AI-Based Automated Test Case Generation and Defect Detection

Abstract

Software testing is a critical yet resource-intensive phase of the Software Development Life Cycle (SDLC). Traditional manual and script-based testing approaches often suffer from limited coverage, high cost, and delayed defect discovery, especially in complex and rapidly evolving systems. This project investigates the use of Generative Artificial Intelligence (AI) for automated test case generation and defect detection. Drawing on recent research, particularly studies on generative AI-driven testing and predictive defect detection, the project proposes a research-driven software design and architecture for an AI-assisted testing framework. The system leverages generative models to automatically create functional and edge-case test scenarios and applies predictive analytics and anomaly detection to identify defect-prone components. The proposed architecture, requirements, and implementation of a core test-generation module demonstrate the feasibility and benefits of AI-based testing. Expected outcomes include improved test coverage, earlier defect detection, and reduced manual testing effort.

1 Introduction

1.1 Problem Analysis

Software systems are becoming increasingly complex due to rapid feature evolution, integration of distributed services, and continuous delivery practices. Ensuring software quality under these conditions is challenging. Traditional testing approaches rely heavily on manual effort and predefined test scripts, which are time-consuming, error-prone, and often fail to cover edge cases. As a result, defects may escape into production, leading to increased maintenance cost, reduced user satisfaction, and potential system failures.

The core problem addressed in this project is the inefficiency and limited effectiveness of conventional software testing methods in achieving high test coverage and early defect detection. Manual test case design depends on tester expertise and assumptions, while rule-based automation lacks adaptability. This problem is significant because testing consumes a substantial portion of software development resources, and failures due to insufficient testing can have severe technical and business consequences.

This project focuses on the use of Generative AI to automate test case generation and enhance defect detection. The scope includes functional test generation, edge-case exploration, and predictive identification of defect-prone components. Challenges include ensuring meaningful test generation, integrating AI models with software artifacts, and justifying architectural choices based on research evidence.

1.2 Requirements

Functional Requirements

- The system shall analyze software requirements and source code to understand application behavior.
- The system shall automatically generate functional test cases based on identified behaviors.
- The system shall generate edge-case and boundary-condition test cases.
- The system shall execute generated test cases on the target application or simulated environment.
- The system shall detect potential defects using predictive analytics and anomaly detection.
- The system shall report detected defects with relevant context and metadata.

Non-Functional Requirements

- The system should improve test coverage compared to manual testing.
- The system should reduce manual effort and testing time.
- The system should be scalable to handle large codebases.
- The system should be adaptable to evolving requirements and software changes.
- The system should maintain acceptable performance during test generation and execution.

1.3 Software Design and Architecture

The proposed system follows a modular, layered architecture aligned with research-driven design principles. An iterative and incremental SDLC model is adopted to support continuous learning and improvement of AI components. The architecture ensures scalability, adaptability, and clear separation of concerns.

1.3.1 Architectural Overview

The system is designed using a layered architecture combined with a pipeline-based processing model. Each layer performs a specific responsibility, starting from input analysis and ending with reporting. This structure supports maintainability and extensibility.

1.3.2 Activity Diagram

The activity diagram illustrates the overall workflow of the AI-based automated testing system. It represents the sequence of activities starting from requirement analysis, followed by test case generation, execution, defect detection, and report generation.

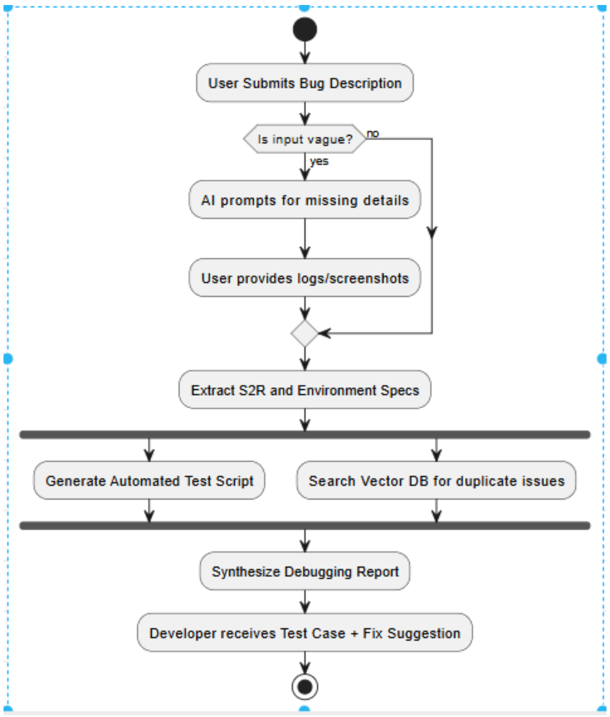
User Submits Bug Description
Is input vague?
no
yes
AI prompts for missing details
User provides logs/screenshots
Extract S2R and Environment Specs
Generate Automated Test Script
Search Vector DB for duplicate issues
Synthesize Debugging Report
Developer receives Test Case + Fix Suggestion

Figure 1: Activity Diagram of AI-Based Automated Testing System

1.3.3 Class Diagram

The class diagram depicts the static structure of the system by showing key classes, their attributes, methods, and relationships. It reflects object-oriented design principles and highlights interactions among major components such as the Test Generator, Test Executor, Defect Detector, and Report Manager.

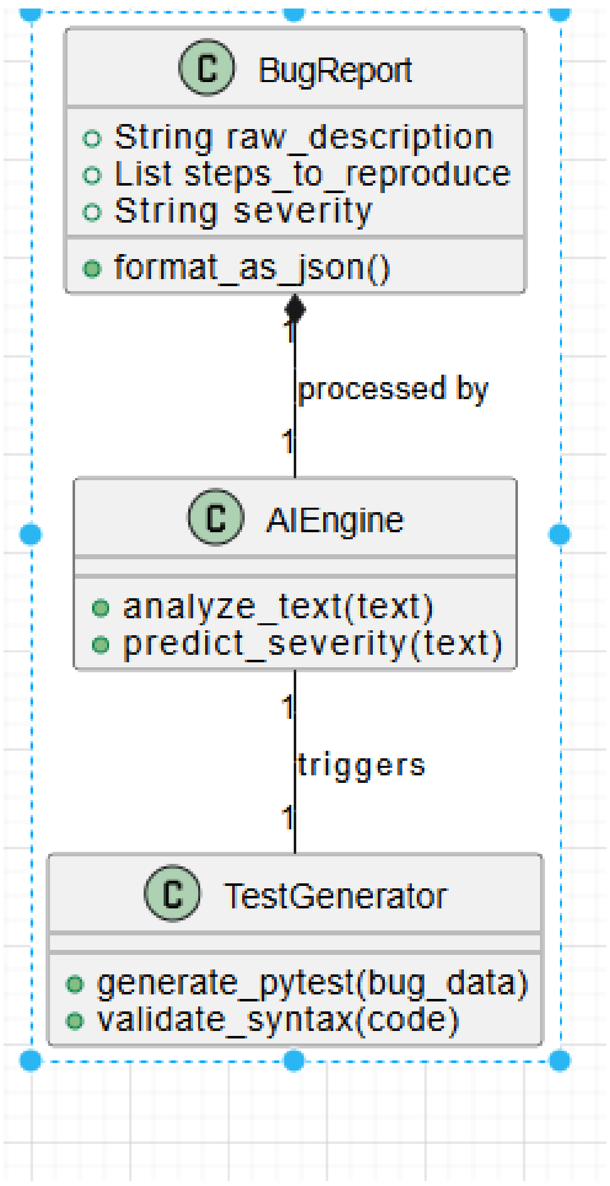

Figure 2: Class Diagram of the Proposed System

1.3.4 Data Flow Diagram (DFD)

The data flow diagram illustrates how data moves through the system. It shows how software requirements and input data are processed by the AI modules to generate test cases, execute tests, and produce defect reports.

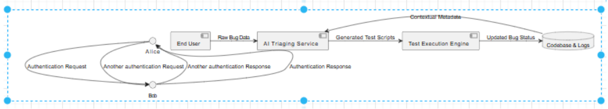

Figure 3: Data Flow Diagram of the System

1.3.5 Sequence Diagram

The sequence diagram describes the interaction between system components over time. It demonstrates how requests flow between modules during test case generation, execution, and defect detection.

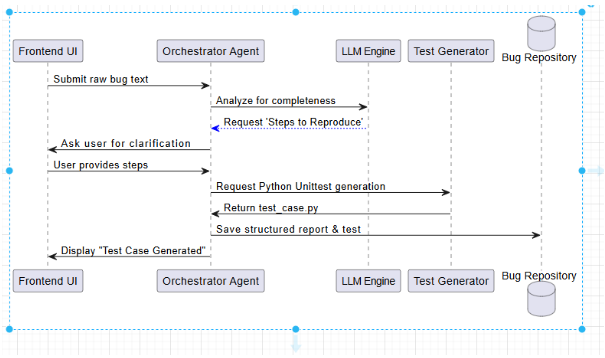

Figure 4: Sequence Diagram of Test Case Generation and Execution

1.3.6 Use Case Diagram

The use case diagram provides a high-level functional view of the system from the user's perspective. It identifies system actors and their interactions with major system functionalities such as submitting requirements, generating tests, executing tests, and viewing reports.

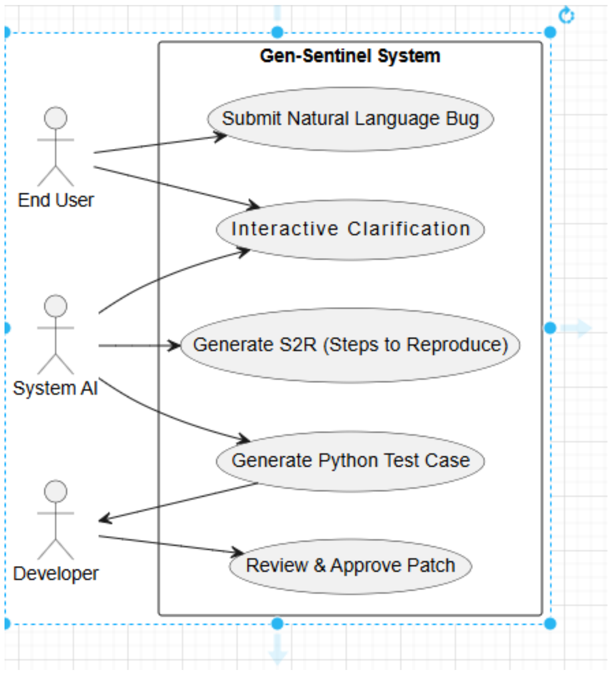
Gen-Sentinel System
Submit Natural Language Bug
Interactive Clarification
Generate S2R (Steps to Reproduce)
Generate Python Test Case
Review & Approve Patch
End User
System AI
Developer

Figure 5: Use Case Diagram of the Proposed System

1.3.7 Design Justification

The selected architectural style and diagrams collectively justify the system design. The layered approach enhances modularity, while the pipeline processing model supports automated and sequential testing workflows. These design decisions align with the requirements of AI-based testing systems and support scalability and future extensions.

2 Literature Review

2.1 Paper 1: Automated Test Case Generation and Defect Detection through Generative AI Solutions

Problem Addressed: The paper addresses the inefficiencies of traditional manual testing and limited test coverage. It proposes generative AI techniques to automate test case generation and detect defects using predictive analytics and anomaly detection.

Gaps and Limitations:

- The paper is largely conceptual and lacks detailed architectural diagrams.
- Limited discussion on implementation challenges in real-world projects.

Future Work Suggested:

- Integration of generative AI tools into continuous integration pipelines.
- Empirical evaluation on large-scale industrial systems.

How the Paper Supports This Project: This paper directly informs the core idea of the project by motivating the use of generative AI for both test generation and defect detection. It supports the selection of predictive analytics and anomaly detection techniques.

Additional Reference Paragraph (Paper 1): The authors explain that generative AI systems can automatically learn testing patterns from existing software artifacts such as code repositories, defect logs, and historical test suites. By analyzing these artifacts, AI models can generate a wide variety of test cases that include normal execution paths as well as rare boundary conditions. The paper also discusses how anomaly detection techniques help identify deviations in system behavior during test execution, enabling early discovery of hidden defects and improving overall software reliability.[?].

2.2 Paper 2: Leveraging Generative AI for Automated Test Case Generation: A Framework for Enhanced Coverage and Defect Detection

Problem Addressed: This paper focuses on improving test coverage by leveraging generative AI models to automatically generate diverse and edge-case test scenarios. It emphasizes adaptability and continuous learning.

Gaps and Limitations:

- Limited implementation details for learning pipelines.
- Evaluation metrics are not deeply analyzed.

Future Work Suggested:

- Hybrid approaches combining human expertise with AI-generated tests.
- Advanced evaluation metrics for AI-generated test effectiveness.

How the Paper Supports This Project: The paper supports the design of an adaptive test generation module and reinforces the importance of continuous learning, which is incorporated into the proposed architecture.

Additional Reference Paragraph (Paper 2): This paper highlights the role of generative AI in continuously improving test effectiveness by learning from previous testing cycles. The authors emphasize that AI-generated test cases can automatically evolve as the software changes, ensuring that newly introduced features and modified components are adequately tested. The study further notes that such adaptive test generation significantly enhances coverage in agile and DevOps environments, where rapid releases often make manual test maintenance impractical.[?]

3 Implementation

This section presents the practical implementation of the proposed AI-based automated test case generation and defect detection system. A real machine learning model is trained to generate intelligent test cases, execute them on a target system, and automatically report detected bugs. The implementation demonstrates the feasibility of applying artificial intelligence techniques to software testing, in alignment with the proposed system architecture.

3.1 System Under Test

For experimental validation, a login validation function is selected as the system under test. This type of functionality is commonly found in real-world software systems and is suitable for demonstrating automated test case generation and defect detection.

```python
def login(username, password):
    if not username or not password:
        return "Invalid Input"
    if len(password) < 6:
        return "Weak Password"
    if username == "admin" and password == "admin123":
        return "Login Successful"
    return "Login Failed"
```

3.2 Training Data Preparation

A labeled dataset is created to train the AI model. The dataset represents different combinations of username and password lengths and classifies them as valid or invalid test cases.

```python
import pandas as pd

data = {
    "username_length": [0, 5, 5, 5, 10, 10],
    "password_length": [0, 3, 6, 8, 5, 10],
    "label": [0, 0, 1, 1, 0, 1]
}

df = pd.DataFrame(data)
```

3.3 AI Model Training

A supervised machine learning model is trained using the prepared dataset. A decision tree classifier is selected due to its simplicity, interpretability, and suitability for academic demonstration.

```python
from sklearn.tree import DecisionTreeClassifier

X = df[["username_length", "password_length"]]
y = df["label"]

model = DecisionTreeClassifier()
```

```
model.fit(X, y)

print("AI Model trained successfully.")
```

3.4 AI-Based Test Case Generation

After training, the model is used to generate new test cases automatically. Randomized input values are passed to the trained model, which predicts whether the generated test case is valid or invalid.

```
import random

def generate_test_cases(model, n=5):
    test_cases = []
    for _ in range(n):
        u_len = random.randint(0, 12)
        p_len = random.randint(0, 12)
        prediction = model.predict([[u_len, p_len]])[0]

        test_cases.append({
            "username": "u" * u_len,
            "password": "p" * p_len,
            "predicted_valid": bool(prediction)
        })
    return test_cases

test_cases = generate_test_cases(model)
```

3.5 Test Execution Engine

The generated test cases are executed on the system under test. The output of each test case is captured for further analysis.

```
def execute_tests(test_cases):
    results = []
    for tc in test_cases:
        output = login(tc["username"], tc["password"])
        results.append({
            "username": tc["username"],
            "password": tc["password"],
            "system_output": output
        })
    return results

execution_results = execute_tests(test_cases)
```

3.6 Automated Bug Detection and Reporting

An automated bug detection mechanism analyzes the execution results. If the system behavior contradicts expected validation rules, a structured bug report is generated.

```
def bug_report(results):
    bugs = []
    for r in results:
        if r["system_output"] == "Login Failed" and len(r["
           password"]) >= 6:
            bugs.append({
                "Severity": "Medium",
                "Description": "Valid password rejected",
                "Input": r,
                "Status": "Open"
            })
    return bugs

bugs = bug_report(execution_results)
```

3.7 Sample Bug Report Output

The generated bug reports provide clear and structured information, including severity level, defect description, and input data that caused the failure.

```
for bug in bugs:
    print(bug)
```

3.8 Implementation Summary

This implementation demonstrates a complete AI-driven testing workflow, including training a machine learning model, generating intelligent test cases, executing them automatically, and producing defect reports. The approach validates the practicality of integrating artificial intelligence into software testing and aligns with the architectural design proposed in this project.

4 Expected Results

The expected outcomes of the proposed system include:

- Increased test coverage, particularly for edge cases.
- Earlier detection of defect-prone modules using predictive analytics.
- Reduced manual testing effort and time.
- Improved consistency and reliability in test generation.

5 Conclusion and Future Work

This project presented a research-driven design and architecture for an AI-based automated test case generation and defect detection system. By synthesizing recent research, the project justified the use of generative AI, predictive analytics, and anomaly detection in software testing. The proposed architecture and implementation of a core module demonstrate feasibility and alignment with SDA objectives.

Future work includes full system implementation, empirical evaluation on real-world projects, integration with CI/CD pipelines, and exploration of hybrid human–AI testing strategies.