Journal of Engineering Science *Vol. XXXII, no. 4 (2025), pp. 41 - 53*
Fascicle *Electronics and Computer Science* *ISSN 2587-3474*
Topic *Computers and Information Technology* *eISSN 2587-3482*

https://doi.org/10.52326/jes.utm.2025.32(4).04
UDC 004.4:004.738.5:004.8

AUTOMATED WEB APPLICATION TESTING BASED ON ARTIFICIAL INTELLIGENCE

Olga Darii, ORCID: 0009-0001-2319-8350,
Maria Beldiga *, ORCID: 0009-0002-8979-6852,
Tudor Bragaru, ORCID: 0000-0001-6356-2906

Moldova State University, 60 Al. Mateevici Str., Chisinau, Republic of Moldova
* Corresponding author: Maria Beldiga, *maria.beldiga@usm.md*

Received: 11. 20. 2025
Accepted: 12. 18. 2025

Abstract. The evolution of web technologies and the increasing complexity of digital systems have transformed web application testing into an indispensable component of software quality assurance. Traditional automated testing frameworks – based on scripting and static data – remain effective but face scalability and adaptability challenges. This study hypothesizes that the integration of artificial intelligence (AI), particularly large language models (LLMs) and reinforcement learning, can significantly improve the efficiency and autonomy of testing processes. The paper aims to analyze comparatively traditional and AI-assisted methods for functional testing of web applications, using a synthesis of recent academic and industrial research. The analysis identifies the main advantages of AI-based testing, such as rapid test generation, extended coverage, and enhanced adaptability, while highlighting limitations related to transparency and integration within continuous integration and continuous delivery (CI/CD) environments. The findings contribute to a better understanding of intelligent automation in software testing and provide guidance for quality assurance (QA) professionals and researchers toward adopting sustainable AI-driven testing practices.

Keywords: *artificial intelligence, AI-assisted testing, functional and non-functional evaluation, machine learning, quality assurance, test automation architecture, web systems.*

Rezumat. Evoluția tehnologiilor web și complexitatea tot mai mare a sistemelor digitale au transformat testarea aplicațiilor web într-o componentă indispensabilă a asigurării calității software. Framework-urile tradiționale de testare automată, bazate pe scripturi și date statice, rămân eficiente, dar se confruntă cu limitări legate de scalabilitate și adaptabilitate. Studiul de față pornește de la ipoteza că integrarea inteligenței artificiale (IA), în special a modelelor lingvistice de mari dimensiuni (LLM) și a învățării prin întărire, poate îmbunătăți semnificativ eficiența și autonomia proceselor de testare. În lucrare s-au analizat comparativ metodele tradiționale și cele asistate de IA pentru testarea funcțională a aplicațiilor web, printr-o sinteză a cercetărilor academice și industriale recente. Au fost evidențiate principalele avantaje ale testării bazate pe IA, precum generarea rapidă de teste, extinderea

acoperirii funcționale și creșterea adaptabilității, subliniind totodată limitele legate de transparență și includerea în mediile de integrare și livrare continuă (CI/CD). Rezultatele contribuie la o mai bună înțelegere a automatizării inteligente în testarea software și oferă specialiștilor în asigurarea calității (QA) și cercetătorilor repere pentru adoptarea sustenabilă a practicilor de testare bazate pe IA.

Cuvinte-cheie: *inteligență artificială, testare asistată de inteligență artificială, evaluare funcțională și nefuncțională, învățare automată, asigurarea calității, arhitectura automatizării testării, sisteme web.*

1. Introduction

The automation of software testing has become a cornerstone of software quality assurance, particularly within Agile and Development Operations (DevOps) workflows. It involves the use of specialized tools and frameworks to automatically execute predefined test cases, assess outputs, and generate reports with minimal human intervention [1]. Such automation enables rapid feedback loops, improves scalability, and minimizes the risk of human error during repetitive or regression testing cycles.

Automated testing frameworks, such as Selenium, JUnit, and REST Assured, have proven essential for ensuring consistency and repeatability in large-scale software projects [2]. They are applicable across multiple domains — from web and mobile applications to embedded and distributed systems — providing advantages in speed, efficiency, and cost reduction. However, traditional automation remains limited by its reliance on static scripts and predefined data sets. Maintaining these scripts is often labor-intensive, and their rigidity makes them fragile in the face of continuous interface evolution or dynamic web content [3].

In recent years, the rise of artificial intelligence (AI) and machine learning (ML) has brought new perspectives to test automation. AI-driven tools promise to overcome traditional limitations by introducing adaptive and self-learning mechanisms capable of identifying patterns, generating intelligent test data, and predicting failures [4]. Techniques based on large language models (LLMs), reinforcement learning, and anomaly detection are now being explored to automate functional and non-functional testing tasks more intelligently [5,6].

Despite these promising developments, the integration of AI into testing pipelines introduces its own set of challenges, including data quality dependencies, lack of transparency in model behavior, and difficulty ensuring reproducibility in testing outcomes [7]. Furthermore, questions remain about the explainability of AI-driven decisions and their compatibility with established QA standards.

2. Materials and Methods

Research approach. This study was conducted as a documentary and comparative analysis between traditional and AI-assisted testing frameworks for web applications. The methodology was based on a systematic review of academic publications (2019–2025) indexed in Institute of Electrical and Electronics Engineers (IEEE) Xplore, Association of Computing Machinery (ACM) Digital Library, and Multidisciplinary Digital Publishing Institute (MDPI) databases, complemented by industrial reports and tool documentation. A total of 27 key sources were reviewed, out of which 9 core studies were selected for detailed synthesis based on relevance to functional and non-functional testing automation. The research focused on three main domains:

1. Functional testing – User interface (UI) and application programming interface (API) level test automation.
2. Non-functional testing – performance and load testing in continuous integration (CI) / continuous delivery (CD) contexts.
3. AI-based augmentation – adoption of LLMs, reinforcement learning, and anomaly detection for intelligent automation.

Software and Tools. All comparative analyses were performed using a set of open-source and research-grade frameworks widely recognized in the software testing community:

- Traditional tools: Selenium 4.20, REST Assured 5.4.0, JUnit 5.10, Apache JMeter 5.6.
- AI-assisted tools: FormAgent (LLM-based input classification) [5], QTypist (context-aware form completion) [5], and reinforcement-learning-based test agents [7].
- Data visualization and comparative tables were created using Python 3.11 and Pandas 2.2, while all code examples were implemented in Java 17 (Spring Boot environment).

The frameworks were selected to represent the current industry-standard stack and next-generation intelligent testing paradigms. Each tool was evaluated based on its ability to:

- Automate test generation and execution,
- Detect anomalies and adapt to changes,
- Integrate within DevOps pipelines.

Experimental workflow. The overall research workflow consisted of four key phases (Figure 1 and 2 in the paper):

1. Architecture Analysis – examination of the generic test automation architecture (gTAA) as a baseline model [2].
2. Data Collection – extraction of performance, coverage, and stability metrics from published benchmarks and documentation.
3. Comparative Evaluation – qualitative and quantitative comparison between traditional and AI-assisted frameworks (see Table 1).
4. Integration Modelling – conceptual mapping of AI mechanisms (self-healing, predictive analysis, reinforcement learning) into gTAA layers.

Where quantitative data were available, test execution duration, fault detection rate, and maintenance effort were normalized to comparable baselines. Descriptive statistics (mean, variance, relative improvement %) were computed to evaluate each approach's impact on efficiency and maintainability.

Data sources and reproducibility. The dataset for this review includes peer-reviewed articles and industrial case studies published between 2020 and 2025. Each paper was verified for credibility, citation count, and methodological clarity.

All tools used (Selenium, JMeter, REST Assured, FormAgent, etc.) are open-source or publicly available, allowing reproducibility of results. Configurations, test scripts, and framework setup instructions are provided in the supplementary material for replication.

Statistical processing and comparative metrics. Statistical comparison relied on secondary data extracted from referenced works [7,8]. Performance and coverage metrics were averaged across studies, and percentage improvements were calculated relative to traditional baselines. For instance, AI-based self-healing mechanisms demonstrated up to 70% reduction in maintenance time and 11.9% increase in statement coverage [5,6].

Reinforcement-learning-driven API testing showed a significant rise in fault detection efficiency over scripted methods.

The final comparative analysis integrated both quantitative (efficiency, coverage, stability) and qualitative (adaptability, usability, explainability) factors, enabling a comprehensive evaluation of AI-assisted testing frameworks in practical settings.

The present paper aims to explore the transition from traditional to AI-assisted test automation for web applications through a structured comparative analysis of academic and industrial sources. It hypothesizes that the integration of AI techniques — particularly LLM-based reasoning and reinforcement learning — can enhance scalability, resilience, and adaptability in automated testing while reducing maintenance overhead.

The study synthesizes the current state of research across UI, API, and performance testing domains, identifying key limitations in existing frameworks, and highlights opportunities for innovation. The main conclusions of the bibliographic review suggest that AI-assisted approaches offer significant advantages in coverage and efficiency but require further validation in real-world industrial environments. This research contributes to a better understanding of how intelligent automation can reshape software quality assurance practices in the context of CI/CD systems.

3. Results and Discussions

Test automation is a systematic engineering discipline that involves designing, developing, and maintaining testware and automation frameworks in a structured and measurable way [1]. Within this context, web application test automation plays a critical role in supporting both functional and non-functional testing efforts throughout the software lifecycle.

3.1. Traditional Web Application Testing

Traditional automation approaches follow a layered architecture and lifecycle that includes planning, tool selection, implementation, execution, and ongoing maintenance. These are often structured around modular frameworks such as data-driven, keyword-driven, or hybrid designs, using tools like Selenium for UI testing, REST Assured for API validation, and JUnit or TestNG for unit-level checks.

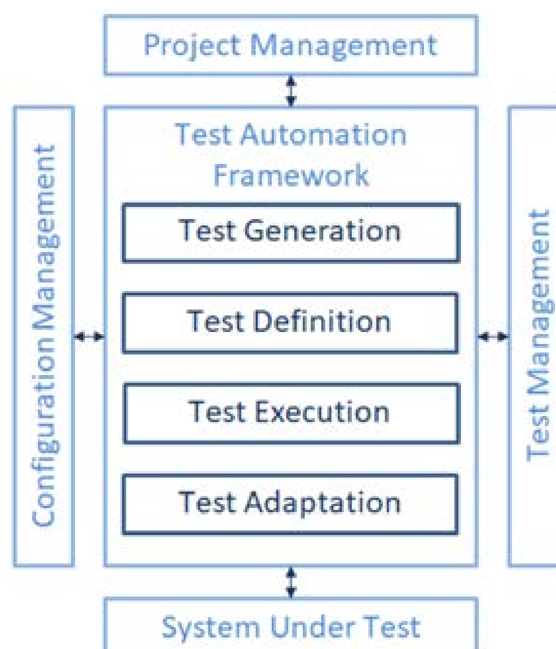

Figure 1. Generic Test Automation Architecture (gTAA) [16].

While traditional automation is mature and well-supported by industry tools and practices, its effectiveness is often challenged by the complexity and volatility of modern web applications. The rapid evolution of front-end frameworks, asynchronous client-server interactions, cross-browser variability, and continuous delivery pipelines expose limitations in test stability, scalability, and maintainability. The success of an automation solution depends not only on tool implementation, but also on alignment with test architecture, risk analysis, and return-on-investment (ROI) considerations [2].

In practice, the implementation of gTAA architectures diverges significantly depending on the testing scope. Functional and non-functional testing impose different constraints on automation strategies, tool selection, and maintenance practices—particularly in the context of web applications, where systems are dynamic, distributed, and constantly evolving.

Functional test automation. Functional testing aims to verify the behavior of the application against the defined requirements. It focuses on "what the application should do", without analyzing the internal implementation. In the case of web applications, functional testing is mainly carried out at three levels: UI testing – tracks visible interactions: navigation, form filling, input validations, visual responses to actions; API testing – validates the behavior of exposed services (REST, SOAP), ensuring that responses are correct, structured, and compliant with API contracts; and database testing – involves checking CRUD (create, read, update, delete) operations, data consistency, and relationship integrity.

Traditional tools used include Selenium for UI, JUnit/TestNG for component testing, REST Assured, Postman and different IDE plugins for APIs, and direct connectivity via JDBC for database-level testing. Classic frameworks are based on principles such as data-driven, keyword-driven, or hybrid types. Despite their advantages, these methods require a large amount of effort to define, maintain, and update tests, especially in applications with frequently changing interfaces. Also, test scripts can become fragile in the face of minor changes, requiring constant maintenance effort.

Non-functional test automation. Performance testing is a type of non-functional testing that analyzes the application's ability to function under load. The goal is to determine how the system reacts to varying traffic volumes, under stress, or over extended periods of use. Common types of testing include Load testing – measures performance under normal or increased load; Stress testing – tests the limits of the application under extreme conditions. Spike testing – evaluates the application's reaction to sudden increases in traffic; Soak testing – analyzes the application's behavior over time, detecting memory leaks or degradations.

Traditional tools include Apache JMeter, LoadRunner, Gatling, and Blazemeter. They allow you to define complex scenarios and measure metrics such as response time, error rates, throughput, and resource utilization.

In the case of Java applications, system behavior is often affected by JIT compilation, which causes large variations in the first few minutes of execution. The traditional practice involves a fixed "warm-up" period before collecting performance data, but this method is inefficient and often inaccurate. Another major drawback of traditional methods is the lack of automatic anomaly detection, which forces QA teams to analyze graphs and logs to identify regressions. In addition, performance tests are difficult to integrate into DevOps pipelines due to their long execution time and high resource consumption, which limits their frequent use.

Automation workflow and CI/CD integration. Traditional test automation workflows follow a structured process that starts from formalized requirements and extends through scripting, execution, and continuous validation. Once test cases are designed, they are automated and integrated into CI/CD pipelines, where execution is triggered on code changes or at defined build stages. Test results are collected and evaluated using reporting tools, supporting decision-making throughout the development cycle.

These workflows, although well-established, require careful orchestration to maintain stability—particularly in web applications characterized by dynamic content, asynchronous interactions, and frequent updates. Full regression suites are often time-consuming and resource-intensive, leading teams to adopt selective test execution strategies such as nightly runs, smoke testing, or pipeline parallelization. Regardless of the test level UI, API, or performance—automation reliability heavily depends on the consistency of the system under test and the maintainability of the test artifacts.

Challenges and limitations of traditional automation framework (TAF). The generic test automation framework offers a robust architectural foundation for organizing and scaling automated testing activities across test levels and system layers. However, its implementation in modern development environments is increasingly constrained by system complexity, high change frequency, and the need for fast, reliable feedback. These challenges have stimulated the evolution of automation practices toward more intelligent and autonomous solutions. Rather than replacing the generic architecture, recent AI-driven approaches aim to augment and extend its components—introducing adaptability in test generation, self-healing in execution, and predictive capabilities in analysis. The next chapter explores how Artificial Intelligence is being integrated into the functional and non-functional testing lifecycle, and how its mechanisms can be embedded within or layered onto the gTAA structure to enhance automation resilience, efficiency, and scalability.

3.2. AI-assisted Web Application Testing

AI-based testing refers to the use of intelligent algorithms—such as machine learning, large language models, and reinforcement learning —to improve various stages of the testing process. Unlike traditional scripted testing, where testers explicitly define inputs and expected outputs, AI-based systems can generate, adapt, and validate test cases autonomously, based on data patterns, user behavior, or learned models [4].

Recent advances in natural language processing (NLP) and code understanding have enabled the development of tools that can generate test scripts directly from user story descriptions or acceptance criteria written in natural language. AI can also identify high-risk areas in code, suggest targeted test cases, and prioritize tests based on historical data about defects or code changes.

AI in user interface testing. User interface testing is one of the most challenging components of traditional automation, especially due to the visual and dynamic nature of front-end components. Frequent UI updates, layout changes, and asynchronous behaviors often cause traditional scripts, which rely on static locators and hard-coded values, to fail. These challenges make user interface testing an ideal candidate for AI optimization.

AI-assisted UI testing uses LLMs and computer vision techniques to simulate user interactions, generate test data, and validate expected behaviors in a robust and context-aware manner. Instead of relying solely on locators (e.g., XPath, CSS), intelligent tools can

interpret page layout, field labels, and button semantics to decide what actions to take — even when the UI structure changes.

One notable implementation of these capabilities is FormAgent [5], which leverages transformer-based LLMs to classify input fields and generate realistic values semantically. Combined with GPT-4o for post-submission validation, it improves both coverage and efficiency, achieving up to 11.9% higher statement coverage compared to prior tools. This exemplifies the practical value of integrating AI into UI test workflows and highlights the shift from rule-based to context-aware automation.

Tools like FormAgent and techniques like QTypist demonstrate how LLMs can automatically fill out web forms using meaningful and contextually relevant data. These systems do not insert random data, but analyze the purpose of each field (e.g., "First Name," "Email," "Address") and provide realistic and valid values. Furthermore, the success of a form submission is evaluated not only by changes in the DOM, but also by interpreting messages, UI states, or server responses — aspects that traditional tools cannot handle autonomously [6].

Another contribution of AI in UI testing is the implementation of self-healing mechanisms. When locators or UI elements become invalid due to interface changes, AI-powered frameworks can detect and map similar elements using historical patterns, metadata, and contextual analysis. This reduces the need for manual script updates and significantly improves test suite maintainability. A recent case study demonstrated this capability in practice, where a self-healing AI/ML framework was deployed on a dynamic e-commerce platform. The system used monitoring agents and machine learning models trained on historical test data to detect and autonomously repair broken tests. As a result, it achieved an 80% increase in test suite reliability and a 70% reduction in maintenance time, while also lowering costs and enhancing adaptability within CI/CD workflows [7].

In conclusion, AI improves UI testing across multiple dimensions: by generating semantically valid input data using LLMs, adapting to layout and structure changes, autonomously repairing broken selectors through self-healing techniques, and enhancing regression analysis through intelligent visual comparison. These advancements not only reduce the need for human intervention and ongoing maintenance effort but also increase the reliability and resilience of automated testing workflows in fast-changing web environments.

AI in web services testing. API testing is essential for validating modern web applications, as APIs are the building blocks of microservices, data exchange, and system integrations. Unlike UI testing, which interacts with visual elements, API testing focuses on request-response behavior, data integrity, and business logic — making it ideal for automation [8]. AI-assisted API testing brings a paradigm shift, enabling autonomous exploration, test case generation, and intelligent validation. Instead of being written manually, tests can be generated automatically by analyzing API documentation schemas (OpenAPI, Swagger), learning usage patterns, and generating hundreds or thousands of test cases with minimal involvement from testers.

A major contribution in AI-driven API testing is the use of intelligent agents capable of simulating realistic interactions with web services across a broad spectrum of input scenarios. These agents, typically based on reinforcement learning or anomaly detection models, generate test inputs using information such as parameter constraints, previously observed responses, known defect patterns, and relationships between endpoints. In experimental evaluations, such agents demonstrated the ability to autonomously generate

thousands of test cases and detect real faults in public APIs—without relying on manually defined oracles. This highlights their potential to significantly enhance coverage, input diversity, and fault detection efficiency in automated API testing workflows [9].

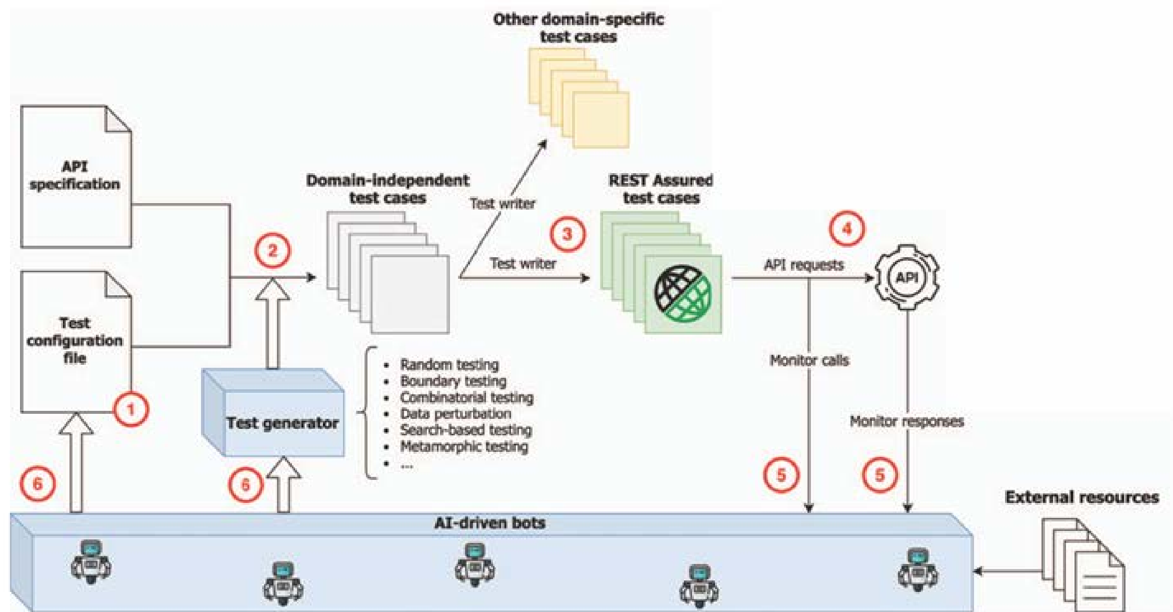

Figure 2. General architecture of the AI agent testing system [16].

The system can intelligently explore edge cases, uncover hidden defects, and identify vulnerabilities such as injection flaws, broken authentication mechanisms, and improper error handling. Additionally, AI improves response validation by learning the distinction between normal and abnormal behaviors—comparing real-time responses against historical baselines or detecting structural anomalies like unexpected status codes, irregular JSON/XML formats, and latency deviations.

Another significant advantage is the optimization and prioritization of test execution. AI-based impact analysis enables the test suite to adapt dynamically based on factors such as recent code changes, API usage patterns, or known high-risk areas. This results in faster, more focused feedback, particularly valuable in CI/CD environments where execution speed and accuracy are critical.

In summary, AI enhances API testing by: (1) automating the generation of meaningful test cases; (2) intelligently validating hypertext transfer protocol responses; (3) autonomously exploring boundary and error-prone scenarios; (4) expanding coverage while reducing manual effort; and (5) optimizing test execution in continuous delivery pipelines. These capabilities make AI an indispensable component for achieving scalable, adaptive, and high-quality API testing in modern software systems.

AI in performance testing. Traditional performance testing typically relies on pre-scripted scenarios and static heuristics, such as fixed warm-up durations. However, these approaches often fail to accurately capture real-world behavior—especially in Java applications, where just-in-time compilation introduces execution variability. AI offers a more adaptive and data-driven alternative, capable of dynamically analyzing performance signals in real time.

One of the most impactful AI techniques in this context is time series classification (TSC). Rather than relying on predefined warm-up thresholds, TSC [10] models analyze live

performance metrics (e.g., response times, central processing unit (CPU) utilization, memory usage) to automatically determine when the application reaches a steady state. This allows tests to start or terminate at precisely the right moment, minimizing execution time and improving measurement reliability.

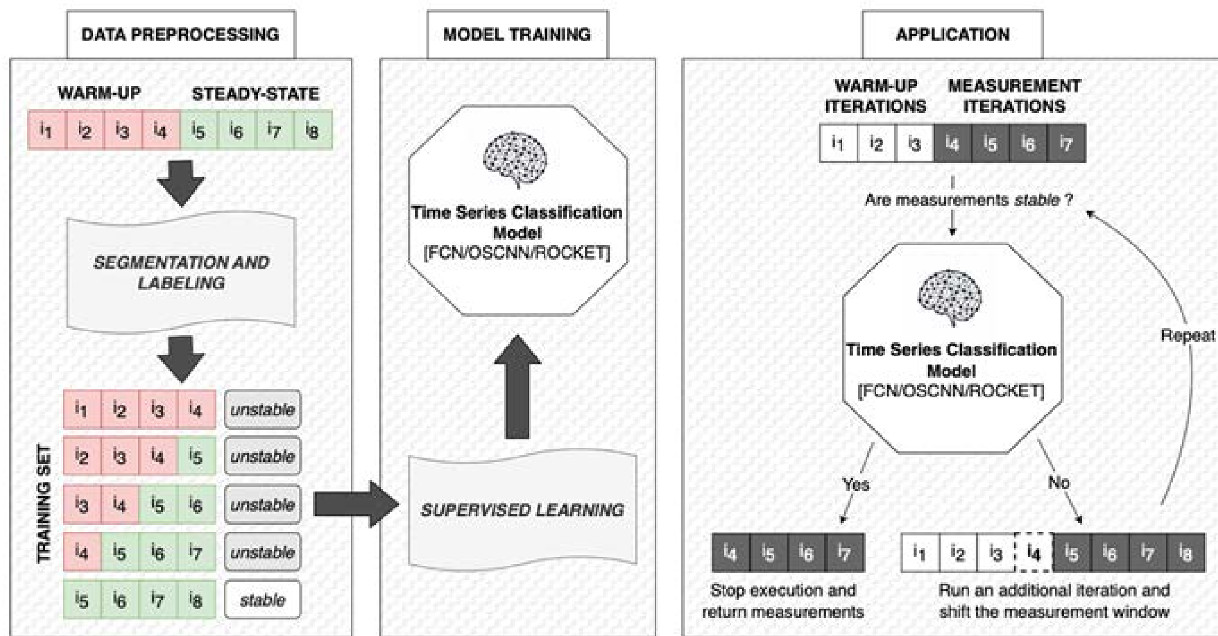

Figure 3. Overview of the main phases of the AI-assisted framework: Data Preprocessing, Model Training, and Application [8].

AI-powered tools can also adjust the testing window dynamically, optimizing when to stop or extend execution—contrasting with rigid traditional methods. In DevOps pipelines, this intelligence translates into faster feedback cycles, improved resource utilization, and greater alignment with production-like conditions.

Benefits and limitations of AI-assisted testing. The integration of AI into automated software testing has brought a paradigm shift in both functional and non-functional test processes. Unlike traditional frameworks that rely on static test generation, execution, and adaptation, AI-augmented frameworks introduce dynamic learning, semantic analysis, and intelligent decision-making [11,12].

While gTAA integrates project management, test management, and configuration pipelines using deterministic logic, AI frameworks adapt in real-time, adjusting test selection, input data generation, and validation based on system feedback and learned patterns.

In performance testing, AI-based systems extend traditional frameworks by incorporating TSC [13] for warm-up detection and measurement stability, reducing test duration without sacrificing accuracy. Similarly, in API testing, reinforcement learning and behavioral modeling allow for more diverse and meaningful test inputs, surpassing the limitations of scripted testing.

AI-enhanced frameworks offer several benefits: increased test coverage by autonomously exploring edge cases, reduced maintenance through self-healing mechanisms, and improved semantic alignment with business logic via AI-driven test case generation. Anomaly detection further enhances scalability, feedback speed, and resource efficiency, particularly in CI/CD environments [14].

However, AI-assisted testing faces challenges such as the lack of transparency in AI decisions, which raises issues in explainability and traceability—critical in regulated environments. AI models also depend on high-quality training data and remain in experimental stages for many testing tools, with limited production validation. Additionally, AI introduces complexity, requiring validation of the models themselves and integration into existing testing pipelines [15].

In summary, AI-enhanced frameworks add autonomy, resilience, and intelligence to traditional test automation, offering the potential to overcome current limitations and drive software testing toward greater efficiency and effectiveness.

3.3. Comparative Analysis: Traditional vs AI-assisted TAF

This paragraph presents a comparative documentary analysis aimed at introducing an AI-assisted testing approach for web applications. By synthesizing recent research on both traditional and AI-driven testing frameworks, the section proposes a practical methodology for integrating AI to enhance testing. This provides a roadmap for both researchers and practitioners to evaluate the current landscape and explore future improvements.

Table 1

Artificial Intelligence vs Traditional Automation Frameworks

Criterion	Traditional testing	AI-assisted testing	Cases / System under test characteristics
Test generation	Manual, scripted	Automated, based on machine learning/ large language models	Dynamic or large applications
Maintenance	High, requires constant updates	Reduced, with self-healing mechanisms	Systems with frequent changes or user interface instability
Adaptability to changes	Low	High, through contextual learning	Agile, fast-paced environments
Anomaly detection	Limited to predefined assertions	Dynamic, using predictive models	Detecting unexpected issues and regressions
Validation	Based on fixed rules	Contextual and flexible	Context-sensitive validation (e.g., application user interfaces)
Scripting effort	High	Minimal or non-existent	Minimal scripting and automation
(CI/CD) Integration	Partial, high latency	Optimized for continuous execution	CI/CD pipelines for faster feedback
Exploration capacity	Limited, predefined	Extended, through autonomous input generation	Complex, variable inputs
Scalability	Resource-constrained	Optimized through impact analysis	Scaling tests across large systems with varied inputs

Strategic selection of testing frameworks. AI vs Traditional Approaches. Traditional testing frameworks like Selenium and JUnit offer stability and maturity, but they face significant challenges in rapidly evolving environments, requiring high maintenance and

struggling with adaptability to frequent UI and API changes. These frameworks are limited in their ability to detect anomalies, as they rely on predefined assertions.

AI-assisted testing frameworks introduce autonomy, dynamic adaptation, and self-healing mechanisms, making them ideal for environments with constant updates. AI-driven systems can autonomously generate test cases, adapt to changes, and detect anomalies without predefined rules.

However, the rise of AI tools requires careful selection, as improper integration could lead to inefficiency and unnecessary complexity. A balanced approach that leverages the strengths of both traditional and AI-driven methods is crucial for maximizing testing effectiveness.

Framework selection based on system under test (SUT) characteristics. To guide framework selection, this paper introduces a Framework Selection Guide based on SUT characteristics. Key factors such as UI stability, API complexity, performance requirements, and rate of change influence the choice between traditional or AI-assisted frameworks. AI tools should be used strategically, offering value for systems with high volatility, like frequent UI updates, while stable systems may continue benefiting traditional methods. The goal is to avoid a one-size-fits-all approach, favoring tailored automation strategies. By evaluating these factors, teams can select the most suitable testing framework aligned with the SUT's needs.

4. Conclusions

This paper explores the transformation occurring in web application testing, as AI-driven frameworks are integrated alongside traditional testing methods. Our comparative analysis reveals that while traditional frameworks provide stability, control, and transparency, they struggle with the increasing complexity and dynamic nature of modern applications. The challenges include high maintenance costs, brittle test scripts, and difficulty in adapting to frequent UI or API changes.

AI-assisted testing frameworks offer a paradigm shift, providing autonomy, adaptability, and continuous learning. These systems excel at automatic test case generation, contextual validation, anomaly detection, and self-healing, greatly enhancing scalability, resilience, and resource optimization. However, AI integration comes with methodological and technical challenges—such as lack of explainability, tool maturity, and data dependency—which prevent it from being a universal solution.

Recommendations for practitioners. QA engineers are recommended to analyze adopting a hybrid testing approach where AI supports repetitive tasks, while traditional methods remain in place for critical, regulated, or complex scenarios. It's also essential to continuously evaluate AI tools' performance and reliability in real business applications, while investing in training to ensure proper integration and use of AI.

Recommendations for researchers. For researchers, exploring AI model validation and meta-testing techniques for automated testing, along with contributions to the development of open-source frameworks, would support the continued evolution of AI-based testing solutions.

Future directions. As part of our working plan, we aim to develop a comprehensive guide for framework selection based on SUT characteristics. This long-term initiative will leverage AI-assisted testing frameworks, integrating industry feedback and practical experience into an adaptable guide for researchers and practitioners. The guide will focus on

selecting the appropriate testing framework based on factors like UI complexity, API reliability, and performance needs, ensuring customized solutions for real-world applications.

In the short term, we will continue gathering practical experience by working closely with industry partners, conducting pilot studies, and evaluating AI tools under various application scenarios. This hands-on experience will inform the framework guide and provide actionable insights for practitioners looking to integrate AI-enhanced testing into their workflows.

The continuous development of Large Language Models and their integration into DevOps pipelines, combined with emerging technologies such as RPA, AIOps, and autonomous testing, promises to modernize quality assurance processes. Soon, AI-based testing will not just be a competitive advantage but a necessity for sustaining software delivery in dynamic, scalable, and user-centric environments.

Conflicts of Interest: The authors declare no conflict of interest

References

1. Fewster, M.; Graham, D. *Software Test Automation: Effective Use of Test Execution Tools*; Addison-Wesley: Boston, MA, USA, 2021, 448p.
2. Meszaros, G. *xUnit Test Patterns: Refactoring Test Code*; Addison-Wesley: Boston, MA, USA, 2018, 944p.
3. Jain, A.; Sharma, P. Challenges in Web Application Test Automation. *J. Softw. Eng. Res.* 2020, 5, pp. 45–58.
4. Kim, S.; Hassan, A. Automated Testing in Continuous Delivery Pipelines. *IEEE Softw.* 2022, 39, pp. 72–81.
5. Xie, X.; Zhang, L.; Wang, T.; Chen, P. Intelligent Automation in Software Testing Using AI and Machine Learning. *ACM Comput. Surv.* 2023, 55, pp. 1–32.
6. Zhao, H.; Li, P. Reinforcement Learning for Automated Web UI Testing. *Empir. Softw. Eng.* 2024, 29, pp. 331–349.
7. Li, Y.-F.; Das, P.K.; Dowe, D.L. Two Decades of Web Application Testing—A Survey of Recent Advances. *Inf. Syst.* 2014, 43, pp. 20–54.
8. Lai, C.-F.; Liu, C.-H. Using Webpage Comparison Method for Automated Web Application Testing with Reinforcement Learning. *Int. J. Eng. Technol. Innov.* 2024, 14, pp. 87–102.
9. Zhang, W.; Huang, J.; Sun, Q. A Reinforcement Learning Approach to Guide Web Crawlers to Explore Web Applications for Improving Code Coverage. *Electronics* 2024, 13, 427.
10. Nguyen, T.; Park, E. Using Large Language Model to Fill in Web Forms to Support Automated Web Application Testing. *Information* 2025, 16, 102.
11. Costa, A.; Silva, F. Towards an AI-Driven User Interface Design for Web Applications. *Procedia Comput. Sci.* 2024, 237, pp. 179–186. https://doi.org/10.1016/j.procs.2024.05.094.
12. Balsam, S.; Mishra, D. Web Application Testing—Challenges and Opportunities. *J. Syst. Softw.* 2024, 219, 112186. https://doi.org/10.1016/j.jss.2024.112186.
13. Amershi, S.; Weld, D.; Vorvoreanu, M.; Fourney, A.; Nushi, B.; Collisson, P.; Inkpen, K.; Bennett, P. Guidelines for Human–AI Interaction. In *Proceedings of the CHI Conference on Human Factors in Computing Systems (CHI '19)*, Glasgow, UK, 4–9 May 2019; ACM Press: New York, NY, USA, 2019, pp. 1–13.
14. Martín-López, A. AI-Driven Web API Testing. In *Proceedings of the 42nd ACM/IEEE International Conference on Software Engineering (ICSE '20)*, Seoul, Republic of Korea, 24–30 May 2020; ACM Press: New York, NY, USA, 2020, pp. 1–8.
15. Traini, L.; Menna, F. AI-Driven Java Performance Testing: Balancing Result Quality with Testing Time. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24)*, Sacramento, CA, USA, 23–27 September 2024; IEEE Press: Piscataway, NJ, USA, 2024, 454 p.
16. ISTQB ALTAE Homepage. Available online: https://www.istqb.org/certifications/certified-tester-advanced-level-test-automation-engineering-ctal-tae-v2-0/ (accessed on 20 November 2025).

Citation: Darii, O.; Beldiga, M.; Bragaru, T. Automated web application testing based on artificial intelligence. *Journal of Engineering Science*. 2025, XXXII (4), pp. 41-53. https://doi.org/10.52326/jes.utm.2025.32(4).04.

Publisher's Note: JES stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Copyright:© 2025 by the authors. Submitted for possible open access publication under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

Submission of manuscripts: jes@meridian.utm.md