

AI-Driven Software Testing: An Adaptive Agentic Framework for Context-Aware Test Generation, Validation, Prioritization, and Refinement

Sanika Jaysing Dorge¹, Prof. Pratiksha Vinayak Gaikwad², Dr. H. B. Jadhav³

Student, Department of Computer Engineering¹

Guide, Department of Computer Engineering

Co-Guide, Department of Computer Engineering³

Adsul's Technical Campus, Chas Ahilyanagar, India

Abstract: *Software testing is a critical activity in the software development life cycle, but conventional test automation still requires substantial human effort for test design, maintenance, execution, and analysis. Recent advances in artificial intelligence (AI), machine learning (ML), natural language processing (NLP), and large language models (LLMs) have created new opportunities to automate and improve several testing activities. This paper presents a structured review of AI-driven software test automation, emphasizing test-case generation, test prioritization, defect prediction, test execution, failure analysis, and automated test repair. The study also examines the emerging role of LLMs in understanding requirements and source code and in producing developer-readable tests. AI-assisted approaches can improve coverage, fault detection, and testing productivity, but generated tests may contain hallucinations, weak assertions, flaky behavior, or inadequate contextual understanding. Therefore, human supervision, execution-based validation, coverage measurement, mutation testing, and secure integration with CI/CD pipelines remain important. The paper proposes a practical AI-driven testing workflow that combines requirement analysis, intelligent test generation, automated execution, feedback, and continuous refinement.*

Keywords: Artificial Intelligence, Software Testing, Test Automation, Machine Learning, Large Language Models, Test Case Generation, AI4SE, Defect Prediction

I. INTRODUCTION

Software testing identifies defects and provides evidence that a software system satisfies its functional and non-functional requirements. As modern applications become distributed, cloud-based, mobile, data-driven, and continuously deployed, the number of possible execution paths and system states increases rapidly. Manual testing alone therefore does not scale. Conventional automation reduces repetitive work, but test scripts still have to be designed, updated, executed, and investigated when failures occur. Artificial intelligence provides a broader approach to automation by allowing testing systems to learn from code, requirements, execution traces, historical defects, and previous test results. Machine learning can be used for defect prediction and test prioritization, NLP can transform natural-language requirements into test scenarios, and LLMs can generate unit, API, integration, and acceptance tests from source code and documentation. The objective of this paper is to explain the major techniques used in AI-driven software test automation, present a practical workflow, compare important approaches, discuss current tools and limitations, and identify future research directions. The paper is a literature-based technical review and does not claim an independent benchmark experiment.



II. BACKGROUND AND AI TECHNIQUES

A. Machine Learning for Software Testing. Machine learning models can learn relationships between software characteristics and testing outcomes. Historical bug reports, code metrics, commit information, execution logs, and previous test results can be used as input features. Classification models can estimate whether a component is likely to contain defects, while ranking models can help prioritize tests under limited execution time.

B. Natural Language Processing NLP techniques can process requirements, user stories, issue descriptions, and acceptance criteria. The system can identify actors, actions, inputs, expected outcomes, constraints, and boundary conditions. These elements can then be transformed into test scenarios.

C. Large Language Models LLMs extend NLP by generating structured outputs from natural-language prompts and software context. An LLM can receive a Java method and generate JUnit tests, or receive an API specification and produce test cases for valid, invalid, and boundary inputs.

III. EXISTING SYSTEM

The conventional system uses manually designed requirements-to-test workflows and script-based automation. Testers interpret requirements, create test cases, develop scripts, execute them through testing tools, and manually investigate failures. The approach reduces repetitive execution but continues to require significant human effort for test design, maintenance, prioritization, and failure analysis.

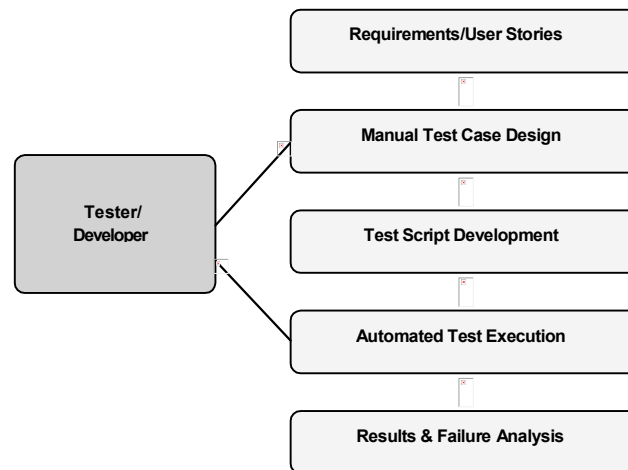


Fig.1 Existing System Architecture

The existing architecture is mainly script-centered. When the application changes, test scripts and expected results may require manual updates. Historical failures and execution information are not necessarily used systematically to generate or prioritize future tests.

IV. PROPOSED SYSTEM

The proposed system introduces an AI-assisted, feedback-driven workflow. Requirements, source code, API specifications, configuration, existing tests, historical defects, and execution logs are analyzed. NLP and static analysis identify functions, interfaces, dependencies, constraints, and business rules. The AI engine generates candidate tests covering normal, boundary, invalid, exception, and negative scenarios. Tests are validated before controlled execution, and execution feedback is used for repair or regeneration.



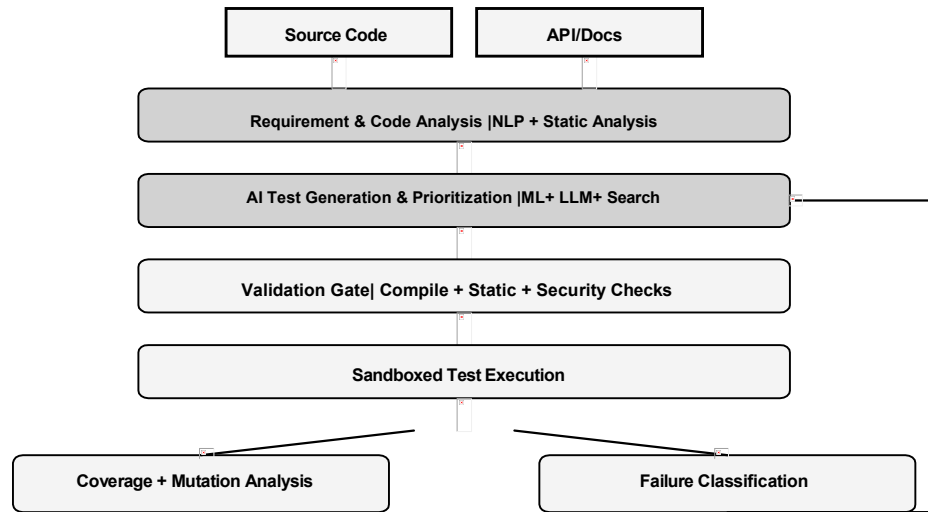


Fig2. Proposed AI-Driven Software Test Automation Architecture.

A. Requirement and Code Analysis: The first stage collects requirements, user stories, API specifications, source code, configuration, and existing tests. NLP and static-analysis methods identify functions, interfaces, dependencies, constraints, and important business rules. Retrieval-augmented generation can supply relevant project documentation as context.

B. Test Case Generation: The AI engine generates candidate test cases covering normal, boundary, invalid, exception, and negative scenarios. A structured format may contain test ID, input, precondition, action, expected result, and priority. For API testing, cases can include endpoint, method, parameters, headers, response assertions, and negative cases.

C. Automated Execution and Measurement: Generated tests are executed in an isolated environment. Execution results are collected together with statement, branch, path, and mutation coverage where appropriate. Failed tests are classified as genuine defects, invalid generated tests, environment failures, or flaky tests. This step is essential because syntactically correct AI-generated code is not necessarily a valid or useful test.

D. Feedback, Repair, and Maintenance: The system feeds execution information back to the AI component. Tests that fail because of incorrect imports, weak assertions, unavailable fixtures, or changed interfaces can be repaired or regenerated. When the application changes, impacted tests can be identified and prioritized.

V. PROPOSED FRAMEWORK

The framework contains ten logical stages: planning, retrieval, codebase analysis, generation, validation, prioritization, execution, analysis, refinement, and reporting.

Stage	Component	Responsibility
1	Planning Agent	Define testing goals, risks, and required scenarios.
2	RAG Retriever	Retrieve relevant requirements, code, documentation, and tests.
3	Codebase Analyzer	Extract syntax, methods, classes, and dependencies.
4	Generation Agent	Generate executable unit or integration tests.
5	Validation Agent	Compile tests and check execution/assertion failures.
6	Prioritization	Rank tests by quality and cost.
7	Execution Agent	Run tests and collect objective evidence.



8	Analysis Agent	Find gaps, surviving mutants, failures, and redundancy.
9	Refinement Agent	Generate targeted improvements.
10	Reporting Agent	Produce reproducible results and metrics.

Table I. Components of the proposed adaptive testing framework.

I. Retrieval-Augmented Context: The system collects source code, requirements, documentation, existing tests, dependency information, and relevant execution history. A retrieval layer selects context related to the target component before an LLM is invoked.

II. Planning: The planning agent converts the target component and requirements into a testing plan containing normal cases, boundary conditions, invalid inputs, exception paths, state-dependent cases, and high-risk dependencies.

III. Generation: The generation agent creates candidate tests from the plan and retrieved context. The generated tests are expected to contain executable code and explicit assertions.

IV. Validation and Execution: Candidate tests are compiled and executed in an isolated environment...

V. Analysis and Refinement: The analysis agent identifies uncovered behavior, surviving mutants, failed tests, redundant scenarios, and weak assertions. The refinement agent uses this structured feedback to generate targeted additions or repairs.

VI. COMPARATIVE ANALYSIS OF AI TECHNIQUES

A. Machine Learning: defect prediction and test prioritization; benefit—learns from historical project data; limitation—depends on representative training data.

B. NLP: requirement-to-test transformation; benefit—connects natural language with testing; limitation—ambiguous requirements can cause errors.

C. LLMs: unit/API/acceptance test generation; benefit—strong code and language understanding; limitation—hallucinations, weak assertions, and context limits.

D. Search-Based Testing: input generation and coverage optimization; benefit—optimizes measurable testing objectives; limitation—search cost can increase for complex systems.

E. Reinforcement Learning: adaptive test selection and exploration; benefit—learns from execution feedback; limitation—training and reward design can be difficult.

VII. ADAPTIVE TEST PRIORITIZATION

The framework assigns each test a priority score using measurable properties:

$Priority(T_i) = w_c \times CoverageGain + w_m \times MutationGain + w_u \times Uniqueness \times ExecutionCost$. CoverageGain measures additional source coverage contributed by a test. MutationGain measures its contribution to killing previously surviving mutants. Uniqueness represents the degree to which the test adds behavior not already represented in the suite. ExecutionCost represents time or computational expense. The weights are configurable and should be tuned experimentally rather than treated as universal constants. The score is recalculated after refinement. A test that kills a previously surviving mutant can therefore increase in priority, while a redundant test can decrease in priority.

- A. Hallucination and Incorrect Tests
- B. Test Oracle Problem
- C. Flaky and Non-Maintainable Tests
- D. Security, Privacy, and Intellectual Property

VIII. ALGORITHM

Input: repository R, requirements Q, existing tests T, and a selected LLM or LLM ensemble. Output: validated and prioritized test suite T*.

1. Parse the repository and construct code and dependency representations.
2. Retrieve relevant requirements, source code, documentation, existing tests, and execution history.



3. Generate a testing plan.
4. Generate candidate tests using the plan and retrieved context.
5. Compile and execute the candidates.
6. Measure coverage, mutation results, failures, redundancy, and execution cost.
7. Rank tests using the adaptive priority function.
8. Identify uncovered behavior and surviving mutants.
9. Generate targeted improvements from the structured feedback.
10. Repeat validation and refinement until the stopping condition or improvement plateau is reached.
11. Return the validated and

IX. TOOLS AND INTEGRATION WITH CI/CD

AI-driven testing can be integrated with existing software engineering tools rather than replacing them. Unit-testing frameworks such as JUnit or pytest can execute generated tests, while browser and API automation frameworks can execute end-to-end scenarios. Static analysis and coverage tools can provide feedback to the AI component. CI/CD platforms can trigger the pipeline after each commit or pull request. An effective pipeline can be organized as: code commit → impact analysis → AI-assisted test generation or selection → static validation → test execution → coverage and mutation analysis → defect/failure classification → developer review → merge or remediation. Access controls, secret filtering, sandboxing, and logging should be implemented because source code and test data may contain sensitive information.

X. CHALLENGES

1. An AI model may invent APIs, classes, variables, expected outputs, or configuration details. Compilation, execution, coverage, and mutation testing should be used to detect such cases.
2. Generating an input is often easier than determining the correct expected output. AI systems need reliable oracles derived from specifications, invariants, reference implementations, metamorphic relations, or domain rules.
3. Generated tests may depend on timing, external services, unstable selectors, random data, or implementation details. Test generation should therefore include maintainability constraints and prefer stable interfaces and meaningful assertions.
4. Sending proprietary source code, logs, credentials, or user data to an external AI service can create privacy and security risks. Enterprise deployments should apply data minimization, access controls, secret redaction, secure model hosting where required, and audit logging.
5. Evaluation and Reproducibility: AI outputs can vary across prompts, model versions, temperature settings, and context. Research evaluations should report datasets, prompts, models, tool versions, execution environments, coverage measures, mutation scores, and failure categories.

XI. FUTURE DIRECTIONS

Future AI-driven testing systems are likely to become more context-aware and agentic. Instead of generating a test once, an AI agent may inspect the repository, identify risky components, generate tests, execute them, analyze failures, repair tests, and open a development ticket. Retrieval-augmented generation can improve project-specific knowledge, while static analysis can constrain the model to valid program structures. Hybrid approaches can combine LLM semantic understanding with symbolic execution, fuzzing, static analysis, search-based optimization, and conventional test frameworks.

XII. EXPERIMENTAL METHODOLOGY

The evaluation compares four configurations to isolate the effect of the proposed components.



Configuration	Description
A — Conventional	Existing automated test - generation tool or conventional test suite.
B— Single LLM	One LLM generates tests from source code and context.
C— Multi-LLM	Multiple LLMs cooperate through generation and refinement.
D—Adaptive Agentic	RAG, codebase context, specialized agents, validation, prioritization, and feedback.

Dataset: A reproducible evaluation can use HumanEval-Java or another public Java benchmark for controlled method-level experiments and a separate set of open-source repositories for repository-level testing. Multiple projects should be used to reduce the risk of conclusions being specific to one codebase.

Implementation: JUnit can execute generated tests, JaCoCo can measure line and branch coverage, and PIT can provide mutation-testing evidence. Static analysis can extract syntax and dependencies, while a vector store can support RAG. Model names, versions, prompts, temperature, retrieval settings, and hardware should be reported.

Metric	Purpose
Compilation rate	Measures syntactic and API compatibility of generated tests.
Execution success	Measures successful execution without infrastructure or runtime errors.
Line coverage	Measures executed source statements.
Branch coverage	Measures exercised decision branches.
Mutation score	Measures fault - detection capability against injected faults.
Fault detection	Measures real or seeded defects detected.
Redundancy	Measures duplicate or behaviorally overlapping tests.
Execution time	Measures testing efficiency.
Token/API cost	Measures AI resource cost.
Readability	Optional developer evaluation of maintainability.

Table.III. Evolution Matrices

XII. RESEARCH

H1: Feedback-based refinement increases mutation score compared with single-pass LLM generation.

H2: Codebase-aware retrieval improves branch coverage for dependency-rich subjects.

H3: Adaptive prioritization reaches a selected coverage or mutation threshold with fewer executed tests.

H4: Validation and refinement reduce compilation failures and redundant tests.

XIII. DISCUSSION

The framework changes the role of an LLM from a one-time test author to a component of an evidence-driven testing process. A generated test can appear plausible while containing an incorrect expected value, missing an important path, or duplicating an existing scenario. Objective execution evidence provides a stronger basis for deciding whether a generated test should be retained. RAG provides project-specific information, static analysis provides structural information, and specialized agents divide testing into manageable responsibilities. The execution environment then acts as a source of objective evidence. Compilation failures, coverage changes, mutation outcomes, and execution cost can be transformed into structured feedback for the next generation cycle. The architecture is also model-independent at the workflow level. Different foundation models can be connected to the same generation or refinement interfaces, allowing future studies to compare models without redesigning the entire testing process.

XIV. THREATS TO VALIDITY

Internal validity may be affected by prompt design, model randomness, retrieval quality, and model-version differences. These factors should be controlled and reported. Construct validity is affected by the selected coverage and mutation metrics; neither metric alone proves functional correctness. External validity may be limited if only small



benchmarks are used, so controlled benchmarks should be complemented by larger open-source repositories. Cost and latency should also be reported because an accuracy improvement may not justify excessive AI usage.

XV. SECURITY AND RESPONSIBLE USE

AI-driven testing can expose source code, configuration details, API schemas, or defect information when cloud-hosted models are used. Sensitive projects should use approved deployment arrangements, access controls, redaction, or locally hosted models where required. AI-generated commands and tests should be reviewed before they are allowed to affect the system.

XVI. FUTURE WORK

Future work can extend the framework to API testing, integration testing, regression-test selection, GUI testing, security testing, and autonomous CI/CD execution. Multimodal models can add screenshots and visual state to the retrieval and analysis process. A cost-aware stopping policy could also learn when another LLM call is unlikely to provide sufficient additional testing value. Another direction is repository-level defect localization, where failed tests and code dependencies are jointly analyzed to identify likely fault locations.

XVII. CONCLUSION

This paper presents an adaptive agentic framework for AI-driven software testing that integrates retrieval-augmented context, codebase awareness, specialized AI agents, automated execution, test prioritization, and feedback-driven refinement. The framework addresses a practical weakness of direct LLM test generation: a generated test is not necessarily useful merely because it is syntactically valid or linguistically plausible. The proposed workflow forms a closed loop in which software context is retrieved, tests are generated, executed, measured, prioritized, analyzed, and refined. This creates a measurable research direction for evaluating whether context-aware and agentic AI can improve the effectiveness and efficiency of automated software testing. The research contribution is an integrated, experimentally testable architecture. Its effectiveness should be established through reproducible experiments against appropriate baselines, rather than through assumed performance values.

XVIII. ACKNOWLEDGMENT

The authors would like to thank the Department of Computer Engineering, [College Name], for academic guidance and support in preparing this review study.

REFERENCES

- [1] V. Garousi, N. Joy, and D. Taibi, "AI-Powered Software Testing Tools: Full Autonomy Remains a Distant Goal," *IEEE Software*, vol. 42, no. 4, pp. 98 – 111, 2025, doi: 10.1109/MS.2025.3557895.
- [2] A. Faraji and N. Pombo, "AI-Driven Software Test Automation: An AI4SE-Oriented Survey of Techniques, Tools, and Challenges," *IEEE Access*, vol. 13, pp. 183296 – 183313, 2025, doi: 10.1109/ACCESS.2025.3623944.
- [3] "Acceptance Test Generation with Large Language Models: An Industrial Case Study," 2025 IEEE/ACM International Conference on Automation of Software Test (AST), 2025, doi: 10.1109/AST66626.2025.00007.
- [4] "Comparative Study on Test Case Generation Using Generative AI," 2025 Intelligent Methods, Systems, and Applications (IMSA), 2025, doi: 10.1109/IMSA65733.2025.11166964.
- [5] "Framework and Performance Evaluation of Test Case Generation for Large Language Models in Software Testing," 2025 4th International Conference on Electronic Information Technology (EIT), 2025, doi: 10.1109/EIT67313.2025.11232178.
- [6] "Intent-Based E2E Automated Test Case Generation for Web Applications Using LLM," 2025 IEEE 49th Annual Computers, Software, and Applications Conference (COMPSAC), 2025, doi: 10.1109/COMPSAC65507.2025.00161.



- [7] “Program Interoperable Large Language Model Software Testing Scheme: A Case Study on JavaScript Engine Fuzzing,” IEEE Transactions on Dependable and Secure Computing, vol. 22, no. 6, pp. 6179 – 6195, 2025, doi: 10.1109/TDSC.2025.3581213.
- [8] A. M. Abdullin, P. Derakhshanfar, and A. Panichella, “Test Wars: A Comparative Study of SBST, Symbolic Execution, and LLM-Based Approaches to Unit Test Generation,” in 2025 IEEE Conference on Software Testing, Verification and Validation (ICST), 2025, pp. 221–232, doi: 10.1109/ICST62969.2025.10989033.
- [9] S. Kumar et al., “Artificial Intelligence in Software Testing: A Systematic Review,” 2023 IEEE Region 10 Conference (TENCON), 2023, doi: 10.1109/TENCON58879.2023.10322349.
- [10] Y. Tang et al., “Software Testing With Large Language Models: Survey, Landscape, and Vision,” IEEE Transactions on Software Engineering, vol. 50, no. 4, pp. 911 – 936, 2024, doi: 10.1109/TSE.2024.3368208.

