SONATA SOFTWARE

GEN AI POC Design Document

Table of Contents:

1. Introduction

1.1 Background

The Test Case Generator is a web-based application that automates the creation of unit test cases from code snippets provided by users. By leveraging Generative AI and NLP models, this application facilitates quick, reliable test case creation, significantly improving efficiency in the software testing lifecycle.

1.2 Purpose

This document details the design and implementation of the Test Case Generator, covering its architecture, functionalities, and operational flow. It provides an overview of how developers can use the application to generate test cases in various languages and analyze its backend processing and error handling.

1.3 Scope

The POC allows users to input code snippets, select programming languages (e.g., Python, Java, C#), and generate customizable test cases automatically. This application is designed to assist developers by automating repetitive tasks, focusing on generating high-quality unit tests.

2. Architecture Overview

2.1 Frontend

The user interface, built with **React**, provides a seamless user experience:

- **Code Input Section**: Allows users to enter code snippets.
- **Test Case Configuration**: Enables users to set the number and types (positive/negative) of test cases.
- **Language Selector**: Dropdown to select programming languages.

- **Result Display & Copy Functionality**: Shows generated test cases with options to copy or close the output.
- **Loading Spinner**: Indicates data processing status.

2.2 Backend

The backend, implemented in **Flask**, handles data processing and manages communication with the Generative AI model:

- **Request Validation & Input Sanitization**: Ensures valid and secure inputs.
- **Generative Model Interaction**: Processes requests with the AI model to generate contextually accurate test cases.
- **Error Handling and Response Management**: Provides informative feedback and handles errors smoothly.

3. Design and Implementation

3.1 User Interface Design

The UI consists of intuitive form inputs for code and test case configurations, with optional fields for scenario-based input. React manages state across components to ensure a dynamic user experience.

3.2 Document Processing

Processes the user's code input and converts it into a structured format suitable for the AI model. Each request includes code, language, and test case specifications to guide test case generation.

3.3 Query Processing

Constructs prompts based on user input and language selection. The AI model uses these prompts to create test cases, which are tailored to the specified language and parameters.

3.4 Error Handling

Comprehensive error handling includes checks for invalid or unsupported languages, incomplete inputs, and API connectivity issues. Meaningful messages guide users in troubleshooting.

4. Functionality

4.1 Code Input

Accepts a range of programming languages and enables direct code input, making the tool adaptable to various user requirements.

4.2 Language Selection

The dropdown supports multiple languages, ensuring generated test cases are compatible with the syntax and conventions of the chosen language.

4.3 Automated Test Case Generation

Utilizes AI to generate both positive and negative test cases based on the user's parameters. NLP models ensure relevance and coverage across essential code scenarios.

5. Output and Testing

5.1 Test Cases

A variety of test cases validate each functionality, from input validation to error handling. Tests are run for different language selections and test case configurations.

5.2 Test Results

User feedback and performance metrics indicate that the application meets its intended objectives, providing accurate and timely test cases.

6. Conclusion

6.1 Achievements

The Test Case Generator streamlines the unit testing process, allowing developers to focus on other critical tasks by automating repetitive test case creation.

6.2 Challenges

Key challenges included handling unsupported languages, managing response times for large code inputs, and refining error messages. These were mitigated with efficient backend management and modular design.

6.3 Future Enhancements

Potential improvements include support for additional programming languages, enhanced test case customization, and interactive guidance to further optimize user experience.

Developed By,

Kavipriyaa P

Digital Associate Engineer (26610)