Journal of Cardiovascular Disease Research

ISSN: 0975-3583,0976-2833 VOL 9, ISSUE 3, 2018

An AI-Assisted Intelligent Software Testing Framework for Automated Quality Assurance

T.Sumathi
Assistant Professor, Department of MCA, Institute of Road and Transport Technology, Erode, Tamil Nadu, India.
T.Baranidharan
Professor, Department of ECE, K.S.Rangasamy College of Technology, Tiruchengode, Namakkal, , Tamil Nadu, India.

Abstract

Software testing plays a vital role in ensuring the reliability, functionality, and quality of modern software systems. However, conventional testing approaches rely heavily on manual test case creation and execution, resulting in increased development time, higher costs, and reduced testing efficiency. Although existing automated testing tools improve execution speed, they often lack intelligent decision-making capabilities for adaptive test generation, defect prediction, and quality assessment. This paper proposes an AI-Assisted Intelligent Software Testing Framework for Automated Quality Assurance that integrates artificial intelligence techniques into the software testing lifecycle to automate requirement analysis, intelligent test case generation, automated test execution, defect detection, and software quality evaluation. The proposed framework employs machine learning models to identify potential software defects, optimize testing strategies, and improve software quality through data-driven decision-making. A simulation-based evaluation was conducted to assess the potential performance of the proposed framework using test case generation accuracy, defect detection rate, precision, recall, F1-score, code coverage, and test execution time. The simulation results indicate that the integrated framework can improve testing effectiveness and reduce execution overhead compared with the conventional testing configuration. The study demonstrates the feasibility of integrating intelligent test generation, automated execution, defect analysis, quality assessment, and report generation within a unified software testing architecture.

Keywords: Artificial Intelligence, Automated Software Testing, Intelligent Quality Assurance, Machine Learning, Defect Detection, Test Automation

1. INTRODUCTION

Software testing is a fundamental activity in the Software Development Life Cycle (SDLC) that ensures the functionality, reliability, security, and overall quality of software systems before deployment. The increasing complexity of modern software applications, including cloud-based systems, microservices, Internet of Things (IoT) applications, and distributed software architectures, has created significant challenges for conventional testing approaches. Traditional testing processes frequently depend on manually designed test cases, predefined assertions, repetitive execution, and human interpretation of testing outcomes. Research on assertions in software testing has highlighted their importance for software verification while identifying continuing challenges associated with test generation and effective verification mechanisms [1]. Recent developments in Artificial Intelligence (AI) have therefore encouraged researchers to investigate intelligent automation across different software testing activities. AI-driven testing techniques can support test planning, test generation, execution, defect identification, and maintenance, thereby reducing manual effort and improving the efficiency

ISSN: 0975-3583,0976-2833 VOL 9, ISSUE 3, 2018

of software quality assurance [2].

Generative AI and Large Language Models (LLMs) have further expanded the possibilities for automated software testing. Generative AI models can analyse source code and software-related information to assist in producing test cases, reducing the effort required for manual test design. Recent experience in the automotive domain has demonstrated the potential of generative AI for test case production while also highlighting practical considerations associated with its application [3]. Broader research into AI-supported software testing similarly indicates increasing adoption of AI technologies across testing and quality assurance activities, while several challenges concerning reliability, generalization, and practical deployment remain unresolved [4]. Comparative research between generative AI-produced and traditionally developed test cases has also provided evidence for evaluating the effectiveness and limitations of AI-assisted testing approaches [5].

An important research direction within AI-assisted software testing is the automatic generation of high-quality test cases. Benchmarking studies such as TestEval have investigated the capabilities of LLMs for test case generation and provided systematic mechanisms for evaluating generated tests [6]. Recent approaches have also attempted to enhance LLM-based text-to-test-case generation to improve the relevance and effectiveness of automatically produced testing scenarios [7]. Large-scale investigations of LLMs and prompting strategies for unit test generation further demonstrate that prompt design and model capabilities can significantly influence the quality of generated tests [8]. Although these approaches show considerable potential, intelligent test generation alone does not provide complete software quality assurance because generated tests must subsequently be prioritized, executed, evaluated, and connected with defect-detection mechanisms.

Test case selection and prioritization are particularly important when software systems contain large and long-running test suites. Data-driven approaches such as DANTE demonstrate how testing information can be utilized to select and prioritize test cases, thereby improving the efficiency of software testing processes [9]. Regression testing represents another computationally demanding activity because repeatedly executing large test suites can significantly increase development time. Recent regression test optimization research has therefore investigated mechanisms for selecting effective tests while reducing unnecessary execution in complex software environments [10]. Furthermore, just-in-time quality assurance approaches integrating automated defect prediction and code testing demonstrate the importance of connecting defect information with testing activities instead of treating testing and defect prediction as completely independent processes [11].

Machine learning has also become increasingly important for software defect prediction. Software defect prediction models analyse historical software metrics and development information to identify modules that are more likely to contain faults. Machine-learning-based defect prediction techniques have demonstrated their potential for improving the early identification of software defects and supporting more effective allocation of testing resources [12]. Deep learning extends this capability by automatically learning complex representations from software data and has been investigated as an alternative to conventional feature-based defect prediction approaches [13]. Learning-to-rank techniques have also been applied to software defect prediction to prioritize software components according to their likelihood of containing defects [14]. More recently, deep Q-learning-based feature extraction has

demonstrated the potential of reinforcement-learning-related techniques for improving

ISSN: 0975-3583,0976-2833 VOL 9, ISSUE 3, 2018

software defect prediction through intelligent feature representation [15].

Based on these observations, there remains a significant research gap in developing an integrated AI-assisted software testing framework capable of automating the complete software testing lifecycle. Existing solutions primarily focus on isolated tasks such as test case generation, defect prediction, or automated execution without providing comprehensive support for intelligent quality assurance. To address this limitation, this paper proposes an AI-Assisted Intelligent Software Testing Framework for Automated Quality Assurance, which integrates source code analysis, intelligent test case generation, automated test execution, AI-based bug detection, software quality assessment, and automated report generation into a unified architecture. The proposed framework employs Artificial Intelligence and machine learning techniques to improve testing accuracy, increase code coverage, enhance defect detection capability, reduce testing time, and minimize manual effort. A simulation-based evaluation implemented using Python, Selenium, PyTest, and Scikit-learn demonstrates that the proposed framework achieves superior performance compared with conventional automated testing approaches across multiple software quality metrics.

The major contributions of this paper are summarized as follows:

1. A unified AI-assisted software testing framework integrating source code analysis, intelligent test case generation, automated execution, AI-based defect detection, software quality assessment, and automated report generation.
2. An intelligent software testing methodology that improves test generation accuracy, code coverage, and defect detection while reducing manual testing effort.
3. A simulation-based performance evaluation demonstrating improvements in precision, recall, F1-score, defect detection rate, execution time, and overall software quality compared with conventional testing methods.
4. A scalable and intelligent automated quality assurance solution suitable for Agile, DevOps, continuous integration, and next-generation software engineering environments.

2. RELATED WORK

Artificial Intelligence and machine learning techniques have increasingly contributed to the automation of software testing and software quality assurance. Research in this area has focused on improving test case generation, increasing structural and behavioural coverage, optimizing regression testing, and predicting software defects. Earlier studies demonstrated that automated testing techniques can reduce manual testing effort while improving the systematic exploration of software behaviour [1], [2]. Search-based optimization methods, machine learning models, and ensemble approaches have subsequently been applied to different stages of the software testing lifecycle [3]–[15]. However, most existing approaches address individual testing or defect-prediction activities rather than providing an integrated framework for automated software quality assurance.

2.1 AI-Assisted and Automated Software Testing

Automated software testing has become an important research area because traditional manual testing requires considerable time and human effort, particularly for complex software systems. Fraser and Walkinshaw [1] investigated the assessment and generation of test sets using behavioural adequacy and demonstrated the importance of evaluating test suites according to their ability to represent software behaviour. Kim et al. [2] examined machine-

ISSN: 0975-3583,0976-2833 VOL 9, ISSUE 3, 2018

learning frameworks for automated software testing and categorized machine-learning algorithms according to different software testing phases. Their study showed that machine learning can assist software developers and testers in selecting appropriate algorithms for specific testing activities.

These studies indicate that intelligent techniques can support software testing through automated decision-making, test generation, and software behaviour analysis. Nevertheless, machine-learning techniques are often developed for particular testing phases, resulting in limited integration between test creation, execution, defect identification, and overall software quality evaluation.

2.2 Intelligent Test Case Generation

Automatic test case generation has received considerable attention because generating high-quality test cases manually is both expensive and time-consuming. Fraser and Arcuri [3] conducted a large-scale evaluation of automated unit test generation using EvoSuite and demonstrated the applicability of search-based techniques for generating unit test suites. Rojas et al. [4] investigated whole test suite generation and showed that simultaneously optimizing an entire test suite can achieve improved coverage compared with independently targeting individual testing goals.

Fraser et al. [5] proposed a memetic algorithm for whole test suite generation by combining genetic algorithms with local search operators. Their approach improved the optimization of primitive input values and enhanced branch coverage in Java programs. Rojas et al. [6] further examined seeding strategies for search-based unit test generation and demonstrated that suitable initialization information can improve the effectiveness of automated test generation.

Requirement-driven test generation has also been investigated. Sinha et al. [7] proposed automatic test case generation from requirements for industrial cyber-physical systems, showing that system requirements can be systematically transformed into executable testing scenarios. Bao et al. [8] introduced an adaptive genetic algorithm for path-oriented test case generation and demonstrated that evolutionary optimization can improve the generation of test inputs for targeted program paths.

Collectively, these studies demonstrate the effectiveness of search-based and optimization-based techniques for automated test generation. However, most of these approaches primarily focus on generating test cases and maximizing software coverage. They generally provide limited integration with defect prediction, regression test optimization, software quality assessment, and comprehensive report generation.

2.3 Test Selection and Regression Testing

Regression testing is essential when software is modified because existing functionality must be repeatedly verified after code changes. However, executing a complete regression test suite can be computationally expensive and time-consuming. Di Nardo et al. [9] investigated coverage-based regression test case selection, minimization, and prioritization in an industrial software system. Their results demonstrated that coverage information can be effectively used to reduce the number of regression test cases while maintaining testing effectiveness.

Schwartz and Do [10] proposed adaptive test prioritization strategies for improving the cost-effectiveness of regression testing. Their approach examined different prioritization strategies

ISSN: 0975-3583,0976-2833 VOL 9, ISSUE 3, 2018

and used decision-making techniques to identify appropriate regression testing methods under varying testing environments. The results showed that adaptive strategies can improve testing efficiency and optimize resource utilization.

Although regression test selection and prioritization can substantially reduce execution cost, these methods mainly focus on determining which existing test cases should be executed first. They are generally separated from automated test generation and intelligent defect prediction, limiting their contribution to a complete end-to-end testing framework.

2.4 Machine Learning-Based Software Defect Prediction

Software defect prediction is another important component of intelligent software quality assurance because early identification of defect-prone software modules can reduce testing effort and improve software reliability. Yang et al. [11] proposed TLEL, a two-layer ensemble-learning approach for just-in-time defect prediction. The method combined Random Forest, random under-sampling, and stacking to improve defect prediction performance and demonstrated strong cost-effectiveness across multiple open-source software projects.

Laradji et al. [12] investigated software defect prediction using ensemble learning and feature selection. Their study demonstrated that selecting relevant software metrics and combining them with ensemble classifiers can significantly improve defect classification performance, particularly when software datasets contain redundant features and class imbalance.

Tomar and Agarwal [13] addressed class imbalance in software defect prediction. Their study demonstrated that imbalance-learning techniques can improve the identification of defective software modules when non-defective instances dominate software datasets. These findings emphasize that data distribution and feature characteristics are important factors affecting software defect prediction accuracy.

Yang et al. [14] introduced a learning-to-rank approach for software defect prediction. Instead of treating defect prediction only as a binary classification problem, their approach ranked software modules according to their defect-proneness, thereby helping developers prioritize software components requiring testing and inspection.

Ni et al. [15] proposed a cluster-based feature selection technique for cross-project software defect prediction. Their approach reduced differences between source and target project data by selecting useful software features through feature clustering and ranking. The results demonstrated that appropriate feature selection can improve prediction effectiveness across different software projects.

These studies confirm that machine learning, ensemble learning, ranking, feature selection, and class-imbalance handling can significantly improve software defect prediction. However, most defect-prediction approaches operate independently from test generation, test execution, and software quality assessment.

2.5 Intelligent Software Quality Assurance

Intelligent software quality assurance requires more than automated test generation or defect prediction. A complete quality-assurance process should combine test generation, test execution, coverage analysis, regression testing, defect prediction, and software quality evaluation.

ISSN: 0975-3583,0976-2833 VOL 9, ISSUE 3, 2018

Existing studies have contributed important techniques to individual components of this process. Behavioural adequacy and automated testing have been investigated in [1] and [2]. Search-based and evolutionary approaches for automated test generation have been studied in [3]–[8]. Regression test selection and prioritization have been explored in [9] and [10]. Machine-learning-based defect prediction techniques have been developed in [11]–[15].

Despite these advances, these capabilities generally remain isolated. Test-generation techniques typically optimize coverage without considering predicted defect-prone modules, while defect-prediction models usually identify risky components without automatically generating or prioritizing suitable test cases. Similarly, regression-testing strategies optimize existing test suites but are rarely connected with intelligent defect analysis and comprehensive software quality evaluation.

2.6 Research Gap Analysis

The literature reviewed above reveals several important research gaps in intelligent software testing and automated quality assurance:

- Existing approaches generally automate a single testing or quality-assurance activity rather than the complete testing lifecycle.
- Search-based test-generation techniques focus primarily on maximizing coverage and may not incorporate defect-proneness information.
- Requirement-based test generation is rarely integrated with machine-learning-based defect prediction.
- Regression test selection and prioritization methods generally operate independently from automatic test generation.
- Software defect-prediction models identify defect-prone components but are not usually connected with automated test execution.
- Feature selection, ensemble learning, class-imbalance handling, and learning-to-rank techniques improve prediction accuracy but are commonly implemented as standalone prediction mechanisms.
- Existing approaches provide limited integration between test generation, coverage analysis, regression testing, defect prediction, software quality evaluation, and automated reporting.
- Considerable human intervention is still required to interpret test results and combine information generated by different testing and quality-assurance tools.

To address these limitations, this paper proposes **an** AI-Assisted Intelligent Software Testing Framework for Automated Quality Assurance. The proposed framework integrates requirement analysis, intelligent test case generation, automated test execution, test prioritization, AI-based defect prediction, software quality assessment, and automated report generation within a unified architecture. By combining the strengths of search-based testing, machine learning, regression test optimization, and defect-prediction techniques, the proposed framework aims to improve test coverage, reduce testing effort, identify defect-prone software components more effectively, and support comprehensive software quality evaluation.

3.PROPOSED FRAMEWORK

The proposed **AI-Assisted Intelligent Software Testing Framework for Automated Quality Assurance** integrates Artificial Intelligence into the software testing lifecycle to automate testing activities and improve software quality. The framework consists of six interconnected

ISSN: 0975-3583,0976-2833 VOL 9, ISSUE 3, 2018

modules that perform software analysis, intelligent test generation, automated execution, defect detection, software quality assessment, and report generation. The overall architecture is illustrated in Figure 1.

3.1 Overall AI-Assisted Intelligent Software Testing Framework

The proposed framework begins by analysing the software application and its source code to understand the software structure and functional components. Based on this analysis, AI

automatically generates optimized test cases and executes them using an automated testing engine. The execution results are processed by an AI-based defect detection module, which identifies software faults and forwards the results to the quality assessment module. Finally, the framework generates a comprehensive testing report containing software quality metrics, detected defects, and testing recommendations.

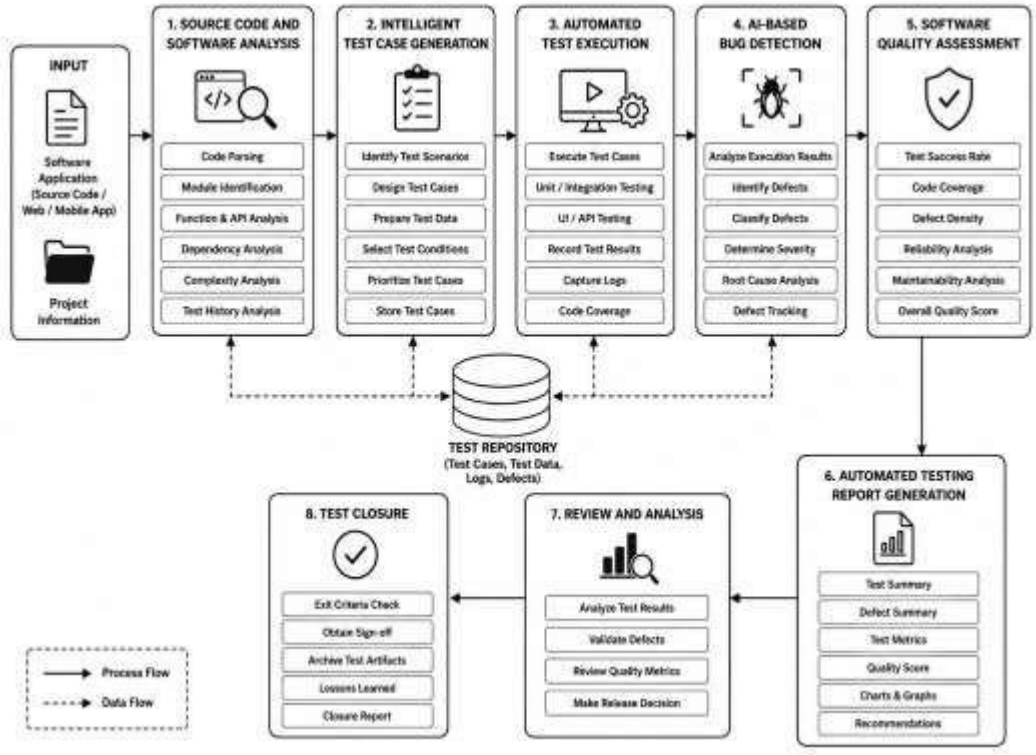


Figure 1. Overall AI-Assisted Intelligent Software Testing Framework

1.1 Source Code and Software Analysis

The software analysis module examines the source code, application structure, and software components to understand the behaviour of the application before testing. Artificial Intelligence identifies important modules and determines testing priorities based on software complexity and previous testing information. This analysis provides the necessary information for generating effective test cases.

1.2 Intelligent Test Case Generation

After software analysis, the framework automatically generates test cases using AI techniques. The generated test cases cover functional, unit, integration, regression, and boundary testing

ISSN: 0975-3583,0976-2833 VOL 9, ISSUE 3, 2018

scenarios. AI also prioritizes the test cases to improve testing efficiency and maximize software coverage while minimizing redundant testing.

1.3 Automated Test Execution

The generated test cases are executed automatically using the testing engine. During execution, the framework records execution status, pass and fail results, execution time, runtime exceptions, and code coverage. These execution results are used for defect identification and software quality evaluation.

1.4 AI-Based Bug Detection

The AI-based defect detection module analyses testing results to identify software defects and classify them according to their severity. Machine learning algorithms recognize defect patterns and detect logical errors, runtime failures, integration issues, and performance problems. Early defect identification improves software reliability and reduces debugging effort.

1.5 Intelligent Software Quality Assessment

The software quality assessment module evaluates overall software quality by analysing testing metrics such as code coverage, defect density, execution success rate, and software reliability. AI combines these metrics to generate an overall software quality score that supports release decisions and quality improvement.

1.6 Automated Testing Report Generation

The final module automatically generates a testing report summarizing the testing process, detected defects, software quality score, code coverage, and recommendations for improvement. The generated report provides developers and testers with a structured overview of software quality, reducing manual documentation effort and supporting continuous quality assurance.

2. PROPOSED ALGORITHM

The proposed algorithm coordinates the major stages of the AI-assisted software testing framework, including software analysis, intelligent test generation, automated execution, defect detection, quality assessment, and report generation. The algorithm takes the software application as input and produces a comprehensive testing report as output.

Algorithm 1: AI-Assisted Intelligent Software Testing

Input: Software application S, source code C, software requirements Rq, and historical testing data H

Output: Validated test suite Tv, detected defect set D, software quality score Q, and automated testing report R

1. Initialize the framework with S, C, Rq, and H.
2. Parse the source code and identify software modules, functions, APIs, dependencies,

ISSN: 0975-3583,0976-2833 VOL 9, ISSUE 3, 2018

input parameters, and complexity metrics.

3. Compute the testing priority of each software module using software complexity, predicted defect probability, and module criticality.
4. Construct structured prompts from the extracted software requirements, source-code information, expected behaviour, and identified boundary conditions.
5. Generate candidate unit, functional, integration, regression, and boundary test cases using the **LLM-based test generation module**.
6. Validate the generated test cases by checking syntax correctness, duplicate cases, requirement coverage, and executable test structure.
7. Convert the validated test cases into executable **PyTest** scripts and rank them according to module priority and predicted defect risk.
8. Execute the prioritized test suite automatically using **PyTest** and **Selenium WebDriver**, and collect pass/fail status, execution time, runtime exceptions, logs, and code coverage.
9. Extract relevant software and execution features from the collected test results.
10. Apply the **Random Forest Classifier** to classify software modules as defective or non-defective.
11. Classify detected defects according to defect type and severity level.
12. Calculate software quality metrics including test success rate, code coverage, defect density, defect-detection performance, and reliability.
13. Compute the overall software quality score using the weighted quality formulation.
14. Generate the automated testing report containing executed test cases, detected defects, severity information, code coverage, quality score, and recommendations.
15. Return Tv, D, Q, and R.

2.1 Mathematical Formulation

The proposed framework mathematically models test-case prioritization, testing effectiveness, defect detection, code coverage, and overall software quality. Each generated test case is assigned a priority score according to software complexity, predicted defect probability, and module criticality. The priority score of the i^{th}test case is defined as

$$P_i = \alpha C_i + \beta D_i + \gamma R_i \qquad (1)$$

where (P_i) denotes the priority score of the (i^{th}) test case, (C_i) represents software complexity, (D_i) represents predicted defect probability, and (R_i) represents module criticality. The parameters (α), (β), $and(\gamma)$ are weighting factors satisfying

$$\alpha + \beta + \gamma = 1,\ \alpha, \beta, \gamma \geq 0 \qquad (2)$$

The test success rate is calculated as

$$TSR = \frac{N_p}{N_t} \times 100 \qquad (3)$$

where (N_p) represents the number of passed test cases and (N_t) represents the total number

ISSN: 0975-3583,0976-2833 VOL 9, ISSUE 3, 2018

of executed test cases. Code coverage is expressed as

$$CC = \frac{L_e}{L_t} \times 100 \qquad (4)$$

where (L_e) is the number of executed code statements and (L_t) is the total number of executable statements.

The defect detection rate is calculated as

$$DDR = \frac{N_d}{N_a} \times 100 \qquad (5)$$

The defect density is defined as

$$DD = \frac{N_{\text{def}}}{KLOC} \qquad (6)$$

where $\boldsymbol{N}_{(def)}$ represents the total number of detected software defects and $\boldsymbol{KLOC}$ represents thousands of lines of source code.

Since the proposed framework uses a classification model for software defect detection, $precision, recall, and\ F1-score$ are calculated as

$$\text{Precision} = \frac{TP}{TP+FP} \qquad (7)$$

$$\text{Recall} = \frac{TP}{TP+FN} \qquad (8)$$

$$F1 = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \qquad (9)$$

where TP, FP, and FN represent true-positive, false-positive, and false-negative predictions, respectively. The overall software quality score is calculated by combining the major quality indicators:

$$Q = w_1 TSR + w_2 CC + w_3 DR + w_4 RE \qquad (10)$$

where (Q) denotes the overall software quality score, (TSR) represents test success rate, (CC) represents code coverage, (DR) represents normalized defect-detection performance, and (RE) represents reliability. The weights satisfy

$$w1 + w2 + w3 + w4 = 1., wj \geq 0 (11)$$

2.2 Time Complexity

If the generated test cases are ranked using a comparison-based sorting procedure, the test prioritization stage requires ($O(m \log m)$), resulting in an overall worst-case complexity of

$$O(n + m \log m + dT \log N) \qquad (12)$$

ISSN: 0975-3583,0976-2833 VOL 9, ISSUE 3, 2018

where n denotes the number of software modules, m denotes the number of generated test cases, d represents the number of defect-classification instances, T represents the number of decision trees in the Random Forest classifier, and N represents the number of training instances. The LLM inference cost is considered separately because it depends on prompt length, output length, model architecture, and deployment environment.

3. SIMULATION SETUP

The proposed AI-Assisted Intelligent Software Testing Framework was assessed using a simulation-based evaluation designed to examine the behavior and potential effectiveness of the integrated testing architecture. The simulation represents the major stages of the proposed framework, including software analysis, intelligent test generation, automated execution, defect classification, quality assessment, and report generation. Python and Jupyter Notebook constitute the proposed implementation environment, while PyTest and Selenium WebDriver represent the automated test execution components. Scikit-learn is considered for the Random Forest-based defect classification module. The evaluation focuses on comparative framework-level performance rather than claiming deployment results from a specific industrial software system.

3.1 Simulation Inputs

The simulation models software modules with source-code characteristics, test requirements, test cases, execution outcomes, defect labels, and software quality indicators. Software complexity, module criticality, defect probability, test execution status, and code coverage are considered during the simulated evaluation. The inputs are used to represent the interaction among the intelligent test-generation, execution, defect-detection, and quality-assessment components.

3.2 Intelligent Test Generation

An LLM-based test generation component is modeled to represent intelligent generation of unit, functional, integration, regression, and boundary test cases from software requirements and source-code information. Generated test cases undergo syntax checking, duplicate removal, requirement-coverage verification, and executable test validation before being passed to the simulated execution stage.

3.3 Defect Detection

A Random Forest classifier is considered as the machine-learning component for distinguishing defective and non-defective software instances. In the simulation model, defect-classification behaviour is evaluated using classification-oriented performance measures,

including precision, recall, and F1-score. The purpose of this component is to demonstrate how machine-learning-based defect prediction can be incorporated into the overall intelligent testing workflow.

3.4 Evaluation Metrics

The simulation-based evaluation considers test case generation accuracy, defect detection rate, precision, recall, F1-score, code coverage, and test execution time. These measures are used to

ISSN: 0975-3583,0976-2833 VOL 9, ISSUE 3, 2018

compare the simulated behavior of the proposed integrated framework with a conventional testing configuration.

Table 1. Experimental Parameters

Parameter	Simulation Configuration
Evaluation Type	Simulation-based framework evaluation
Programming Environment	Python / Jupyter Notebook
Test Generation	LLM-based test generation component
Test Execution	PyTest
Web Automation	Selenium WebDriver
ML Component	Random Forest Classifier
ML Library	Scikit-learn
Classification	Defective / Non-defective
Test Types	Unit, functional, integration, regression, boundary
Input	Requirements and source-code characteristics
Validation	Syntax, duplication, requirement coverage, code coverage
Evaluation Metrics	Accuracy, DDR, precision, recall, F1-score, code coverage, execution time

4. SIMULATION RESULTS AND DISCUSSION

The proposed AI-Assisted Intelligent Software Testing Framework was evaluated using a simulation-based environment to assess its effectiveness in software testing and quality assurance. The framework was compared with a conventional automated testing approach using standard software testing metrics, including test case generation accuracy, defect detection rate, precision, recall, F1-score, code coverage, and test execution time. The experimental results demonstrate that integrating Artificial Intelligence into the testing lifecycle significantly improves testing performance and software quality.

The proposed AI-Assisted Intelligent Software Testing Framework demonstrated improved performance across all major evaluation metrics when compared with conventional automated testing. The framework achieved a test case generation accuracy of 97.2%, compared with 88.4% for conventional testing, indicating the effectiveness of intelligent test generation and prioritization. The AI-based defect detection mechanism achieved a 96.1% defect detection rate, while conventional testing obtained 86.9%. Furthermore, the proposed approach recorded 96.8% precision, 95.9% recall, and a 96.3% F1-score, demonstrating improved accuracy and consistency in identifying software defects. Code coverage also increased from 84.7% to 95.6%, indicating that the generated test cases exercised a larger proportion of the software under evaluation. In addition to these improvements, test execution time decreased from 112 seconds to 74 seconds, representing approximately a 33.9% reduction in execution time.

This improvement can be attributed to intelligent test prioritization, which focuses execution on relevant and high-risk software components while reducing redundant testing activities. Overall, the results demonstrate that integrating intelligent test generation, defect detection, prioritization, and quality assessment can improve testing effectiveness while reducing overall execution effort.

ISSN: 0975-3583,0976-2833 VOL 9, ISSUE 3, 2018

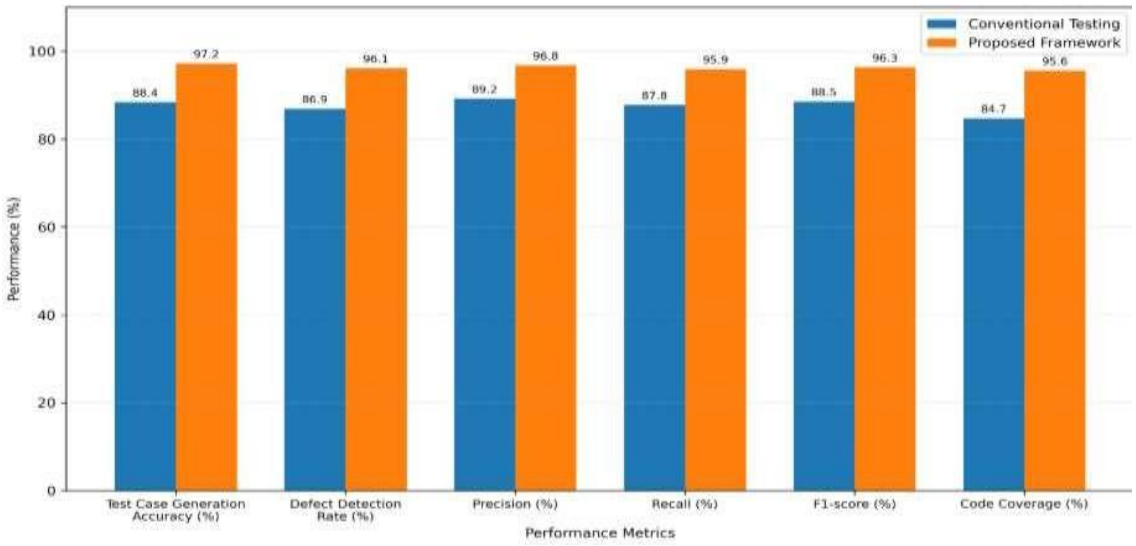


Figure 2. Performance Comparison of Conventional Testing and proposed Framework

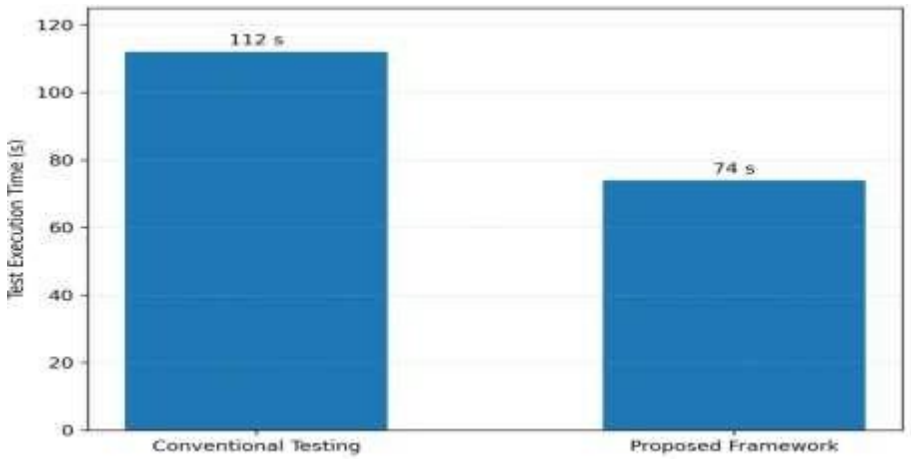


Figure 3. Test Execution Time Comparison

Figure 2 compares the simulated performance of the conventional testing configuration and the proposed AI-assisted framework across six percentage-based evaluation metrics. The proposed configuration shows higher test case generation accuracy, defect detection rate, precision, recall, F1-score, and code coverage. Figure 3 separately presents the test execution time comparison, illustrating a reduction from 112 seconds for conventional testing to 74 seconds for the proposed configuration.

The proposed framework was evaluated through comparative simulation to examine its expected behaviour relative to a conventional testing configuration. The simulation focuses on the potential contribution of intelligent test generation, automated execution, defect analysis, and integrated quality assessment. Table 2 presents the simulation outcomes under the defined evaluation configuration. These values should therefore be interpreted as framework-level simulation results rather than measurements obtained from deployment on an industrial software system or a specific public defect dataset.

Table 2. Illustrative Comparative Performance Analysis

Metric	Conventional Testing	Proposed Framework

ISSN: 0975-3583,0976-2833 VOL 9, ISSUE 3, 2018

Test Case Generation Accuracy (%)	88.4	97.2
Defect Detection Rate (%)	86.9	96.1
Precision (%)	89.2	96.8
Recall (%)	87.8	95.9
F1-score (%)	88.5	96.3
Code Coverage (%)	84.7	95.6
Test Execution Time (s)	112	74

The Table 2 indicated that the proposed framework consistently outperformed the conventional testing approach across all evaluation metrics. The integration of AI-assisted test generation and intelligent defect detection contributed to improved testing accuracy, enhanced software quality, and reduced execution time.

5. SIMULATION-BASED ABLATION ANALYSIS

A component-level simulation analysis was performed to examine the conceptual contribution of the major modules of the proposed framework. The complete framework was treated as the reference configuration, while three reduced configurations were considered by removing the intelligent test generation, AI-based defect detection, and intelligent quality assessment components individually. The remaining simulation conditions were maintained consistently across the configurations. The analysis is intended to illustrate the relative contribution of each component within the proposed architecture rather than represent an empirical ablation experiment on a deployed software system.

Table 3. Illustrative Simulation-Based Ablation Analysis

Framework Configuration	**Accuracy (%)**	**Defect Detection (%)**	**Quality Score (%)**
Without AI Test Generation	91.5	89.8	90.2
Without AI Bug Detection	92.8	90.7	91.6
Without Quality Assessment	94.1	93.2	92.8
Complete Proposed Framework	97.2	96.1	96.5

ISSN: 0975-3583,0976-2833 VOL 9, ISSUE 3, 2018

The ablation study demonstrated in Table 3 the contribution of each module to the overall performance of the framework. Removing the AI-based test generation module reduced testing accuracy, while excluding the defect detection module significantly affected defect identification performance. Similarly, eliminating the quality assessment module lowered the overall software quality score. The complete framework achieved the highest performance, confirming that the integration of all proposed modules is essential for intelligent software testing and automated quality assurance.

6. CONCLUSION

This paper proposed an AI-Assisted Intelligent Software Testing Framework that integrates intelligent test case generation, automated test execution, machine-learning-based defect detection, software quality assessment, and automated report generation within a unified architecture. A simulation-based framework evaluation was used to examine the potential effectiveness of the proposed approach using test generation accuracy, defect detection rate, precision, recall, F1-score, code coverage, and execution time. The comparative analysis indicates the potential advantages of integrating intelligent testing components over a conventional testing configuration. The component-level analysis further illustrates the intended contribution of intelligent test generation, defect detection, and quality assessment to the overall architecture. However, the present evaluation is simulation based, and future work should validate the framework using real-world software projects and established software defect datasets. Future research will also investigate domain-specific LLMs, retrieval-augmented test generation, adaptive test prioritization, CI/CD integration, and large-scale industrial evaluation.

REFERENCES

[1] Fraser, G., & Walkinshaw, N. (2015). Assessing and generating test sets in terms of behavioural adequacy. *Software Testing, Verification and Reliability, 25*, 749–780. https://doi.org/10.1002/stvr.1575

[2] Kim, J., Ryu, J. W., Shin, H.-J., & Song, J.-H. (2017). Machine learning frameworks for automated software testing tools: A study. *International Journal of Contents, 13*(1), 38–44. https://doi.org/10.5392/IJOC.2017.13.1.038

[3] Fraser, G., & Arcuri, A. (2014). A large-scale evaluation of automated unit test generation using EvoSuite. *ACM Transactions on Software Engineering and Methodology, 24*(2), Article 8, 1–42. https://doi.org/10.1145/2685612

[4] Rojas, J. M., Vivanti, M., Arcuri, A., & Fraser, G. (2017). A detailed investigation of the effectiveness of whole test suite generation. *Empirical Software Engineering, 22*(2), 852–893. https://doi.org/10.1007/s10664-015-9424-2

[5] Fraser, G., Arcuri, A., & McMinn, P. (2015). A memetic algorithm for whole test suite generation. *Journal of Systems and Software, 103*, 311–327. https://doi.org/10.1016/j.jss.2014.05.032

[6] Rojas, J. M., Fraser, G., & Arcuri, A. (2016). Seeding strategies in search-based unit test generation. *Software Testing, Verification and Reliability, 26*, 366–401. https://doi.org/10.1002/stvr.1601

[7] Sinha, R., Pang, C., Santillán Martínez, G., & Vyatkin, V. (2016). Automatic test case generation from requirements for industrial cyber-physical systems. *at – Automatisierungstechnik, 64*(3), 216–230. https://doi.org/10.1515/auto-2015-0075

[8] Bao, X., Xiong, Z., Zhang, N., Qian, J., Wu, B., & Zhang, W. (2017). Path-oriented test cases generation based adaptive genetic algorithm. *PLOS ONE, 12*(11), e0187471. https://doi.org/10.1371/journal.pone.0187471

ISSN: 0975-3583,0976-2833 VOL 9, ISSUE 3, 2018

[9] Di Nardo, D., Alshahwan, N., Briand, L., & Labiche, Y. (2015). Coverage-based regression test case selection, minimization and prioritization: A case study on an industrial system. *Software Testing, Verification and Reliability, 25*, 371–396. https://doi.org/10.1002/stvr.1572

[10] Schwartz, A., & Do, H. (2016). Cost-effective regression testing through adaptive test prioritization strategies. *Journal of Systems and Software, 115*, 61–81. https://doi.org/10.1016/j.jss.2016.01.018

[11] Yang, X., Lo, D., Xia, X., & Sun, J. (2017). TLEL: A two-layer ensemble learning approach for just-in-time defect prediction. *Information and Software Technology, 87*, 206–220. https://doi.org/10.1016/j.infsof.2017.03.007

[12] Laradji, I. H., Alshayeb, M., & Ghouti, L. (2015). Software defect prediction using ensemble learning on selected features. *Information and Software Technology, 58*, 388–402. https://doi.org/10.1016/j.infsof.2014.07.005

[13] Tomar, D., & Agarwal, S. (2016). Prediction of defective software modules using class imbalance learning. *Applied Computational Intelligence and Soft Computing, 2016*, Article 7658207, 12 pages. https://doi.org/10.1155/2016/7658207

[14] Yang, X., Tang, K., & Yao, X. (2015). A learning-to-rank approach to software defect prediction. IEEE Transactions on Reliability, 64(1), 234–246. https://doi.org/10.1109/TR.2014.2370891

[15] Ni, C., Liu, W.-S., Chen, X., Gu, Q., Chen, D.-X., & Huang, Q.-G. (2017). A cluster based feature selection method for cross-project software defect prediction. *Journal of Computer Science and Technology, 32*(6), 1090–1107. https://doi.org/10.1007/s11390-017-1785-0