Varhadi Shreya Sachin, 2026, 14:5
ISSN (Online): 2348-4098
ISSN (Print): 2395-4752

International Journal of Science, Engineering and Technology
An Open Access Journal

Artificial Intelligence For Software Testing: A Review Of Automated Test Generation, Defect Prediction, Test Prioritization And LLM-Assisted Testing

Varhadi Shreya Sachin [1]Prof. S. S. Bhor[2] Prof. M. C. Kolse[3] Prof. S. K. Said[4]
Department of Artificial Intelligence and Data Science
Jaihind College of Engineering
Pune, Maharashtra, India

Abstract—Software testing consumes a large share of development effort, and manual test design cannot keep pace with frequent releases. Artificial intelligence and machine learning can automate test case generation, predict where defects are likely, prioritize which tests to run and help analyse results. This paper reviews AI for software testing. It examines search-based and feedback-directed automated test generation, including evolutionary generation of unit test suites and random testing guided by execution feedback, and whitebox fuzzing that uses symbolic execution to reach deep program paths. It reviews defect prediction from code metrics and change history, systematic evidence on fault-prediction performance, and reinforcement learning for test case prioritization and selection in continuous integration, where the goal is to find failures early within a limited time budget. It also reviews the use of large language models trained on code to generate unit tests, with empirical findings on coverage, correctness and the need for human review. A five-layer architecture comprising code and history data, risk modelling, test generation, execution and triage, and feedback layers is proposed, along with research gaps, objectives, a comparison of techniques and challenges such as the oracle problem, flaky tests, dataset quality and trust. The review concludes that AI makes testing faster and better targeted but does not remove the need for human judgement.

Index Terms—AI for software testing, automated test generation, search-based testing, defect prediction, test prioritization, continuous integration, large language models, fuzzing

I. INTRODUCTION

1. Motivation

In a modern project, developers may commit many times a day, and a full regression suite can take hours to run. Writing tests for new code and deciding which existing tests to run are both expensive decisions that are usually made by intuition.

AI offers help on both fronts. Harman *et al.* [1] showed that software engineering problems, including test generation, can be formulated as search problems and solved with metaheuristics. Machine learning can also learn from a project's history which files tend to break and which tests tend to catch failures.

This paper reviews AI techniques across the testing lifecycle and proposes an architecture that links risk prediction, test generation and continuous integration.

2. Objectives Of The Review

- To review AI techniques for automated test generation.
- To examine defect prediction and risk-based testing.
- To study learning-based test prioritization in continuous integration.
- To assess LLM-assisted test generation and its limitations.
- To propose an integrated AI-assisted testing architecture.

II. OVERVIEW OF KEY CONCEPTS

1. Testing Activities Amenable To Ai

Fig. 1 maps AI techniques onto test generation, test selection and prioritization, defect prediction and result analysis.

© 2026 Varhadi Shreya Sachin, This is an Open Access article distributed under the terms of the Creative Commons Attribution License (http://creativecommons.org/licenses/by/4.0/), which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly credited.

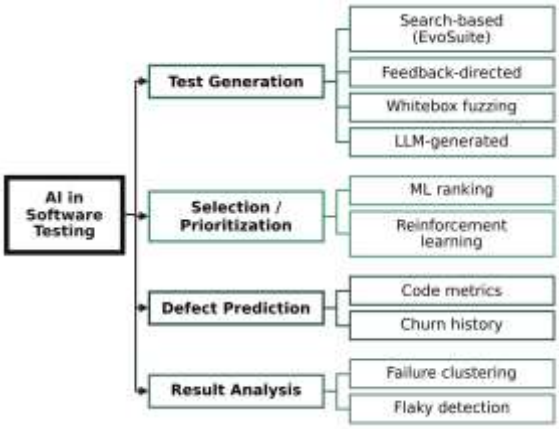


Fig. 1. AI techniques across software testing activities.

2. The Oracle Problem

Generating inputs is easier than knowing the expected output. Without a specification, automated tests often only detect crashes or assert current behaviour, which limits their ability to find functional faults.

III. REVIEW OF EXISTING WORK

Automated test generation. Pacheco *et al.* [2] proposed feedback-directed random testing, which builds new test inputs from previously executed sequences, improving over pure random testing. Fraser and Arcuri [3] developed EvoSuite, which uses evolutionary search to generate JUnit test suites that maximize code coverage and include assertions. Godefroid *et al.* [4] combined symbolic execution with fuzzing in whitebox fuzz testing to reach code paths that random inputs miss.

Defect prediction. Nagappan and Ball [5] showed that relative code churn measures predict defect density in large systems. Hall *et al.* [6] systematically reviewed fault prediction studies and found that model performance depends strongly on the data and the modelling choices, cautioning against over-optimistic claims. Random forests [7] and gradient boosting [8] are common models in this area.

Test prioritization in CI. Spieker *et al.* [9] applied reinforcement learning to prioritize and select test cases in continuous integration, learning from past failures to run likely-failing tests first within a time budget.

LLM-assisted testing. Chen *et al.* [10] evaluated large language models trained on code and showed they can synthesize functions from descriptions. Schäfer *et al.* [11] empirically evaluated LLM-based unit test generation and reported substantial coverage with the caveat that many generated tests need repair or review. Table I summarizes these works.

Sr. No.	Ref.	Year	Methodology	Findings	Research Gap
1	[5]	2005	Code churn defect prediction	Churn predicts defects	Large systems only
2	[2]	2007	Feedback-directed random testing	Better than random	Weak oracles
3	[4]	2008	Whitebox fuzzing	Deep path coverage	Scalability
4	[3]	2011	EvoSuite search-based generation	High coverage suites	Readability
5	[1]	2012	Search-based SE survey	Unified formulation	Survey
6	[6]	2012	Fault prediction review	Performance depends on data	Heterogeneous studies
7	[9]	2017	RL test prioritization	Earlier failure detection	Needs history
8	[11]	2024	LLM unit test generation	Good coverage with repair	Review required

TABLE I Summary of Literature Survey

IV. IDENTIFIED RESEARCH GAPS

- **G1:** Automated tests lack oracles for correct functional behaviour.
- **G2:** Defect prediction models transfer poorly across projects.
- **G3:** Flaky tests mislead learning from past failures.
- **G4:** Generated tests may assert buggy current behaviour.
- **G5:** Developers may not trust or maintain machine-generated tests.

V. SYSTEM ARCHITECTURE

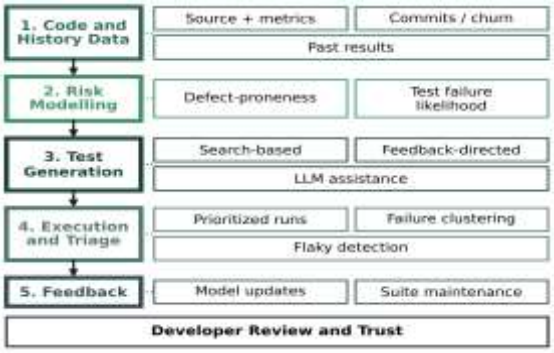


Fig. 2 shows the proposed architecture.

Fig. 2. Proposed five-layer AI-assisted software testing architecture.

1. Code And History Data Layer

Source code, static metrics, commit and churn history, past test results and bug reports are collected from the repository and CI system.

2. Risk Modelling Layer

A model predicts defect-proneness per file or module and estimates the failure probability of each test for the current change.

C. Test Generation Layer

Search-based and feedback-directed generators create unit tests for high-risk modules; an LLM proposes readable tests and assertions from code and documentation, flagged for developer review.

3. Execution And Triage Layer

Prioritized tests run first; failures are clustered, flaky tests are identified by repeated runs and likely root causes are suggested.

4. Feedback Layer

Results update the risk and prioritization models, and accepted tests are added to the regression suite.

VI. IMPLEMENTATION METHODOLOGY

Fig. 3 describes the workflow. Test prioritization maximizes early fault detection, measured by

$$APFD = 1 - (\Sigma_i TF_i)/(n \cdot m) + 1/(2n) \quad (1)$$

where TF_i is the position of the first test revealing fault i. Generated suites are assessed by branch coverage and mutation score,

$$Mutation\ score = mutants_killed / mutants_total \quad (2)$$

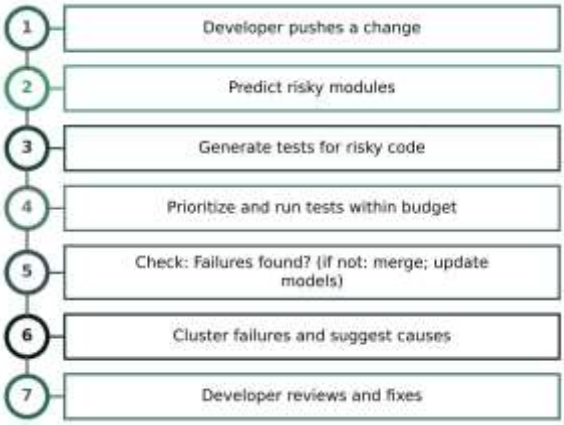


Fig. 3. AI-assisted testing workflow in continuous integration.

VII. PERFORMANCE COMPARISON

Table II compares techniques by automation level, oracle support and effort.

Technique	Automation	Oracle Support	Effort
Manual testing	Low	Strong (human)	Very high
Search-based generation	High	Weak	Low
Whitebox fuzzing	High	Crash-based	Medium
Defect prediction	Advisory	N/A	Low
LLM-assisted tests	High	Medium	Review needed

TABLE II Comparison of AI Testing Techniques

VIII. OPEN ISSUES AND FUTURE DIRECTIONS

Table III summarizes challenges. Future directions include specification mining for oracles, self-healing UI tests, AI-assisted security testing and agentic pipelines that generate, run and repair tests with human approval.

Challenge	Mitigation
Oracle problem	Specification mining; property-based tests
Flaky tests	Repeated runs; flakiness models
Cross-project transfer	Local retraining; project features
Asserting buggy behaviour	Developer review of assertions
Trust and maintenance	Readable tests; CI integration

TABLE III Challenges and Mitigation

IX. CONCLUSION

AI improves software testing by generating tests automatically, predicting risky code, prioritizing regression runs and assisting triage. The reviewed literature shows substantial gains in coverage and early fault detection, while the oracle problem, flaky tests and the need for review remain. Used as an assistant to developers rather than a replacement, AI can make testing faster, cheaper and more reliable.

1. Acknowledgment

The authors express their sincere gratitude to the Department of Artificial Intelligence and Data Science, Jaihind College of Engineering, Pune, for providing the facilities, guidance and encouragement needed to carry out this work. The authors also thank the faculty members and colleagues whose valuable suggestions helped improve the quality of this review.

REFERENCES

1. M. Harman, S. A. Mansouri, and Y. Zhang, "Search-based software engineering: Trends, techniques and applications," ACM Comput. Surv., vol. 45, no. 1, Art. no. 11, 2012.
2. C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in Proc. 29th Int. Conf. Softw. Eng. (ICSE), 2007, pp. 75–84.
3. G. Fraser and A. Arcuri, "EvoSuite: Automatic test suite generation for object-oriented software," in Proc. 19th ACM SIGSOFT Symp. Found. Softw. Eng. (ESEC/FSE), 2011, pp. 416–419.
4. P. Godefroid, M. Y. Levin, and D. Molnar, "Automated whitebox fuzz testing," in Proc. Netw. Distrib. Syst. Secur. Symp. (NDSS), 2008.
5. N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in Proc. 27th Int. Conf. Softw. Eng. (ICSE), 2005, pp. 284–292.
6. T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell, "A systematic literature review on fault prediction performance in software engineering," IEEE Trans. Softw. Eng., vol. 38, no. 6, pp. 1276–1304, 2012.
7. L. Breiman, "Random forests," Mach. Learn., vol. 45, no. 1, pp. 5–32, 2001.
8. T. Chen and C. Guestrin, "XGBoost: A scalable tree boosting system," in Proc. 22nd ACM SIGKDD Int. Conf. Knowl. Discov. Data Min., 2016, pp. 785–794.
9. H. Spieker, A. Gotlieb, D. Marijan, and M. Mossige, "Reinforcement learning for automatic test case prioritization and selection in continuous integration," in Proc. 26th ACM SIGSOFT ISSTA, 2017, pp. 12–22.
10. M. Chen et al., "Evaluating large language models trained on code," arXiv:2107.03374, 2021.
11. M. Schäfer, S. Nadi, A. Eghbali, and F. Tip, "An empirical evaluation of using large language models for automated unit test generation," IEEE Trans. Softw. Eng., vol. 50, no. 1, pp. 85–105, 2024.