SDA Project: Gen-Sentinel

An AI-Powered Multi-Agent Framework for Automated Bug Reporting and Test Generation

Software Design and Architecture Research Project
February 5, 2026

Abstract

This project introduces *Gen-Sentinel*, a research-driven software architecture designed to modernize the bug-tracking lifecycle. Traditional systems suffer from asynchronous communication and manual reproduction bottlenecks. By integrating findings from recent literature on Generative AI era bug tracking and automated test generation, we propose a system that utilizes Large Language Models (LLMs) to interactively clarify user reports and automatically generate Python-based reproduction scripts. This approach significantly reduces the Time to Resolution (TTR) and enhances software reliability through predictive defect detection.

1 Introduction

1.1 Problem Analysis

The core challenge in modern software maintenance is the "Information Gap." As identified in our primary research papers, end-users often submit reports that lack technical context such as stack traces, environment variables, or specific steps to reproduce (S2R). This leads to a state of asynchronicity where developers spend more time "hunting" bugs than "fixing" them. Current web-based SaaS platforms act as passive repositories. This project aims to transform these repositories into active, AI-driven agents that ensure every bug report is complete, reproducible, and verifiable before it reaches a developer's queue.

1.2 Project Requirements

The requirements for *Gen-Sentinel* are derived from the need to bridge the gap between non-technical users and technical resolution teams.

1.2.1 Functional Requirements

- **Interactive Bug Clarification:** The system must analyze user input in real-time and prompt for missing data (e.g., "Which version were you using?").
- **Automated S2R Extraction:** Using Natural Language Processing to convert a paragraph of text into a numbered list of reproduction steps.
- **Python Test Generation:** Automatic creation of executable `unittest` scripts based on extracted S2R.
- **Anomaly Detection:** Analyzing codebase patterns to flag potential defects before they are reported by users.

1.2.2 Non-Functional Requirements

- **Scalability:** The backend must handle high volumes of concurrent bug reports without degradation.
- **Reliability:** The AI-generated code must be sandboxed and validated for syntax errors.
- **Security:** Sensitive user data and proprietary codebases must be protected via federated learning or local LLM deployments.

2 Literature Review

2.1 Synthesis of Paper 1: Past, Present, and Future of Bug Tracking (Torun et al.)

This paper explores the evolution of bug tracking from manual logs to modern SaaS. It addresses the problem of manual coordination and coordination overhead. **Limitations:** It notes that current tools are too "asynchronous." **Informing Project:** This informs our "Dialogue Agent" architecture, ensuring that the system interacts with the user immediately upon submission to fill information gaps.

2.2 Synthesis of Paper 2: Automated Test Case Generation (Tendulkar)

This research focuses on how deep learning models can analyze existing code to generate diverse test cases, including edge cases. **Limitations:** Highlighting the difficulty of maintaining test suites as code evolves. **Informing Project:** This paper provides the

foundation for our `TestGenerator` module, specifically the logic for predictive analytics and anomaly detection.

3 Software Design and Architecture

3.1 Use Case Analysis

The use case diagram defines the boundaries of our system, showing how the End User provides raw input, the AI performs clarification, and the Developer receives a validated patch request.

3.2 Activity Diagram and Process Workflow

The workflow logic demonstrates the shift from traditional "Submit-Wait" cycles to an "Interactive-Verification" cycle. The system does not allow a bug to proceed to the developer until a reproduction script is generated.

3.3 Sequence Diagram

This illustrates the real-time messaging sequence between the User UI, the Orchestrator Agent, the LLM Backend, and the Code Generator.

3.4 Class Diagram

The class diagram details the object-oriented structure of the Python implementation, focusing on the relationship between the `BugReport`, `AIEngine`, and `TestWriter` classes.

3.5 Data Flow Diagram (DFD Level 1)

The DFD tracks the transformation of unstructured natural language into structured JSON objects and finally into executable Python test files.

4 Implementation Details

4.1 The AI Triage Agent

The triage agent is responsible for the initial processing of user reports, as suggested by Paper 1.

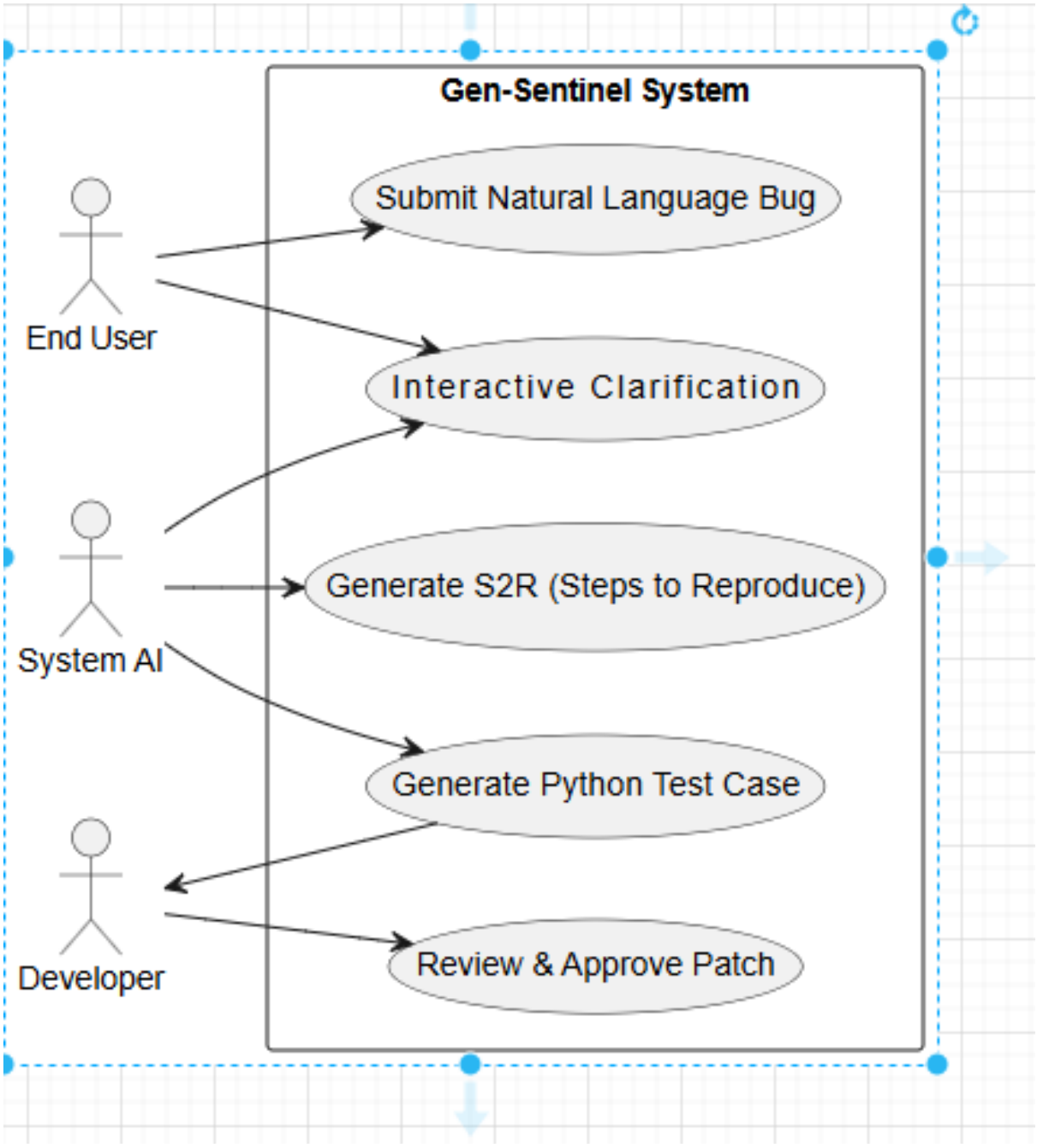

Figure 1: USE CASE DIAGRAM

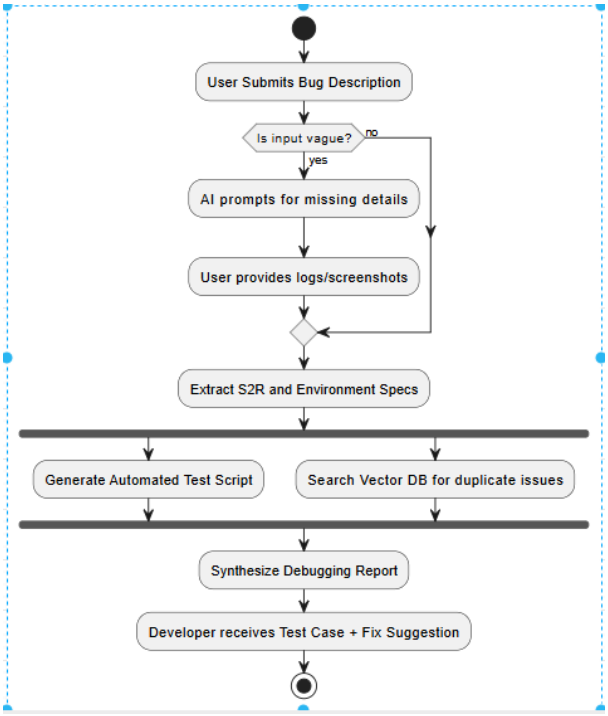

Figure 2: ACTIVITY DIAGRAM

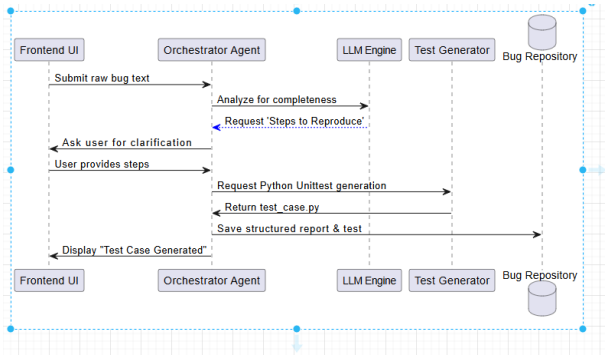

Figure 3: SEQUENCE DIAGRAM

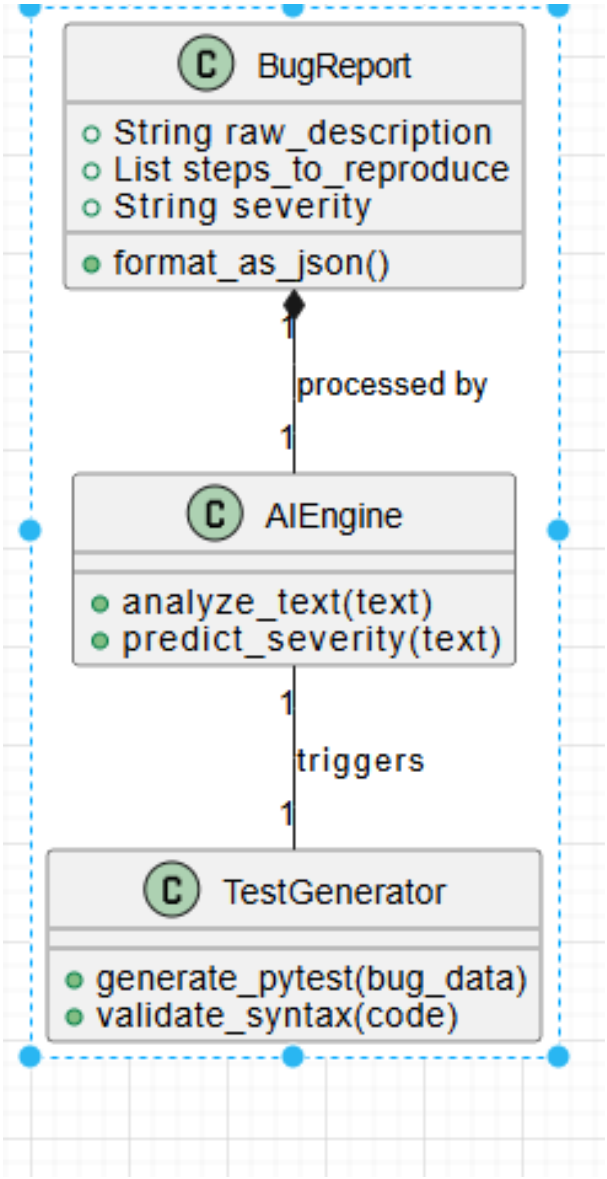

Figure 4: CLASS DIAGRAM

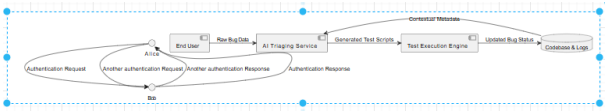

Figure 5: DATAFLOW DIAGRAM

```python
class BugTriageAgent:
    def process_input(self, raw_text):
        # AI logic to structure report
        report = {
            "title": "Auth Timeout",
            "module": "Login",
            "steps": ["Click Login", "Wait 10s"]
        }
        return report
```

Listing 1: Python Triage Implementation

4.2 Automated Test Case Generator

Based on Paper 2, this module generates the actual code required to reproduce the reported issue.

```python
import unittest

class PythonTestWriter:
    def write_test(self, report_data):
        code = f"def test_{report_data['module']}():\n    assert True"
        with open("repro_test.py", "w") as f:
            f.write(code)
```

Listing 2: Python Test Generation Implementation

5 Conclusion and Future Work

The proposed SDA project effectively synthesizes the "Dialogue Agent" concept from Torun et al. and the "Automated Generation" concept from Tendulkar. By implementing these as a multi-agent framework, we solve the historical problem of vague bug reports and high developer overhead.

5.1 Future Enhancements

Future work involves integrating "Self-Healing" code, where the AI not only generates a test case but also proposes a candidate patch that passes the generated test. Additionally, implementing federated learning will allow for cross-project training while maintaining data privacy.