IEEE Computer Society Bangalore Chapter

Internship and Mentorship Program – 2025

Report
on

Project ID: P39

"Enhancing Software Testing with AI-Powered Automation"

Submitted by

NAME	**COLLEGE**
Prathyusha N B	NITTE MEENAKSHI INSTITUTE OF TECHNOLOGY
Abhigna Suresh Babu	NITTE MEENAKSHI INSTITUTE OF TECHNOLOGY
B S Bindushree	NITTE MEENAKSHI INSTITUTE OF TECHNOLOGY
Deepika P	NITTE MEENAKSHI INSTITUTE OF TECHNOLOGY

Under the Mentorship of

Dr.Amritendu Mukherjee
NeuraPixel AI Labs Pvt Ltd.

Contents of the Report

1. Introduction:

Software testing plays an indispensable role in the software development lifecycle. It ensures that applications perform reliably, meet user requirements, and maintain security standards. In traditional settings, software testing is largely manual or rule-based, demanding significant human effort. As software systems grow in complexity and the release cycles shrink in agile environments, manual testing alone becomes insufficient. This leads to longer testing cycles, delayed product releases, and higher chances of defect leakage into production.

Artificial Intelligence (AI) and Machine Learning (ML) have emerged as promising technologies to transform quality assurance practices. They can automate repetitive tasks, analyze large datasets, identify risks, and support predictive decision-making. The goal of this project, titled **'AI-Powered Software Testing Automation**,' is to design a modular framework that integrates AI-driven techniques across multiple phases of the testing workflow. These phases include requirement analysis, test case generation, prioritization, defect prediction, failure categorization, auto-triage, and API testing.

This report provides an overview of the problem statement, limitations of existing systems, the proposed AI-based solution, tools and technologies learned, detailed architectural framework, implementation methodology, results obtained, and conclusions with future scope. By following a structured approach, the project demonstrates how AI can reduce manual effort, increase test coverage, and enhance defect detection.

2. Existing System:

Traditional software testing practices rely heavily on manual processes and expert judgment. Test cases are manually derived from requirement documents, which is tedious and prone to oversight. For example, when a requirement states 'The system must allow login within 5 seconds,' a human tester needs to interpret and write corresponding test cases. This manual dependency often results in missing test cases and inconsistent coverage.

In prioritization, manual systems typically follow static rules such as 'run all critical tests first,' without considering real defect history or system risk factors. This can lead to wasted resources, as low-impact tests may be executed earlier than high-risk ones .

Defect triage in existing systems requires testers and QA leads to manually inspect failure logs, identify the root cause, and assign the defect to the correct team. This process is slow and often inconsistent across organizations. Finally, API testing is mostly script-based using tools such as Postman or JMeter, which lack predictive intelligence. They can check correctness but cannot anticipate anomalies or performance degradation.

Thus, existing systems face challenges in scalability, speed, cost-efficiency, and intelligent decision-making. The need for an AI-based system is evident to address these limitations.

3. Proposed System

The proposed system introduces AI and ML techniques at different layers of the software testing lifecycle. Its modular pipeline ensures that each task is handled intelligently and can integrate seamlessly into modern CI/CD environments. The modules include:

1. **Requirement Analysis & Test Case Generation:** Using NLP (spaCy), requirement sentences containing keywords like *should*, *must*, and *shall* are automatically detected and transformed into structured test cases with IDs, steps, and expected results.
2. **Test Case Prioritization:** A Random Forest classifier is trained using simulated features such as past defect counts, execution time, and failure rates. This enables risk-based prioritization of test cases, ensuring critical ones are executed first.
3. **Defect Prediction & Root Cause Analysis:** Failure logs are vectorized using TF-IDF and classified with a Naive Bayes model into categories like Code Defect, Environment Issue, or Flaky Test.
4. **Failure Categorization & Auto-Triage:** Predicted defect types are automatically mapped to responsible teams (Developer, DevOps, or QA). This eliminates delays in defect assignment and accelerates resolution.
5. **AI-Based API Testing:** API request-response pairs are analyzed using regression models to predict expected response times. If the observed response deviates significantly, anomalies are flagged for investigation.

This approach ensures faster, smarter, and more efficient testing with reduced manual intervention.

4. Knowledge gained - Tools, Technology, Courses etc.

This project provided extensive exposure to both theoretical and practical aspects of AI-driven software testing. The following tools and technologies were explored:

- **Python Programming:** Core language for implementation due to its flexibility and rich ecosystem.
- **Natural Language Processing (spaCy):** Used to tokenize requirement documents, extract sentences, and identify keywords.
- **Data Handling (Pandas, NumPy):** For structuring, cleaning, and analyzing datasets including test cases and logs.
- **Machine Learning (Scikit-learn):** Implemented Random Forests for test prioritization, Naive Bayes for defect prediction, and regression for API response prediction.
- **Vectorization (TF-IDF):** Converted unstructured log messages into machine-readable vectors.
- **Software Testing Knowledge:** Concepts of requirement-based testing, risk-based prioritization, and automated defect triage.

AI Tools for Software Testing:

- **Claude:** Conversational AI by Anthropic for generating test cases, analyzing error logs, suggesting code improvements, and supporting technical documentation and debugging in real-time.
- **Codestral:** Large-scale AI (~22B parameters) for code generation, function completion, and unit test creation; suited for advanced workflows requiring GPU or cloud support.
- **Goose & Goose Selenium:** Open-source AI agent for code generation, debugging, and test creation; Goose Selenium adds browser automation converting natural language tests into executable Selenium scrip

- **Postman (with AI):** API testing platform integrating ML for response prediction and anomaly detection, with scripting, automated test execution, and visual workflows.
- **TestRigor:** No-code AI platform for UI and API testing; supports plain-English test creation, learns from user interactions, and enables end-to-end automation.
- **Codex (OpenAI):** LLM trained on public GitHub code, powering GitHub Copilot; converts natural language to code across multiple languages.
- **Cursor:** AI-first code editor with chatbot assistant for test generation, debugging, and code refactoring in VSCode.
- **Gemini Code Assist (Google):** Conversational AI for coding, testing, and debugging within Google Cloud and Android Studio.
- **GitHub Copilot:** Uses Codex to provide code suggestions, generate unit tests, and offer explanations in IDEs via a chatbot interface.
- **Amazon CodeWhisperer:** AI code suggestion tool for IDEs like VSCode and Cloud9; helps generate test functions and AWS-related tasks.

In addition, the project enhanced problem-solving, critical thinking, and technical writing skills, which are essential for professional software engineers.

5. Architectural Framework;

The architectural design of the proposed system is modular, ensuring clear separation of responsibilities:

1. Input Layer: Collects requirement documents and system logs.
2. NLP Processing Layer: Uses spaCy to tokenize text, extract requirement sentences, and identify testable conditions.
3. Test Case Management Layer: Structures extracted sentences into formal test cases stored in CSV.
4. Prioritization Layer: Applies ML models to assign risk scores and prioritize test cases.
5. Defect Analysis Layer: Processes failure logs, predicts defect types, and identifies root causes.
6. Auto-Triage Layer: Maps predicted defect categories to relevant teams.
7. API Testing Layer: Predicts response times and flags anomalies.

This architecture supports scalability and can be extended with continuous integration pipelines and real-world datasets**.**

6. Implementation Details:

The project was implemented using Python, following a systematic, stepwise workflow designed to integrate AI and machine learning into the software testing lifecycle. Each stage of the workflow was carefully structured to ensure accuracy, efficiency, and ease of integration with modern development practices:

- Step 1: Requirement Processing: Requirement documents, including functional and non-functional specifications, were loaded into the system. Using spaCy, sentences containing actionable requirements were identified by detecting keywords such as *must*, *should*, and *shall*. The extracted sentences were preprocessed to remove irrelevant text, normalize terminology, and ensure consistency across all requirements. This step provided the foundation for automated test case generation by structuring raw textual data into a format suitable for downstream processing.
- Step 2: Test Case Generation: The cleaned requirement sentences were transformed into structured test cases. Each test case included a unique identifier, preconditions, detailed execution steps, and expected outcomes. By automating this step, the system reduced the manual effort required to interpret requirement documents and ensured that no functional requirement was overlooked. This automation also improved traceability, as every test case could be linked back to the original requirement sentence.
- Step 3: Test Case Prioritization: To optimize test execution and focus on high-risk areas, simulated features such as historical defect counts, average execution time, and failure rates were used to train a Random Forest model. The model assigned risk scores to each test case, allowing critical test cases to be executed first. This risk-based prioritization ensured that the most impactful defects were identified early in the testing cycle, improving overall software quality and reducing the time required for regression testing.
- Step 4: Defect Prediction: Failure logs generated from previous test executions were collected and preprocessed. The log messages were vectorized using TF-IDF, converting unstructured textual data into machine-readable features. A Naive Bayes classifier was then trained to categorize defects into types such as *Code Defect*, *Environment Issue*, or

Flaky Test. This step enabled early identification of defect patterns and potential root causes, providing actionable insights for developers and QA teams.

- Step 5: Auto-Triage: Once defect types were predicted, the system automatically mapped each defect to the relevant team—Developer, QA, or DevOps—based on the nature of the issue. This automated triage eliminated delays in defect assignment, reduced human error, and accelerated resolution times. Teams received structured reports containing defect details, predicted categories, and suggested actions, improving overall workflow efficiency.
- Step 6: AI-Based API Testing: API endpoints were tested by analyzing request-response pairs. Regression models were trained to predict expected response times based on historical data. Any significant deviation between the predicted and observed responses was flagged as an anomaly for further investigation. This proactive detection of API performance issues enhanced reliability and reduced the likelihood of runtime errors in production environments.

Throughout the implementation, each step was validated using simulated datasets representing typical project scenarios. The outputs of all modules were analyzed for accuracy, consistency, and completeness. Logging mechanisms were incorporated at every stage to track processing steps, model predictions, and results, ensuring reproducibility and ease of debugging. This structured approach not only demonstrated the effectiveness of AI-driven software testing but also provided a scalable framework that can be adapted to real-world projects of varying complexity.

7. Results :

The project achieved the following results:

- Extracted requirement-based test cases accurately using NLP.
- Generated structured test cases in tabular format with IDs, steps, and expected results.
- Prioritized test cases with risk scores, ensuring high-risk cases were tested earlier.
- Predicted defect types from failure logs with meaningful accuracy.
- Auto-assigned defects to the correct teams, reducing manual triage effort.
- Predicted API response times and flagged anomalies effectively.

Sample outputs included:

- Extracted Test Case Example: "The system shall validate user credentials." → Test case ID TC_001.
- Prioritized Test Case Example: TC_005 with risk score 0.82 marked as high-risk.
- Defect Prediction Example: "NullPointerException in UserService" classified as Code Defect.
- API Testing Example: Predicted response time 180ms vs actual 250ms flagged as anomaly.

These results demonstrate the effectiveness of AI in automating diverse QA activities and improving software testing efficiency**.**

8. Conclusion :

This project validates the application of AI and ML techniques in enhancing software testing practices. By automating requirement analysis, test case generation, prioritization, defect classification, auto-triage, and API testing, the framework reduces manual effort, accelerates release cycles, and improves product quality. The modular pipeline ensures scalability and adaptability to real-world enterprise systems.

Future work may include:

- Incorporating deep learning models such as BERT for advanced NLP.
- Using real-world defect datasets for model training.
- Integrating with tools like JIRA, Selenium, and Jenkins for CI/CD workflows.
- Exploring reinforcement learning for adaptive test case prioritization.

Overall, the project demonstrates that AI-powered testing is not just a conceptual improvement but a practical solution for modern software development challenges.