Vol. 16 (2026): Proceedings of the 2026 International Conference on AI and Technological Innovation Applications (AITIA 2026)
E-ISSN: 3105-0522 P-ISSN: 3105-0514
DOI: https://doi.org/10.70267/aitia.2026300307

AI-Assisted Software Testing: Opportunities and Challenges

Mingyang Peng*

Faculty of Computer Science and Information Technology, University of Malaya, Kuala Lumpur, Malaysia

**Corresponding author: Mingyang Peng.*

Abstract

Software testing has played an important role in checking software reliability, security, and quality. However, testing has remained one of the most costly and time-consuming parts of the software development life cycle, especially as modern systems have become larger and more frequently updated. Traditional manual testing depends on human experience, while automated testing still requires testers to write and maintain many test scripts. This paper reviewed the opportunities and challenges of AI-assisted software testing based on selected academic studies, technical reports, and industry examples. The review found that AI techniques, such as machine learning, search-based optimization, fuzzing, and large language models, can support testing tasks such as test generation, defect prediction, code analysis, test prioritization, and debugging. These applications may reduce manual effort, improve test coverage, and help teams identify risky components earlier. At the same time, the review identified several challenges, including limited data quality, weak generalization across projects, low interpretability, false positives, integration difficulties, and risks related to LLM outputs. The paper argues that AI is unlikely to fully replace human testers in the near future. Instead, AI should be used as an assistant that supports human judgment in software quality assurance.

Keywords

software testing, artificial intelligence, machine learning, test automation, defect prediction

1. Introduction

Software testing is an essential activity in software engineering because it helps developers find defects before software is delivered to users. If a defect is not detected, it may cause system failure, data loss, security problems, financial loss, or poor user experience. Since software is now widely used in areas such as banking, healthcare, transportation, education, and government services, effective testing has become more important. Testing is therefore not only a technical task, but also a quality assurance activity that protects users and organizations from preventable failures.

Although testing is important, it is often costly and time-consuming. Manual testing allows testers to use domain knowledge, creativity, and judgment, but it is difficult to scale when software systems become large. Traditional automated testing can run predefined test scripts quickly, but these scripts still need to be written, checked, and maintained by humans. When requirements change often, test scripts may become outdated and

require extra maintenance. In agile and DevOps environments, teams are expected to release software quickly while still maintaining quality. This creates a need for testing methods that are faster and more adaptive.

Artificial intelligence provides new ways to support software testing. AI systems can learn from past defect data, analyze source code, process requirement documents, generate test inputs, and support debugging. For example, machine learning can be used to predict defect-prone modules, search-based algorithms can help generate test inputs, and large language models can help developers write unit tests or explain failures [1, 2].

However, AI-assisted testing should not be seen as a solution to all testing problems. AI models may need large and clean datasets, and their results are sometimes difficult to explain. A model trained on one project may not work well on another project because programming languages, architectures, team practices, and defect patterns may be different. Companies may also face difficulties when adding AI-based tools into existing continuous integration and delivery pipelines. Therefore, the opportunities of AI-assisted testing need to be discussed together with its limitations.

The purpose of this paper is to examine how AI can support software testing and to identify the main challenges that limit its wider use. The paper first reviews related literature and the technical foundations of AI-assisted testing. It then discusses related literature, major application areas, key opportunities, and current challenges. Finally, it presents future directions and concludes with a balanced view of AI as a supportive tool rather than a complete replacement for human testers.

2. Research Background

Research on AI-assisted software testing has developed from early defect prediction models to newer deep learning and large language model approaches. Early studies mainly applied machine learning to software metrics, such as code complexity, previous defects, file changes, and developer activity. The goal was to estimate which modules were more likely to contain defects so that testers could use their effort more effectively. Survey work on machine learning testing shows that AI techniques have been applied to test input generation, test oracle construction, fault localization, and defect prediction [1].

Search-based software testing is another important research direction. In this area, testing is treated as an optimization problem. Algorithms try to find test inputs that increase coverage, expose failures, or meet specific test goals. Sapienz is a well-known example for Android application testing because it uses multi-objective search to generate test sequences for mobile applications [3]. Search-based software engineering studies also show how optimization methods can support test generation and test prioritization [4].

Fuzzing is closely related to automated testing because it generates many program inputs to find crashes, memory errors, or security weaknesses. Google ClusterFuzz shows how fuzzing can be used at a large scale in industry [5]. Automated whitebox fuzzing is also an important foundation because it uses program analysis to generate inputs that explore different execution paths [6]. These approaches are useful in security testing because unexpected input combinations may reveal serious vulnerabilities.

Natural language processing and large language models have become more relevant because many testing tasks start from human-written documents, such as requirements, user stories, issue reports, and documentation. NLP techniques can help extract test conditions from requirements and turn them into test scenarios. More recently, LLMs have been used to generate unit tests, explain error messages, summarize bug reports, and suggest possible fixes [2, 7]. However, generated tests and explanations still need human checking because they may look correct but fail to test the intended behavior.

Automated debugging and program repair are also important areas. Automated repair aims to generate possible patches after faults are detected. Research on automatic software repair has shown that these techniques can reduce debugging effort, but generated patches may overfit the available tests or introduce new errors [8, 9]. Therefore, AI-based repair should be treated as a source of suggestions rather than final solutions.

3. Literature Review

3.1 Test Case Generation

Test case generation is one of the most common applications of AI in software testing. Traditional test case design often depends on testers reading requirements and manually identifying possible inputs, expected outputs, boundary conditions, and negative scenarios. AI can support this process by analyzing requirements, source code, user stories, or execution traces. Search-based algorithms can generate inputs that aim to increase code coverage or trigger unusual behavior [3]. LLMs can also help draft unit tests from source code or short descriptions [2].

AI-generated test cases can reduce the first stage of manual work, especially when the system has many functions or changing requirements. However, generated tests still require human review. A test case may run successfully but fail to check the correct business rule. Therefore, AI can help with test design, but testers still need to judge whether the generated test is meaningful and related to user requirements. Previous studies suggest that AI-based test generation can reduce part of the manual test design effort, especially when applications contain many possible inputs or user actions. However, the quality of generated tests still depends on whether they check meaningful program behavior.

3.2 Defect Prediction

Defect prediction uses historical data to estimate which software modules are more likely to contain bugs. Common input features include lines of code, code complexity, previous defects, number of commits, change frequency, and developer activity. Machine learning models such as decision trees, random forests, support vector machines, and neural networks can learn from these features and classify files or components according to their defect risk [1, 10].

The value of defect prediction is that it helps testing teams prioritize their effort. If a project has limited time, testers can focus first on high-risk modules. This is useful in large projects where complete manual inspection is not realistic. However, defect prediction depends strongly on data quality. If historical defect data is incomplete, biased, or inconsistent, the model may produce unreliable predictions. Research on defect prediction shows that machine learning can help testers identify risky modules earlier. This is useful because testing teams often have limited time and need to decide which parts of the system should be tested first.

3.3 Code Analysis and Security Testing

AI can also be used to analyze source code and detect patterns related to defects or vulnerabilities. Traditional static analysis tools depend on manually written rules. These rules can be useful for known problems, but they may be limited when defects are complex or depend on context. Learning-based models can identify suspicious patterns and rank warnings according to likely risk. Fuzzing techniques can also generate unexpected inputs that expose crashes or security-related failures [5, 6].

However, AI-based code analysis may generate false positives. Developers may lose trust in a tool if it reports too many warnings that are not real defects. Therefore, AI-based analysis should provide clear explanations and practical ranking methods so that developers can focus on the most important warnings. Studies on fuzzing and security testing show that automated input generation can discover crashes and vulnerabilities that may be missed by normal test cases. However, the reported issues still need to be checked and prioritized by developers.

3.4 Test Prioritization and Regression Testing

Regression testing checks whether new changes break existing functions. In large software systems, running all regression tests after every change may take too much time. AI can help prioritize test cases by predicting which tests are more likely to find faults based on previous results, code coverage, changed files, and dependency information. This allows development teams to run the most important tests earlier, which is useful in continuous integration environments.

The main challenge is that test prioritization models may become outdated when the software architecture changes. A model that worked well in one release may perform poorly in later releases if dependencies, features, or user behavior change. Therefore, continuous monitoring and retraining are needed.

3.5 Automated Debugging and Program Repair

AI can support debugging by locating suspicious code, explaining possible causes of failures, and generating possible patches. Automated program repair aims to produce changes that fix failing tests. Large language models can also assist by explaining stack traces, suggesting test inputs, and summarizing bug reports. These functions can reduce the time developers spend on understanding failures [2, 8, 9].

However, automated repair is still risky. A generated patch may pass existing tests but still be logically incorrect. This is sometimes called overfitting to the test suite. As a result, AI-generated patches should be treated as suggestions rather than final solutions. Human developers must review them carefully and add more tests when necessary. Research on automated repair shows that AI can suggest possible fixes and reduce debugging effort. However, generated patches may only pass existing tests without fully correcting the real problem, so human review remains necessary.

4. Key Finding

AI-assisted testing provides several opportunities for software development teams. First, it can reduce manual effort by automating repeated tasks such as generating test inputs, classifying bug reports, and selecting regression tests. This allows testers to spend more time on exploratory testing, risk analysis, and user-centered evaluation. Second, AI can improve testing coverage by generating inputs that humans may not think of, especially in complex systems with many possible states.

Third, AI can support faster feedback in agile and DevOps environments. When AI tools are added to CI/CD pipelines, they can analyze code changes, identify high-risk files, prioritize tests, and detect possible defects before deployment. This helps teams find problems earlier, when they are usually cheaper and easier to fix. Fourth, AI can support decision-making by giving testers and developers extra information, such as defect probability, failure patterns, or areas of weak coverage.

Fifth, AI can help with software security testing. Fuzzing tools and vulnerability detection models can generate unusual inputs and detect crashes or suspicious behavior [5, 6]. This is useful because security vulnerabilities often come from unexpected combinations of inputs or states. Finally, large language models can make testing easier by helping developers write unit tests, explain test failures, and understand unfamiliar codebases. These benefits suggest that AI can become an important part of software quality assurance when it is used carefully.

Table 1: Summary of AI applications, benefits, and limitations

AI application	Main benefit	Typical input data	Main limitation
Test case generation	Reduces manual test design effort	Requirements, code, models, execution traces	Generated tests may need human review
Defect prediction	Identifies high-risk modules earlier	Historical defects, code metrics, commits	Requires reliable project history
Code analysis	Detects hidden bug or vulnerability patterns	Source code, syntax trees, embeddings	May produce false positives
Regression test prioritization	Runs important tests earlier	Test history, coverage, changed files	May become outdated after project changes
Fuzzing	Discovers crashes and security issues	Program inputs, grammar, seed corpus	May require resources and triage effort
Automated debugging	Suggests fault locations and possible fixes	Failing tests, logs, source code	Generated fixes may be incorrect

5. Discussion

5.1 Challenges and Limitations

5.1.1 Data Quality and Availability

Many AI models require large amounts of training data. In software testing, such data may include defect labels, test results, code coverage, execution logs, and change history. However, many projects do not have clean and complete datasets. Defect data may be missing because not all bugs are reported properly. Some bug reports may be duplicated, incomplete, or linked to the wrong code changes. If the training data is poor, the AI model may learn misleading patterns and produce unreliable results.

Data availability is especially difficult for small projects, new projects, or private industrial systems where data cannot be shared because of confidentiality. This limits reproducibility and makes it hard to compare AI testing tools fairly. Transfer learning and cross-project defect prediction may help, but they still face problems because different projects may have different coding styles, architectures, and development processes.

5.1.2 Interpretability and Trust

Trust is a major issue in AI-assisted testing. Developers and testers may hesitate to accept AI predictions if they do not understand how the model reached a conclusion. For example, if an AI tool labels a file as defect-prone, developers may want to know whether the decision is based on code complexity, recent changes, previous bug history, or another factor. Without explanation, AI output may be ignored even if the model is accurate.

Explainable AI is therefore important. A useful AI testing tool should not only provide a prediction but also explain the reason behind it. This can include highlighting suspicious code areas, showing related historical defects, ranking test cases by risk, or explaining which requirement condition led to a generated test. Better explanations can improve trust and help developers use AI suggestions more effectively.

5.1.3 Generalization Across Projects

An AI model that performs well on one dataset may not perform equally well on another project. This is because software projects differ in programming language, domain, architecture, testing culture, and defect patterns. For example, a model trained on web applications may not work well for embedded systems. Similarly, a defect prediction model trained on open-source projects may not fit a private enterprise project with different development practices.

This generalization problem is a serious limitation for practical use. If companies need to retrain and adjust models for every project, the cost of using AI-assisted testing may become high. Future research should therefore focus on models that can adapt to different projects and provide reliable performance under changing conditions.

5.1.4 Integration with Existing Workflows

Even if an AI model performs well in experiments, it may be difficult to add it into real software development workflows. Development teams already use tools such as Git, GitHub, GitLab, Jira, Jenkins, testing frameworks, and code review platforms. An AI testing tool must fit into these workflows without creating too much extra work. If the tool requires complex configuration or produces too many alerts, developers may stop using it.

Integration also involves organizational issues. Teams need to decide who is responsible for reviewing AI-generated tests, validating AI predictions, maintaining training data, and monitoring model performance. These tasks require collaboration between testers, developers, data scientists, and project managers. Therefore, AI-assisted testing is not only a technical issue, but also a process and management issue.

5.1.5 Practical Risks of LLM-Based Testing

Large language models introduce additional concerns. They may generate code that looks correct but contains hidden errors, weak assertions, or insecure practices. They may also produce explanations that sound confident but are inaccurate. If developers rely on LLM output without checking it, the quality of testing may

decrease instead of improve. There may also be privacy concerns if proprietary source code or bug reports are sent to external AI services.

To use LLMs carefully, organizations should apply human review, avoid sharing sensitive information without permission, and evaluate generated tests using coverage, mutation testing, and real defect detection results. LLMs are useful assistants, but they should not be treated as fully reliable testers.

5.2 Future Directions

The findings suggest that AI-assisted testing is most useful when it supports human testers rather than replacing them. AI is strong at processing large amounts of data, identifying patterns, generating candidate inputs, and automating repeated tasks. Human testers, however, remain important for understanding business requirements, evaluating user experience, judging risk, and deciding whether a test result is meaningful. Therefore, the future of software testing is likely to involve cooperation between AI tools and human professionals.

One important future direction is explainable AI. Testing tools should provide clear reasons for their predictions and recommendations. For example, a defect prediction tool should explain which code metrics or historical patterns contributed to a risk score. A test generation tool should explain which requirement or code branch a generated test is intended to cover. Such explanations can help developers trust the tool and use it more effectively.

Another direction is hybrid testing. AI should be combined with traditional testing techniques such as unit testing, integration testing, static analysis, model-based testing, and exploratory testing. Hybrid approaches can reduce the weaknesses of purely AI-based methods. For example, rule-based static analysis can detect known security patterns, while machine learning can rank or filter warnings based on risk. Similarly, AI-generated tests can be reviewed and improved by human testers.

More real-world evaluation is also needed. Many AI testing studies report good results on benchmark datasets, but real industry environments are more complex. Future studies should examine how AI testing tools perform across different programming languages, project sizes, development processes, and business domains. Long-term studies are also needed to evaluate whether AI tools remain useful after software changes over several releases.

Finally, education and skills are important. Testers and developers will need basic knowledge of AI concepts, data quality, model evaluation, and prompt design for LLM-based tools. At the same time, AI specialists need to understand software engineering practices. This combination of skills will help teams use AI tools carefully and effectively.

6. Conclusion

This paper has reviewed the opportunities and challenges of AI-assisted software testing by examining selected academic studies, technical reports, and industry examples. The review has shown that AI can support many testing activities, including test case generation, defect prediction, code analysis, regression test prioritization, fuzzing, and automated debugging. These applications can improve efficiency, increase coverage, and provide faster feedback during software development. Industry examples such as Sapienz and ClusterFuzz show that AI-related testing techniques can be applied beyond small research prototypes [3, 5].

At the same time, AI-assisted testing faces several limitations. AI models often require high-quality data, and their predictions may not work well across different projects. Many models are difficult to explain, which can reduce trust among developers and testers. Practical integration into existing workflows can also be challenging. Large language models bring new opportunities, but they also bring risks such as inaccurate outputs, weak tests, and privacy concerns.

Overall, AI should be viewed as a useful assistant rather than a full replacement for human testers. The most practical approach is to combine AI capabilities with human judgment, traditional testing methods, and clear validation processes. Future research should focus on explainability, generalization, hybrid testing

methods, and real-world evaluation. If these challenges are addressed, AI-assisted testing can become an important part of modern software quality assurance.

References

[1] J. M. Zhang, M. Harman, L. Ma, and Y. Liu, "Machine Learning Testing: Survey, Landscapes and Horizons," IEEE Transactions on Software Engineering, vol. 48, no. 1, pp. 1–36, 2022, doi: 10.1109/TSE.2019.2962027.

[2] J. Wang, Y. Huang, C. Chen, Z. Liu, S. Wang, and Q. Wang, "Software Testing with Large Language Models: Survey, Landscape, and Vision," IEEE Transactions on Software Engineering, 2024, doi: 10.1109/TSE.2024.3368208.

[3] K. Mao, M. Harman, and Y. Jia, "Sapienz: Multi-objective Automated Testing for Android Applications," in Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA), 2016, pp. 94–105, doi: 10.1145/2931037.2931054.

[4] M. Harman, S. A. Mansouri, and Y. Zhang, "Search-Based Software Engineering: Trends, Techniques and Applications," ACM Computing Surveys, vol. 45, no. 1, Article 11, pp. 1–61, 2012, doi: 10.1145/2379776.2379787.

[5] Google, "ClusterFuzz: Scalable Fuzzing Infrastructure," Google Open Source Documentation. Available: https://google.github.io/clusterfuzz/

[6] P. Godefroid, M. Y. Levin, and D. Molnar, "Automated Whitebox Fuzz Testing," in Proceedings of the Network and Distributed System Security Symposium (NDSS), 2008.

[7] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, and H. Wang, "Large Language Models for Software Engineering: A Systematic Literature Review," ACM Transactions on Software Engineering and Methodology, vol. 33, no. 8, Article 220, 2024, doi: 10.1145/3695988.

[8] M. Monperrus, "Automatic Software Repair: A Bibliography," ACM Computing Surveys, vol. 51, no. 1, Article 17, pp. 1–24, 2018, doi: 10.1145/3105906.

[9] M. Tufano, C. Watson, G. Bavota, M. Di Penta, M. White, and D. Poshyvanyk, "An Empirical Study on Learning Bug-Fixing Patches in the Wild via Neural Machine Translation," ACM Transactions on Software Engineering and Methodology, vol. 28, no. 4, Article 19, pp. 1–29, 2019, doi: 10.1145/3340544.

[10] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell, "A Systematic Literature Review on Fault Prediction Performance in Software Engineering," IEEE Transactions on Software Engineering, vol. 38, no. 6, pp. 1276–1304, 2012, doi: 10.1109/TSE.2011.103.

Funding

This research received no external funding.

Conflicts of Interest

The authors declare no conflict of interest.

Acknowledgment

This paper is an output of the science project.