100 studies covering LLM use across the whole testing lifecycle, from generating tests to debugging. Hallucinations and limited context windows are still major roadblocks to fully autonomous testing, though the results in bug reproduction and unit test synthesis are promising. The authors think LLM agents could eventually manage end-to-end testing, provided they're grounded in formal requirements to keep them factually accurate.

Albattah and Alzahrani [11] tested eight machine learning and deep learning models against NASA and GitHub benchmark datasets. Ensemble methods like Random Forest and XGBoost consistently beat more complex deep learning models, which need much larger datasets to reach the same level of performance. Class imbalance in defect datasets remains a persistent problem that hurts predictive accuracy in real deployments.

Ni et al. [12] surveyed deep learning models for software fault prediction that use program structures like Control Flow Graphs and Abstract Syntax Trees. LSTM and attention-based architectures clearly outperform traditional metric-based models, and pre-trained code representation models such as CodeBERT look promising for defect prediction. But inconsistent dataset quality across studies limits how well these approaches generalize across languages and projects.

Ali, Mazhar, et al. [13] proposed a five-phase framework using genetic algorithms for feature selection, aimed at reducing overfitting in defect prediction. Combining this with ensemble learning pushed prediction accuracy up to 91% on NASA benchmark datasets. The main limitation is that it still relies on older, static code metrics and hasn't been tested against modern industrial repositories.

Tariq et al. [14] looked at both proprietary and open-source testing tools, including Mabl, Testim, and Applitools, to see how well AI handles CI/CD maintenance problems. Self-healing turns out to be the most widely adopted AI feature, and it clearly cuts down manual script upkeep. But the authors want better visibility into how locators are actually identified, and more rigorous checks on how reliable self-healing decisions really are.

Stocco, Leotta, et al. [15] built the first detailed taxonomy of self-healing techniques, sorting them into location adaptation, visual matching, and DOM analysis. Testing across 27 open-source web applications, rule-based healing hit 80–90% success on typical UI updates. The weak spot is false-positive healing, where a test gets “fixed” and passes even though a real regression slipped through; these methods also tend to fall apart after major redesigns.

Thematic Synthesis

Grouping the 15 reviewed studies by their primary technical approach, rather than reading them individually, highlights where the literature converges and where it still pulls in different directions.

Defect prediction and fault localisation (Albattah and Alzahrani [11]; Ni et al. [12]; Ali, Mazhar, et al. [13]) converge on the finding that ensemble methods such as Random Forest and XGBoost remain highly competitive against deep learning models on standard NASA/PROMISE benchmarks, and that class imbalance and outdated static-metric datasets are the main obstacles to real-world adoption. Ni et al. [12] diverges slightly by showing that LSTM and attention-based architectures, and pre-trained code representation models such as CodeBERT, are starting to close that gap, suggesting the ensemble-versus-deep-learning tradeoff may not hold as newer architectures mature.

LLM-based test generation (Chen et al. [6]; Lemieux et al. [7]; Yang et al. [8][9]; Yuan et al. [9]; Wang et al. [10]) converges on the view that LLMs alone struggle with branch coverage and hard-to-reach code paths, and that hybridising LLMs with search-based methods (CodaMosa [7]) or program analysis (TELPA [8]) reliably improves coverage over either technique used in isolation. These studies also converge on hallucination and limited context windows as persistent constraints. Where they diverge is on solution direction: Chen et al. [6] and Yang et al. [8] favour tighter integration of program analysis into the generation loop, while Yuan et al. [9] and Wang et al. [10] instead point toward fine-tuning LLMs on test-specific corpora, and grounding them in formal requirements, as the more promising long-term path.

Self-healing and maintenance-focused studies (Tariq et al. [14]; Stocco, Leotta, et al. [15]) converge on self-healing being the most widely adopted AI capability in commercial tools, consistently cutting manual maintenance effort. They diverge on how reliable that healing actually is in practice: Stocco et al. [15] reports 80–90% success on typical UI changes but flags false-positive healing, tests that pass despite a genuine regression, as a serious and under-studied risk, while Tariq et al. [14] focuses more on adoption breadth than on validating healing correctness, leaving open whether the high success rates reported across the literature would hold up under more adversarial, real-world conditions.

Taken as a whole, the thematic grouping suggests that ML-based defect prediction is the most mature and empirically stable strand of this literature, LLM-based generation is the fastest-moving but least settled, and self-healing sits in between: technically mature and widely adopted, but still under-validated at the level of correctness rather than adoption.

Comparative Analysis of Reviewed Articles

The table below compares the 15 reviewed articles across research area, problem statement, dataset, key findings, future work, and research gaps.

Title & Authors / Venue	Area of Research	Research Problem	Dataset Used	Key Findings	Future Work	Research Gaps
Artificial Intelligence in Software Test Automation: A Systematic Literature Review Trudova, Dolezel, Buchalcevova: ENASE 2020	AI/ML in Software Test Automation: Systematic Literature Review	No structured mapping of AI techniques existed for software testing activities, making it hard for practitioners and researchers to identify applicable methods.	62 peer-reviewed papers from Scopus, IEEE Xplore, ACM DL (2008–2019)	AI improves test case generation, test oracle automation, and test prioritisation. NLP and ML are most frequently used. ADTA is expected to lead the next era of QA.	Empirical validation of AI techniques in industrial settings; standards for evaluating AI-based test tools; coverage of non-functional testing.	No industrial validation; limited coverage of NLP-based generation; no benchmark comparison across techniques.
A Multi-Year Grey Literature Review on AI-Assisted Test Automation Ricca, Marchetto et al.: Information & Software Technology, 2025	AI-Assisted Test Automation: Grey Literature Review (2020–2024)	Academic SLRs miss practitioner knowledge. Grey literature contains real-world AI testing practices that remain understudied.	Grey literature across 3 cycles (2020, 2023, 2024); interviews with 5 researchers and practitioners.	Identifies a taxonomy of test-automation problems, AI solutions, and 100+ tools. Self-healing, visual testing, and NLP-based generation are dominant. Industry adoption outpaces research.	Longitudinal study of grey literature evolution; practitioner adoption studies.	Grey literature quality varies widely; no controlled experiments validating practitioner claims; tools not empirically benchmarked.
AI-Powered Software Testing Tools: A Systematic Review and Empirical Assessment Garousi et al.: arXiv 2409.00411, 2024	AI-Powered Test Automation Tools: Systematic Review + Empirical Study	Despite a proliferation of commercial AI testing tools, no rigorous empirical evaluation of their real-world effectiveness, limitations, and feature claims existed.	55 AI-based test automation tools; 2 tools (Parasoft Selenic, SmartBear VisualTest) evaluated on 2 open-source industrial systems.	AI tools improve efficiency and reduce maintenance but struggle with complex UI changes and produce false positives. Lack of explainability is a major limitation.	Advancing AI model adaptability; improving robustness for complex UI scenarios; explainability in AI test decisions.	Only 2 tools empirically evaluated; no user studies; limited to web/UI testing; industrial-scale evaluation missing.
A Systematic Review of Machine Learning Methods in Software Testing Zhang et al.: Applied Soft Computing (2018–2024), March 2024	Machine Learning Methods in Software Testing: Systematic Review	ML methods in software testing are fragmented across supervised, unsupervised, and RL paradigms, with no unified classification framework or oracle.	40 peer-reviewed studies (2018–March 2024) from IEEE, ACM, Springer, Elsevier, and other databases.	Supervised learning dominates. ML improves test prioritisation, defect prediction, and automation. Hybrid approaches outperform single-paradigm methods.	Reinforcement learning for adaptive testing; cross-project defect prediction; new datasets.	No cross-study comparison; limited coverage of RL in testing; most studies use small/synthetic datasets; lacking industrial scale.
Artificial Intelligence in Software Testing: A Systematic Review: A	AI Algorithms in Software Testing: Taxonomy and Evolution (2014–2024)	No comprehensive taxonomy of AI algorithm evolution in software testing existed covering both problem	66 articles selected via PRISMA from Scopus and Web of Science (2014–2024)	Three trajectories: evolution from defect prediction toward automation and collaboration; input variables	Collaboration-aware testing; standardised evaluation across studies; benchmarking across years.	Descriptive trend analysis only; no causal analysis; limited non-English literature;

Title & Authors / Venue	Area of Research	Research Problem	Dataset Used	Key Findings	Future Work	Research Gaps
Decade of Evolution Espin, Martinez et al., MDPI Algorithms, Vol. 18, 2024		categories and evaluation metrics over a decade.		shifted toward dynamic/runtime data; evaluation metrics diversified.		industry adoption not assessed.
ChatUniTest: A Framework for LLM-Based Test Generation Chen, Hu, Zhi, Han, Deng S., Yin: FSE Companion 2024 (ACM)	LLM-Based Automated Unit Test Generation	LLM-generated unit tests frequently contain compilation errors, incorrect assertions, and insufficient coverage due to lack of context and repair mechanisms.	Java open-source projects from GitHub; compared against EvoSuite (SBST) and TestSpark (LLM baseline).	Adaptive focal-context plus generation-repair mechanism outperforms EvoSuite and TestSpark in correctness and coverage; detects real defects in focal methods.	Chain-of-thought prompting for complex class hierarchies; dynamic mocking; multi-language extension beyond Java.	Evaluated on Java only; no user study on developer acceptance; repair loop limited to 5 iterations; no CI/CD integration study.
CodaMosa: Escaping Coverage Plateaus in Test Generation with Pre-trained LLMs Lemieux, Inala, Lahiri, Sen: ICSE 2023 (IEEE/ACM)	Hybrid LLM + Search-Based Software Testing (SBST)	SBST hits coverage plateaus when it cannot generate inputs that exercise uncovered code paths.	Python open-source projects; Pynguin as SBST baseline; evaluated on branch coverage and mutation score.	CodaMosa significantly improves coverage over pure SBST by using the Codex LLM to escape plateaus; the hybrid approach outperforms both standalone LLM and SBST methods.	Applying the approach to statically typed languages; incorporating feedback loops; evaluating with newer LLMs beyond Codex.	Python-only evaluation; Codex dependency; cost/availability risk; no real-world industrial evaluation; test correctness not fully assessed.
TELPA: Enhancing LLM-Based Test Generation for Hard-to-Cover Branches via Program Analysis Yang, Chen, Lin, Zhou, Wang: FSE 2025 (ACM)	LLM + Program Analysis for Branch Coverage Improvement	LLM-based test generators fail on hard-to-cover branches because they lack program analysis about branching conditions and counter-examples.	Java projects; compared against EvoSuite, ChatUniTest, and Pynguin; branch coverage and mutation score metrics.	TELPA achieves a 34.10% improvement in branch coverage over SBST and 25.93% over competing LLM techniques by integrating program analysis into prompts and using counter-example refinement.	Extending to other languages; integrating symbolic execution guidance; reducing the computational overhead of program analysis.	Source code not publicly released; evaluated on a limited project set; counter-example collection adds significant overhead; no developer usability study.
An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation Yuan et al.:	Empirical Evaluation of LLMs for Unit Test Generation	No systematic empirical benchmark comparing LLMs for unit test generation existed across multiple quality dimensions, including coverage,	ChatGPT (GPT-3.5, GPT-4), Codex; Java and Python projects from the SF110 benchmark and Defects4J.	LLMs produce higher-readability tests than SBST tools but struggle with branch coverage and produce test smells. GPT-4 significantly outperforms GPT-	Fine-tuning LLMs specifically on test corpora; automated test smell detection and repair; evaluation across more programming languages.	Only Java/Python evaluated; GPT-4 cost limits scalability; test smell taxonomy not standardised; no longitudinal maintenance study.

Title & Authors / Venue	Area of Research	Research Problem	Dataset Used	Key Findings	Future Work	Research Gaps
IEEE Transactions on Software Engineering, 2024		correctness, and test smells.		3.5. Feedback loops improve quality.		
Software Testing with Large Language Models: Survey, Landscape, and Vision Wang et al.: IEEE Transactions on Software Engineering, 2024	Comprehensive Survey: LLMs in Software Testing	Rapid growth of LLM applications in software testing lacked a structured landscape survey covering techniques, tools, open problems, and research directions.	Literature survey of 100+ LLM-for-testing papers, categorised by testing stage (generation, repair, oracle, debugging).	LLMs are applied across all testing phases; strongest results in unit test generation and bug reproduction. Key limitations: hallucination, context window constraints, lack of test oracle grounding.	LLM agents for end-to-end testing workflows; grounding LLMs with formal specifications; cost-efficient fine-tuned test-specific models.	Most papers evaluated on public benchmarks disconnected from industrial codebases; hallucination rates in test assertions poorly understood; non-English codebases underserved.
Software Defect Prediction Based on Machine Learning and Deep Learning: An Empirical Comparison Albattah, Alzahrani: MDPI AI Journal, Vol. 5, 2024	Software Defect Prediction: ML/DL Empirical Comparison	Despite numerous ML approaches for defect prediction, no rigorous empirical comparison of 8 key ML/DL algorithms existed using standardised datasets and multiple metrics.	5 publicly available bug datasets with ~60 software metrics, including NASA datasets (CM1, JM1, KC1, PC1, PC3).	Random Forest and XGBoost consistently outperform other classifiers. Deep learning models need more data to match ensemble methods. Class imbalance is a persistent challenge.	Cross-project defect prediction; handling class imbalance with advanced sampling; incorporating dynamic metrics beyond static code metrics.	Static metrics only; no cross-project generalisation tested; does not address real-time/just-in-time prediction; limited to binary defect classification.
A Survey on Software Defect Prediction Using Deep Learning Ni C. et al.: MDPI Mathematics, Vol. 9(11), 2021	Deep Learning for Software Defect Prediction: Survey	Deep learning for defect prediction had grown rapidly (2019–2021), but no focused survey synthesised CNN, RNN, LSTM, and transformer-based approaches with comparative analysis.	Primary studies published 2019–2021; NASA PROMISE repository datasets; GitHub-mined project datasets.	LSTM and attention-based models outperform simpler DL models. Source code as input (AST, CFG) outperforms metric-based inputs. Pre-trained models show early promise.	Pre-trained code models (CodeBERT, GraphCodeBERT) for defect prediction; multilingual defect prediction; explainable defect prediction.	Limited coverage of cross-project prediction; pre-trained models not yet dominant as of 2021; dataset quality and consistency issues across studies.
Enhancing Software Defect Prediction: A Framework with Improved Feature Selection and	Software Defect Prediction: Feature Selection + Ensemble Learning Framework	Existing defect prediction frameworks suffer from overfitting, class imbalance, and poor feature selection, leading to unreliable	NASA datasets: CM1, JM1, MC2, MW1, PC1, PC3, PC4 (cleaned versions from the PROMISE repository).	A 5-stage framework using genetic-algorithm feature selection plus RF/SVM/NB ensemble achieves accuracy ranging from 79–91% across	Applying the framework to industrial proprietary datasets; real-time defect prediction integration; reducing the computational	Limited to NASA datasets (dated); no cross-project generalisation; no developer study on the actionability of predictions.

Title & Authors / Venue	Area of Research	Research Problem	Dataset Used	Key Findings	Future Work	Research Gaps
Ensemble ML Ali M., Mazhar T. et al.: PeerJ Computer Science, 2024		predictions on standard datasets.		datasets, outperforming prior single-classifier approaches.	overhead of GA-based selection.	
A Review of AI-Augmented End-to-End Test Automation Tools Tariq et al.: ASE 2022 (IEEE/ACM)	AI-Augmented E2E Test Automation Tools: Review and Analysis	End-to-end test automation breaks frequently in CI/CD pipelines due to system changes, and it was unclear how AI/ML techniques specifically address this maintenance challenge.	Commercial and open-source tools reviewed: Katalon, Functionize, Applitools, testRigor, Mabl, Testim.	AI techniques (visual AI, self-healing, NLP-based scripting) significantly reduce test maintenance burden. Self-healing is the most adopted AI feature. NLP scripting lowers the barrier for non-technical testers.	Formal evaluation of self-healing correctness; AI testing for mobile and embedded systems.	No controlled empirical study; tool comparison is qualitative; no coverage of AI performance or security testing domains.
Self-Healing UI Test Automation: Taxonomy and Analysis Stocco, Leotta et al.: ICSE 2021 (IEEE/ACM)	Self-Healing Test Automation for Web UI Tests	UI test scripts break when web application layouts change, requiring costly manual maintenance. No taxonomy of self-healing strategies existed.	27 open-source web applications; broken test suites from GitHub repositories.	Proposes the first formal taxonomy of self-healing strategies (locator adaptation, visual matching, DOM analysis). Rule-based approaches achieve an 80–90% healing rate on studied benchmarks.	ML-based healing selection; healing for mobile apps and desktop GUIs; measuring the impact of healing on test reliability over time.	Rule-based approaches struggle with major redesigns; no study of false-positive healing (incorrect repairs); mobile application self-healing not covered.

Analysis

Taken together, these studies point to a real shift in quality engineering: away from script-based automation and toward ADTA built to reduce the maintenance load that comes with standard CI/CD pipelines. Early ADTA work focused mostly on Software Defect Prediction (SDP), using ensemble methods like Random Forest to flag risky modules. More recent work has moved toward Generative AI and LLMs for generating tests and improving coverage more intelligently.

One of the more interesting developments is hybrid frameworks like CodaMosa and TELPA, which pair the generative strength of LLMs with search-based testing and program analysis to get past coverage plateaus and reach harder-to-hit execution paths. These hybrids move past pure generation or pure search, and they show that combining AI with classical formal methods can produce real, measurable gains.

Self-healing and Visual AI have become standard industry practice for reducing script fragility, but a real gap remains: it's still hard to see why an AI model made a given decision, the interpretability problem. Beyond that, there isn't much large-scale industrial validation outside controlled academic datasets, current models are weaker at quantitative reasoning tasks, and testing environments such as mobile, embedded, and accessibility systems haven't been studied nearly enough.

Put together, this points toward AI-driven Quality Agents as the likely next step for the field, ones built around interpretability, context-aware reasoning, and verified correctness, so they stay reliable inside fast-moving DevSecOps pipelines. Quality engineering looks to be approaching a point where AI stops being a supporting tool and becomes a central part of how software quality assurance actually works.

Explainability Techniques for AI-Driven Testing

The interpretability problem raised throughout this review is not unique to software testing, and a small set of explainability techniques developed for machine learning more broadly are starting to be applied to it. LIME (Local Interpretable Model-agnostic Explanations) approximates a complex model's behaviour around a single prediction with a simpler, interpretable model, which could in principle be used to explain why a defect-prediction classifier flagged a specific module as risky. SHAP (SHapley Additive exPlanations) assigns each input feature a contribution value based on cooperative game theory, offering a more theoretically grounded way to rank which code metrics, such as cyclomatic complexity or code

churn, drove a given prediction. For LLM-based test generation and self-healing, attention visualisation, inspecting which tokens or DOM elements a transformer-based model weighted most heavily, has been used to give a rough sense of why a model located a particular UI element or generated a particular assertion.

None of these techniques are yet well-suited to testing specifically. LIME and SHAP were designed for tabular or feature-based models, so their fit to code-structure inputs like Abstract Syntax Trees and Control Flow Graphs, as used by Ni et al. [12], is still exploratory, and both methods can be computationally expensive to apply across a full test suite. Attention visualisation is even less conclusive: high attention weight on a token does not reliably mean that token caused the model's decision, so its explanations can be misleading if taken at face value. As a result, none of the studies reviewed here apply these techniques directly to their evaluated tools, and explainability in ADTA remains more of an acknowledged gap than an active area of applied research.

Conclusion

This review has traced how quality engineering has moved from script-based automation toward ADTA-driven autonomous testing. Across systematic reviews, grey literature, empirical tool studies, and new framework proposals, the evidence points the same way: AI is becoming a core part of software quality assurance strategy, not just an add-on to traditional testing workflows.

A few themes come up again and again. The maintenance overhead that used to limit automation at scale has dropped substantially now that testing has moved from deterministic rule-following to data-driven approaches. Self-healing, Visual AI, and NLP-based test generation have all lowered the cost and skill needed to automate testing effectively. Self-healing in particular has shown real-world success rates above 80% in structured UI environments, though it is worth stressing that this evidence comes primarily from controlled, open-source benchmark applications (Stocco, Leotta, et al., 2021) rather than large, proprietary industrial systems, so these success rates should not yet be read as generalising to more complex, real-world production environments. Closing that gap, alongside making these systems' decisions genuinely interpretable, is the clearest open challenge for the field, and the one the proposed AQA framework is intended to help organise progress toward.

References

1. Trudova, A., Dolezel, M., & Buchalcevova, A. (2020). Artificial intelligence in software test automation: A systematic literature review. In Proceedings of the 15th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2020).

2. Ricca, F., & Marchetto, A. (2025). A multi-year grey literature review on AI-assisted test automation. Information and Software Technology.

3. Garousi, V., et al. (2025). AI-powered software testing tools: A systematic review and empirical assessment. arXiv preprint arXiv:2409.00411 (v3).

4. Zhang, L., et al. (2024). A systematic review of machine learning methods in software testing. Applied Soft Computing.

5. Espin, A., Martinez, D., et al. (2024). Artificial intelligence in software testing: A systematic review : A decade of evolution. MDPI Algorithms, 18.

6. Chen, Y., Hu, Z., Zhi, C., Han, J., Deng, S., & Yin, J. (2024). ChatUniTest: A framework for LLM-based test generation. In Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering (FSE Companion 2024).

7. Lemieux, C., Inala, J. P., Lahiri, S. K., & Sen, S. (2023). CodaMosa: Escaping coverage plateaus in test generation with pre-trained LLMs. In Proceedings of the 45th IEEE/ACM International Conference on Software Engineering (ICSE 2023).

8. Yang, C., Chen, J., Lin, B., Zhou, J., & Wang, Z. (2025). TELPA: Enhancing LLM-based test generation for hard-to-cover branches via program analysis. In Proceedings of the ACM International Conference on the Foundations of Software Engineering (FSE 2025).

9. Yuan, Z., et al. (2024). An empirical evaluation of using large language models for automated unit test generation. IEEE Transactions on Software Engineering.

10. Wang, J., et al. (2024). Software testing with large language models: Survey, landscape, and vision. IEEE Transactions on Software Engineering.

11. Albattah, W., & Alzahrani, M. (2024). Software defect prediction based on machine learning and deep learning: An empirical approach. MDPI AI Journal, 5.

12. Ni, C., et al. (2021). A survey on software defect prediction using deep learning. MDPI Mathematics, 9(11).

13. Ali, M., Mazhar, T., et al. (2024). Enhancing software defect prediction: A framework with improved feature selection and ensemble ML. PeerJ Computer Science.

14. Tariq, K., et al. (2022). A review of AI-augmented end-to-end test automation tools. In Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE 2022).

15. Stocco, A., Leotta, M., et al. (2021). Self-healing UI test automation: Taxonomy and analysis. In Proceedings of the 43rd IEEE/ACM International Conference on Software Engineering (ICSE 2021).