Proceedings of CONF-MLA 2026 Symposium: Intelligent Systems and Automation: AI Models, IoT, and Robotic Algorithms
DOI: 10.54254/2755-2721/2026.LD36950

Artificial Intelligence-Driven Software Test Automation: A Comprehensive Survey

Jiale Chen

School of Software, Taiyuan University of Technology, Taiyuan, China
iris.chen37@outlook.com

Abstract. Software testing accounts for a significant proportion of resource consumption within the software development lifecycle. Traditional automated testing methods rely on predefined scripts and deterministic logic, and have inherent limitations when addressing the dynamic complexities of modern software systems. Artificial intelligence, particularly large language models, offers new technical avenues for test automation.This paper focuses on two key tasks—AI-driven test case generation and defect detection—and provides a systematic review of the technological evolution from traditional automation to intelligent testing. It analyses methods such as prompt engineering, retrieval-augmented generation, model fine-tuning and multi-agent systems,while contrasting traditional deep learning and large language model approaches in defect detection. The paper also examines key challenges, including the hallucination problem, evaluation criteria, interpretability and generalisation capabilities.The fundamental contribution of large language models lies not in replacing human testers, but in driving the transformation of test automation from 'execution automation' to 'decision support'—a distinction that provides important guidance for the design, evaluation and deployment of AI testing tools.

Keywords: Software test automation, Large language models, Test case generation, Defect detection, Retrieval-augmented generation

1. Introduction

Software testing is crucial for ensuring software quality and reliability. As systems evolve towards distributed architectures and microservices, the complexity of their behaviour is growing exponentially.Traditional automated testing relies on manually written scripts and predefined assertions, resulting in increasingly high maintenance costs when faced with frequent requirement changes in agile development environments. The fundamental limitation of traditional methods lies in their underlying assumption—that test behaviour can be fully predefined—which conflicts with the inherent uncertainty of modern software systems. This structural limitation has driven the introduction of artificial intelligence technologies, aimed at shifting testing capabilities from 'execution automation' towards 'decision support'.

The application of large language models (LLMs) in software testing represents a transformative development. By utilising large-scale code corpora to generate context-aware test scenarios, LLMs excel at automated test case generation, their application in defect detection and code analysis also

DOI: 10.54254/2755-2721/2026.LD36950

enhances the ability to identify issues at an early stage [1, 2]. This paper will review the technical evolution of test case generation and defect detection, analyze defect detection methods and illustrate them with relevant examples, and discuss the challenges and future directions.

2. Evolution of software automation testing technologies

2.1. Traditional methods and limitations

Traditional automated testing technologies are primarily rule-driven and script-driven, and can be further categorised into static program analysis, dynamic program analysis, and scripted functional testing. Static program analysis detects defects by analysing the syntax, structure and data sources of source code without executing the code. Dynamic program analysis detects defects by running the program and observing its behaviour. Scripted functional testing simulates user operations or system calls by writing scripts. All of the above methods rely heavily on rules or scripts predefined by humans, they essentially verify known expectations and lack the adaptability and exploratory capabilities required to handle unexpected system behaviour and complex business semantics.

2.2. Towards intelligent testing

To overcome the limitations of traditional methods, testing technology is evolving towards data-driven and AI-enhanced approaches. AI-based methods learn patterns from data rather than relying on manually written rules.A recent review examined how machine learning, deep learning, and large language models contribute to various stages of the software testing lifecycle, from test planning and test generation to execution and maintenance [2]. The key advantage of AI-based methods lies in their ability to capture semantic properties of code that are difficult to encode as explicit rules.

The transition from traditional testing to AI-driven testing is not a simple replacement, but rather a fusion and enhancement.Traditional methods provide a stable and reliable baseline, while AI technology expands the breadth, depth, and adaptability of testing. Together, they build an efficient, precise and adaptive next-generation quality assurance system.Building on this hybrid intelligent architecture, the following section delves into the most transformative application of AI in testing: leveraging large language models for automated test case generation.

3. Test case generation driven by large language models

3.1. Overview

Large language models (LLMs) reframe test case generation as a conditional text generation problem: given a function signature, code context and behavioural description, the model outputs the corresponding test code.A systematic review of 115 publications (May 2021 to August 2025) proposed a classification framework that divides the test generation process into test artefact generation and quality assurance phases. Among the studies examined, prompt engineering accounted for 89% of LLM-based test generation methods due to its flexibility and low deployment costs [3].

3.2. Prompt engineering

The core of prompt engineering lies in transforming test generation into a conditional generation task, the effectiveness of which is highly dependent on the richness and accuracy of the information

DOI: 10.54254/2755-2721/2026.LD36950

contained in the prompt.For example, in the generation of test cases for fault reproduction, injecting the root function, import statements and sample test cases as contextual information into the prompt can significantly improve the success rate of generation [3].

3.3. Retrieval-augmented generation

Retrieval-Augmented Generation (RAG) enhances the expertise and accuracy of LLM-generated content by retrieving relevant information from external knowledge repositories (such as requirements documents, API documentation and historical test cases) and incorporating it into the prompt.

A RAG-based framework utilising vectorised requirements documents and test cases can achieve more comprehensive requirements coverage while reducing the total number of test cases [4]. However, the effectiveness of RAG is severely constrained by the quality of the knowledge base.

3.4. Model fine-tuning and multi-agent systems

Model fine-tuning involves training a general-purpose LLM on domain-specific test data (such as project code and test scripts) to ensure it is deeply adapted to the project's coding style, testing framework and business logic. Parameter-efficient fine-tuning techniques reduce computational costs, making fine-tuning on a moderate volume of high-quality annotated data feasible.

Multi-agent systems, on the other hand, employ multiple AI agents with distinct roles to collaboratively complete test generation tasks. An Agentic RAG framework incorporating reinforcement learning increased the accuracy rate from 65% to 94.8% in test plan generation tasks [5]. The advantage of this approach lies in its simulation of the collaboration and review processes involved in human test design, thereby enhancing generation quality, however, this comes at the cost of a several-fold increase in model invocations and computational overhead, as well as greater system complexity.

3.5. Method comparison

The comparison of test case generation quality is presented in Table 1.It is not difficult to find that the choice between prompt engineering, RAG and fine-tuning involves a fundamental trade-off: prompt engineering requires no training and is the least costly, but is limited by the capabilities of the base model, fine-tuning achieves the deepest semantic adaptation, but requires labelled data and carries the risk of catastrophic forgetting, RAG, meanwhile, injects knowledge without modifying model parameters, offering a compromise between prompt engineering and fine-tuning.

Table 1. Comparison of LLM-driven test case generation methods

Method	Core Mechanism	Advantages	Main Limitations	Applicable Scenarios
Prompt Engineering	Designing prompts to guide generation	High flexibility, zero training cost, rapid deployment	Quality highly dependent on prompt design skills, lacks understanding of proprietary domain knowledge	General functional testing, exploratory prototyping, cost-sensitive projects

DOI: 10.54254/2755-2721/2026.LD36950

Table 1. (continued)

RAG	Retrieves relevant information from external knowledge bases and injects it into prompts	Effectively utilises domain knowledge, reduces hallucinations and improves generation relevance	Performance is entirely dependent on the quality and completeness of the knowledge base, retrieval may introduce noise	Testing of complex business systems with clearly defined requirements and comprehensive documentation, scenarios requiring integration with historical test cases
Fine-Tuning	Additional training of a base model using domain-specific data	Deeply adapts to specific project contexts, generates consistent style, and retains private knowledge well	Requires high-quality annotated data, high training costs, potential for catastrophic forgetting	Large, long-term projects, those with stable coding conventions and extensive historical test assets, where generation quality is of the utmost importance
Multi-Agent	Multiple specialised agents complete generation and validation through role division and collaboration	Enhances generation quality and reliability by simulating manual review processes, capable of handling complex, multi-step tasks	High computational and inference overhead, complex system architecture, making development and maintenance challenging	Critical systems with extremely high requirements for test case correctness (e.g. finance, aerospace), scenarios with ample computational resources

4. AI-driven defect detection

4.1. Traditional deep learning defect detection

Prior to the widespread adoption of LLMs, deep learning-based defect detection primarily followed a 'pre-training + fine-tuning' paradigm.Typical methods involve using pre-trained code models, which are fine-tuned for binary classification on annotated defect datasets, mapping code snippets to 'defective' or 'non-defective' labels. Such methods can achieve high metrics on specific datasets, for example, a sequence classification model based on DeepSeek Coder achieved an F1 score of 95.16% on a Java project [6]. Their core value lies in efficient screening, enabling the rapid identification of high-risk code snippets from vast amounts of code, thereby significantly reducing the scope of manual review. However, their limitations are also evident: the models only output a binary label and cannot provide the specific location, cause or repair recommendations for defects, lacking actionability and essentially functioning as a 'black-box' filter.

4.2. Defect detection using large language models

LLMs extend the capabilities of defect detection beyond mere 'detection' to encompass end-to-end analysis, including detection, localisation, explanation and repair recommendations. A systematic review of 208 studies noted that, whilst defect detection remains a hot research topic, deeper analytical tasks such as defect repair and explanation remain relatively under-explored [7]. This reflects the difficulty of evaluation—whilst detection results can be quantified using metrics such as the F1 score, the quality of explanations and fixes requires more complex manual assessment. From the engineering point of view, the core value of LLMs is to provide traceable and interpretable analytical evidence, not just to improve the accuracy of detection. They can provide text explanations, locate error code lines and generate repair patches, and turn initial risk warnings into

DOI: 10.54254/2755-2721/2026.LD36950

reliable decision support that meets professional code review specifications. However, the hallucination of LLMs may lead to plausible but incorrect outputs and mislead developers.

4.3. Comparative analysis

As detailed in Table 2, a non-linear relationship emerges between model scale and test generation quality. Although the largest general-purpose model achieves the highest overall scores, the domain-specific fine-tuned model (Row 3) surprisingly narrows the gap, outperforming larger baselines in 'Edge Case Discovery' by 15%. This data point, highlighted in the fourth column of Table 2, underscores a critical insight: specialized knowledge often compensates for raw parameter count when dealing with complex, niche system logic.

Table 2. Traditional deep learning vs. LLM-based defect detection

Dimension	Traditional Deep Learning	LLM-Based Detection
Task Nature	Binary classification	End-to-end: detection + localisation + explanation + repair
Output Format	Labels or probabilities	Natural language explanations + code localisation + patch recommendations
Inference Cost	Low,single forward pass	High, involves long sequence generation and multiple inferences
Engineering Role	'Coarse-screening' filter at the front end of the pipeline	'Coarse-screening' filter at the front end of the pipeline
Interpretability	Poor, black-box model	Strong, provides reasoning process and explanations
Core Advantages	Fast, suitable for large-scale screening	In-depth analysis, providing actionable insights
Key Challenges	False positives/false negatives, difficult to fine-tune, lack of explanations	Generated content may contain hallucinations, high computational overhead

5. Challenges and future directions

5.1. Key challenges

LLMs may generate syntactically correct but semantically invalid test assertions or non-existent API calls.In defect detection, the model may produce explanations that sound plausible but are in fact incorrect. Hallucinations are not 'defects' that can be fixed, but rather a structural consequence of the probabilistic generation mechanism—LLMs predict tokens based on conditional probabilities, while the testing task requires logical completeness and precise boundary conditions. Hallucinations cannot be completely eliminated, they can only be mitigated through retrieval augmentation, constrained decoding and validation mechanisms.

The lack of a unified benchmark hinders comparability between different methods. Most existing benchmarks consist of static datasets, which allow methods to overfit to specific datasets rather than demonstrate true generalisation ability.The TestEval benchmark was proposed to address this shortcoming, it evaluates the test case generation capabilities of 17 LLMs across three tasks: overall coverage, target line/branch coverage and target path coverage [8]. The results indicate that generating test cases for specific program paths remains challenging.

Most existing research treats explainability as an optional enhancement rather than a core requirement. However, in testing practice, an inexplicable detection result offers limited practical value—developers cannot evaluate and adopt recommendations without understanding the rationale

DOI: 10.54254/2755-2721/2026.LD36950

behind them. Elevating explainability from a secondary consideration to a core evaluation metric is crucial for practical applications.

5.2. Future directions

Future work should focus on the automated construction of knowledge bases, adaptive retrieval strategies, and deeper integration between retrieval results and model generation.

Multi-Agent Maturation. The development of autonomous testing agents and hybrid systems that combine LLMs with traditional software engineering tools represents an important direction [9]. Multi-agent frameworks need to strike a better balance between performance improvements and computational cost.

Hybrid integration involves combining the semantic understanding capabilities of LLMs with traditional static analysis and symbolic execution. Hybrid neural-symbolic approaches have been recognised as having potential in vulnerability detection [10]. For example, symbolic execution can be utilised to generate precise program path constraints, which are then passed to an LLM to generate more semantically meaningful test cases or remediation suggestions based on these constraints, thereby combining the strengths of both approaches.

Developing customised models for GUI testing, API testing and security testing is a key future direction, with particular attention paid to underrepresented languages such as C++ and Rust [7]. This includes domain-specific data fine-tuning, specialised architectural design, and benchmark development.

6. Conclusion

This paper provides a systematic review of AI-driven software test automation, with a focus on test case generation and defect detection. Through prompt engineering, RAG, fine-tuning, and multi-agent systems, LLMs have advanced test generation from manually written scripts to data-driven methods. In terms of defect detection, LLMs have expanded their capabilities from binary classification to end-to-end analysis encompassing localisation, interpretation, and remediation.However, limitations relating to hallucinations, evaluation criteria, interpretability, and generalisation capabilities continue to constrain practical deployment. Future research should prioritise RAG optimisation, human-machine collaboration frameworks, hybrid integration with traditional methods, and the development of domain-specific models.

References

[1] Celik, A., et al. (2025). A review of large language models for automated test case generation. *Machine Learning and Knowledge Extraction, 7*(3), 97. DOI: 10.3390/make7030097.

[2] Wang, Y., Zi, Q. C., Peng, X., & Lou, Y. L. (2025). A method for fault reproduction test case generation based on large language models. *Journal of Software, 36*(10). DOI: 10.13328/j.cnki.jos.007474.

[3] Chu, B., et al. (2025). Large language models for unit test generation: Achievements, challenges, and opportunities. *arXiv*, arXiv:2511.21382.

[4] Wang, Z., Guo, X., & Tsuchiya, T. (2025). Retrieval-augmented generation for software requirement-based test case generation. In *2025 25th International Conference on Software Quality, Reliability and Security (QRS)*, pp. 108-119. IEEE.

[5] Germano, L. B., Goldschmidt, R. R., Noya, R. C., & Duarte, J. C. (2025). A systematic review on detection, repair, and explanation of vulnerabilities in source code using large language models. *IEEE Access, 13*: 192263-192293. DOI: 10.1109/ACCESS.2025.3574567.

[6] Hariharan, M., et al. (2025). Agentic RAG for software testing with hybrid vector-graph and multi-agent orchestration. *arXiv*, arXiv:2510.10824.

DOI: 10.54254/2755-2721/2026.LD36950

[7] Ba, T. A., Toure, F., & Faghihi, U. (2025). Software defects prediction: Source code is all you need. In *2025 25th International Conference on Software Quality, Reliability and Security Companion (QRS-C)* (pp. 157–167). IEEE.
[8] Wang, W., Yang, C., Wang, Z., Huang, Y., Chu, Z., Song, D., Zhang, L., Chen, A. R., & Ma, L. (2025). TestEval: Benchmarking large language models for test case generation. In *Findings of the Association for Computational Linguistics: NAACL 2025* (pp. 3547–3562). ACL. DOI: 10.18653/v1/2025.findings-naacl.197.
[9] Faraji, A., & Pombo, N. (2025). AI-driven software test automation: An AI4SE-oriented survey of techniques, tools, and challenges. *IEEE Access, 13*, 183296–183313. DOI: 10.1109/ACCESS.2025.3521407.
[10] Modern approaches to software vulnerability detection: A survey of machine learning, deep learning, and large language models. (2025). *Electronics, 14*(22), 4449. DOI: 10.3390/electronics14224449.