Project Documentation

1. Introduction

- Project Title: SmartSDLC - AI-Enhanced Software Development Lifecycle
- Team Members:
    - Karnam Madhuri
    - Inkollu Sushma Meghana
    - Nagamani Pinniboina

2. Project Overview

- Purpose: SmartSDLC is an AI-driven platform designed to significantly enhance efficiency and quality across the Software Development Lifecycle. Its purpose is to automate repetitive tasks, provide intelligent assistance, and streamline workflows for software development professionals, leading to faster delivery of higher-quality software.
- Features:
    - Intelligent Requirement Classification & User Story Generation
    - AI Code Generation
    - Automated Bug Fixing
    - Smart Test Case Generation
    - Code Summarization
    - Interactive AI Chatbot Assistant

3. Architecture

- Frontend: The frontend is a web-based application built using standard web technologies. It provides a responsive and intuitive user interface with distinct tabs for each AI-powered functionality and a floating chatbot.
    - Technology: HTML, CSS, JavaScript.
- Backend: The backend is a Python-based RESTful API that serves as the central hub for all AI logic and data processing. It receives requests from the frontend, manages the AI model, orchestrates AI functionalities, and handles PDF text extraction.
    - Technology: Python (FastAPI).
- AI Model/Service: The core intelligence is provided by a large language model.
    - Technology: ibm-granite/granite-3.3-2b-instruct (accessed via Hugging Face Transformers library).
- PDF Processing: Integrated within the backend for document analysis.
    - Technology: PyMuPDF

4. Setup Instructions

- Prerequisites:
    1. Python 3.8+
    2. Pip (Python package installer)

3. git (for cloning the repository)
4. A web browser for the frontend (Chrome, Firefox, Edge, Safari recommended)
5. Google Colab environment (for backend execution, especially for GPU access)
6. Hugging Face Access Token (for model access)

- Installation:
  1. Clone the repository: https://github.com/Madhuri-Karnam/SmartSDLC---AI-Enhanced-Software-Development-Lifecycle
  2. Navigate to the backend directory: cd Project Files/main.py
  3. Install dependencies: pip install fastapi uvicorn “python-multipart” “pypdfium2>=4.22.0” “PyMuPDF==1.23.8” “transformers==4.30.2” “torch==2.0.1” “accelerate==0.21.0” “langchain==0.0.240” “langchain-community==0.0.1” “protobuf==3.20.0”
  4. Set your Hugging Face Token: Ensure your HUGGING_FACE_TOKEN is correctly set in main.py or as an environment variable.

5. Folder Structure

- Client (Frontend):
  - index.html : Main application entry point.
  - css/ (inline in index.html): Tailwind CSS configuration and custom styles.
  - js/ (inline in index.html): JavaScript logic for UI interaction and API calls.
- Server (Backend):
  - main.py: FastAPI application, AI model integration, API endpoints.
  - requirements.txt: List of Python dependencies.

6. Running the Application

- Frontend: Open the index.html file directly in your web browser. Ensure the FASTAPI_BASE_URL in index.html points to your running backend.
- Backend: Run the uvicorn command as specified in the installation steps within your Google Colab environment or local Python environment.

7. API Documentation

➢ /classify-requirements
- Method: POST
- Input: pdf_file (File, multipart/form-data)
- Output: JSON object with classified_data(categorized requirements).

➢ /generate-code
- Method: POST
- Input: prompt(Form string)
- Output: JSON object with code(generated code string).

➢ /fix-bug
- Method: POST

- Input: code_snippet (Form string)
- Output: JSON object with original_code and fixed_code (strings).

➢ /generate-test-cases
- Method: POST
- Input: code_or_req(Form string)
- Output: JSON object with test_cases (generated test cases string).

➢ /summarize-code
- Method: POST
- Input: code_snippet (Form string)
- Output: JSON object with summary (code summary string)

➢ /chatbot
- Method: POST
- Input: user_message(Form string)
- Output: JSON object with ai_response(chatbot's response string).

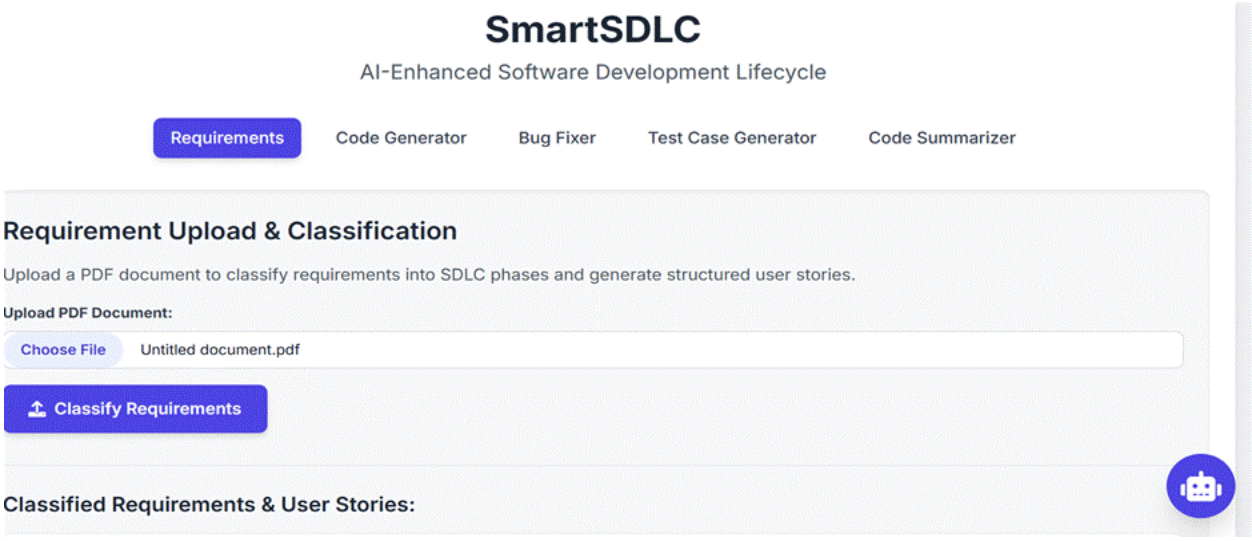
8. Authentication

- The current prototype of SmartSDLC does not implement user authentication or authorization.
- Note: For a production environment, robust authentication and authorization mechanisms would be critical to secure endpoints and manage user access. Access to the Hugging Face model is managed via an API token within the backend.

9. User Interface

1. Intelligent Requirement Classification:

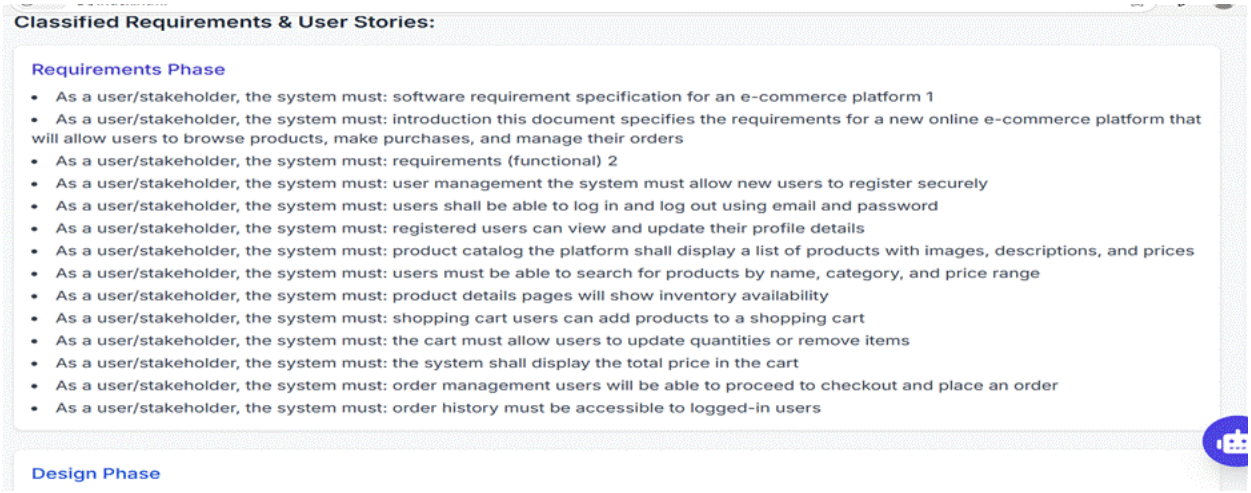
Input:



Description: This feature allows users to upload unstructured text documents (primarily PDFs) containing project requirements. The system then processes this text, identifies key requirement

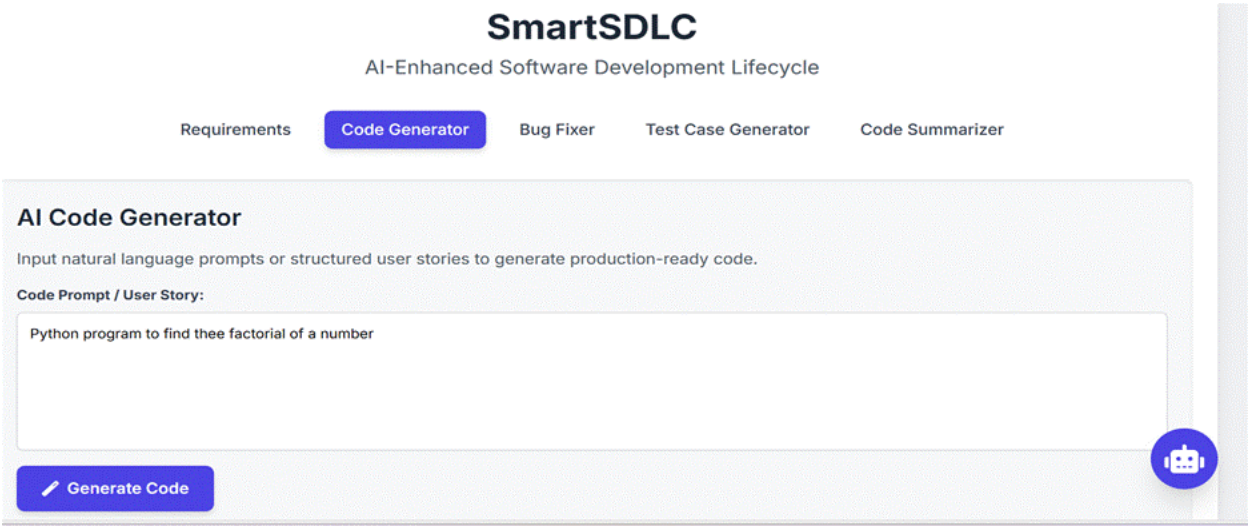
sentences, and classifies them into standard SDLC phases (e.g., Requirements, Design, Development, Testing, Deployment, Other). It can also generate structured user stories from the extracted information.

Output:



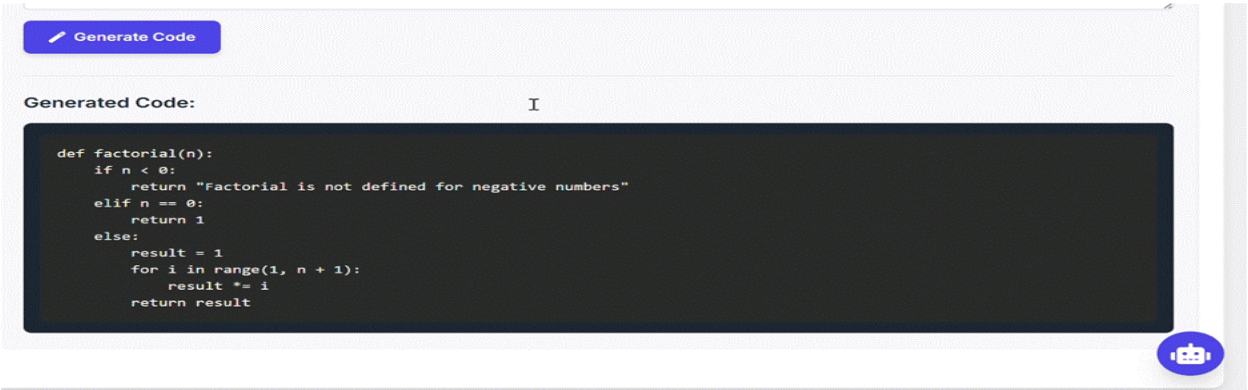
2. AI Code Generator:

Input:



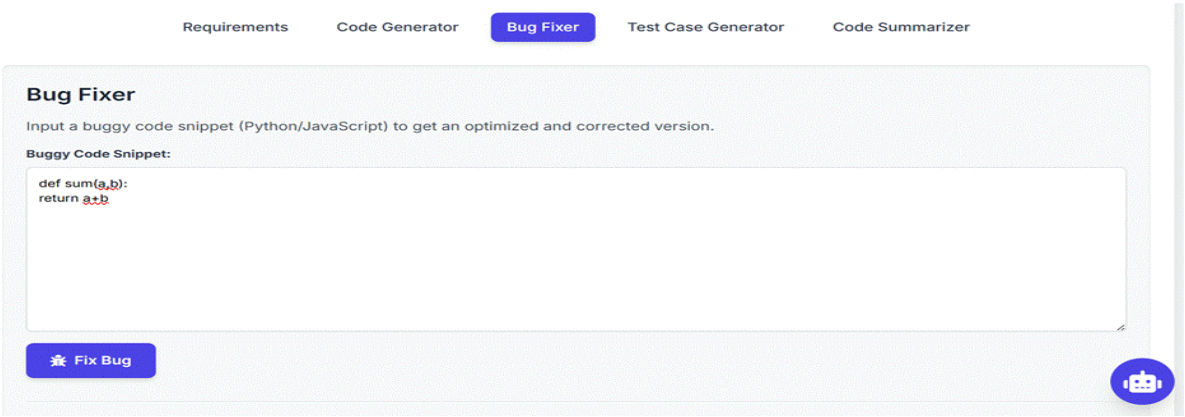
Description: This powerful tool enables developers to generate functional code snippets or full functions by simply providing a natural language prompt or a user story. It significantly accelerates the coding process for boilerplate, common algorithms, or initial feature implementations.

Output:



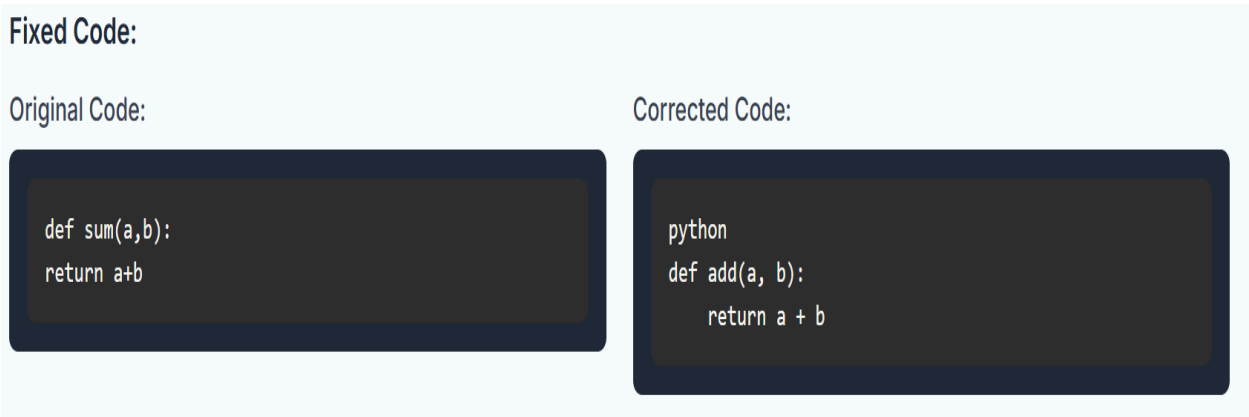
3. Automated Bug Fixer:

Input:



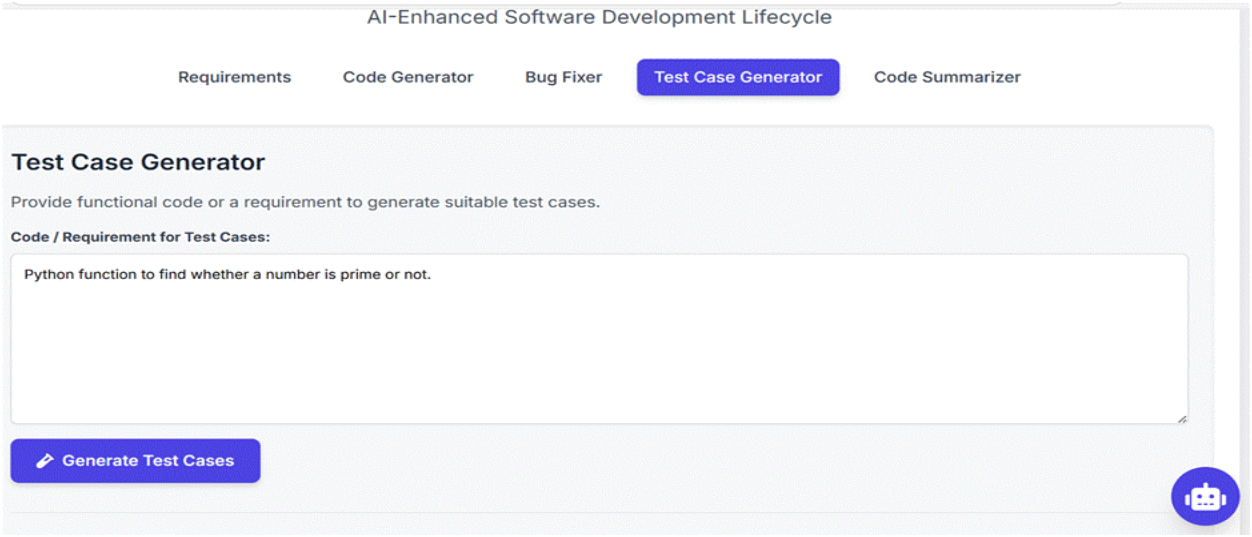
Description: Designed to assist developers in quickly identifying and correcting errors. Users can paste a buggy code snippet, and the AI analyzes it to suggest and provide an optimized, corrected version, reducing manual debugging time and improving code quality.

Output:



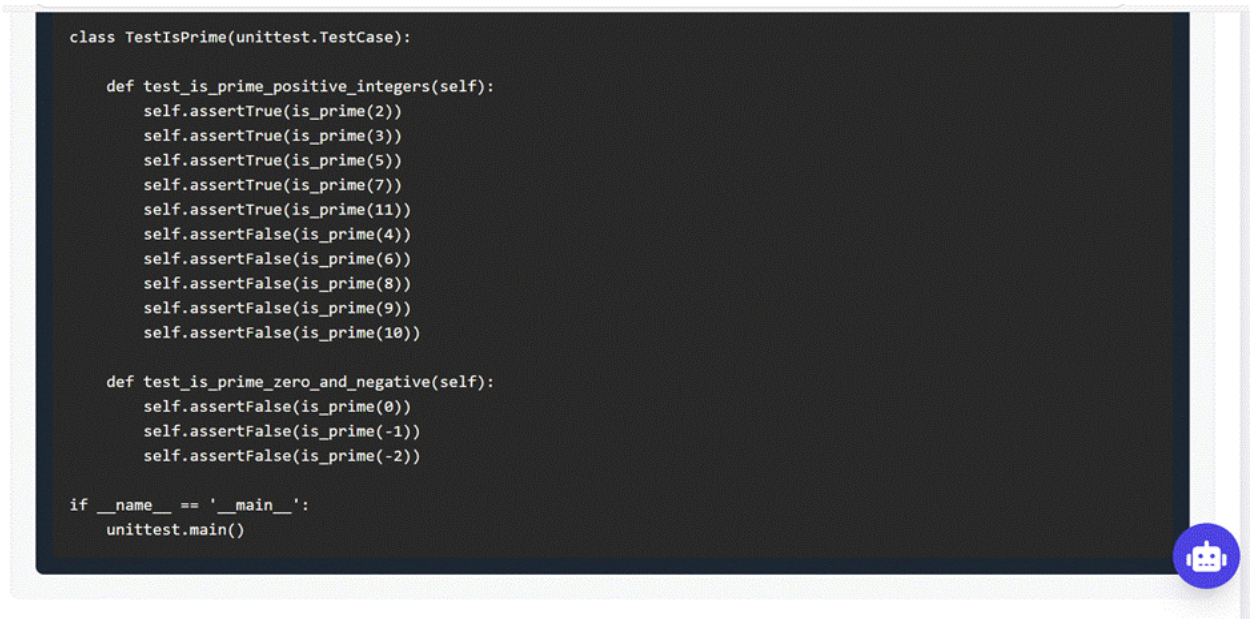
4. Smart Test Case Generator:

Input:



Description: This feature automates the creation of test cases. Developers or QA engineers can provide either functional code or a software requirement, and the AI will generate relevant unit or integration test cases, often adhering to specific testing frameworks like unittest or pytest.

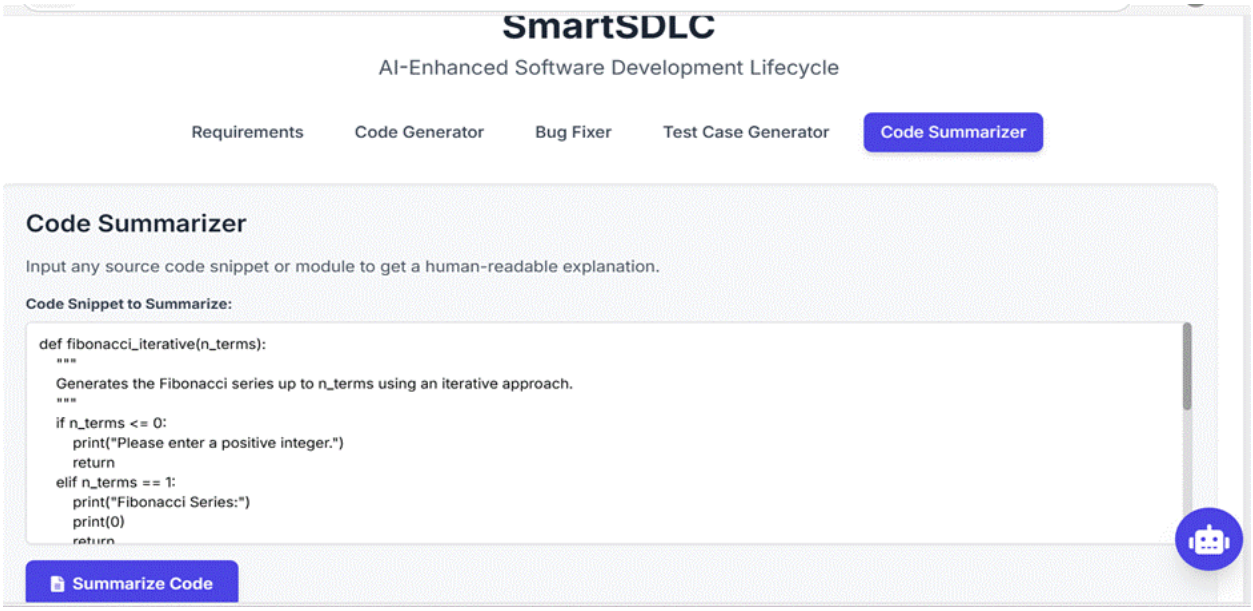
Output:



```
class TestIsPrime(unittest.TestCase):

    def test_is_prime_positive_integers(self):
        self.assertTrue(is_prime(2))
        self.assertTrue(is_prime(3))
        self.assertTrue(is_prime(5))
        self.assertTrue(is_prime(7))
        self.assertTrue(is_prime(11))
        self.assertFalse(is_prime(4))
        self.assertFalse(is_prime(6))
        self.assertFalse(is_prime(8))
        self.assertFalse(is_prime(9))
        self.assertFalse(is_prime(10))

    def test_is_prime_zero_and_negative(self):
        self.assertFalse(is_prime(0))
        self.assertFalse(is_prime(-1))
        self.assertFalse(is_prime(-2))

if __name__ == '__main__':
    unittest.main()
```

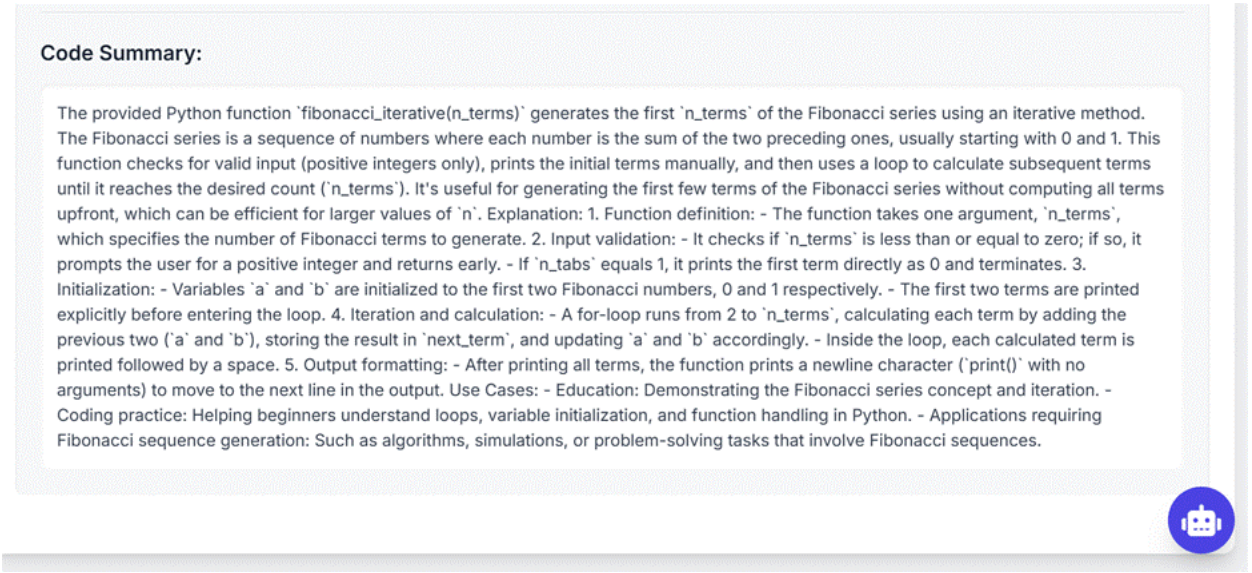
5. Code Summarizer:

Input:



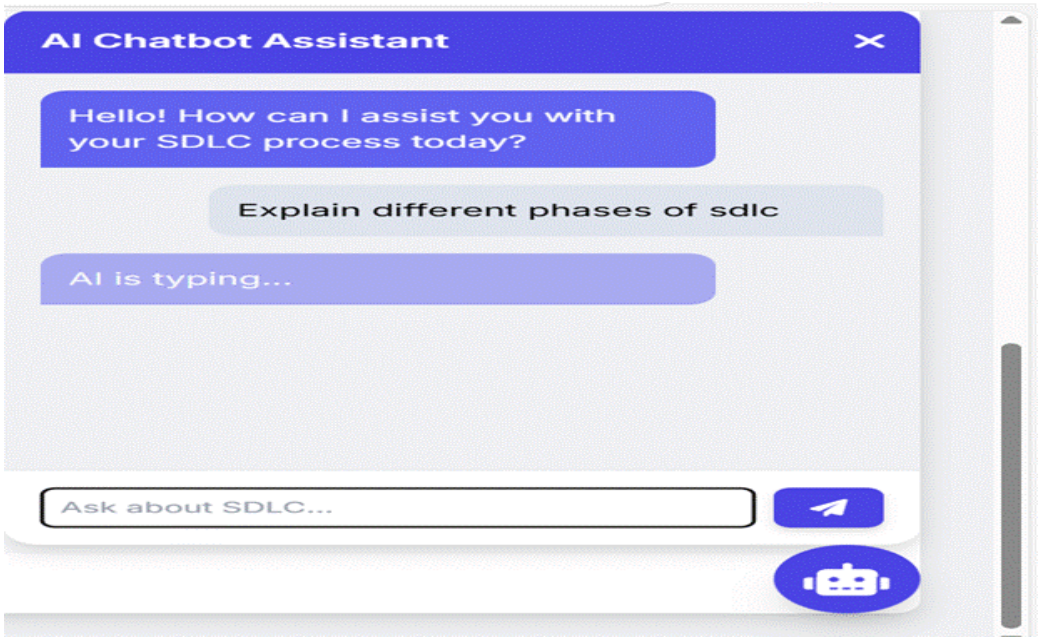
Description: To enhance code comprehension and knowledge transfer, this tool provides human-readable explanations and summaries of any given code snippet or module. It's invaluable for new team members, code reviews, or understanding complex legacy codebases.

Output:



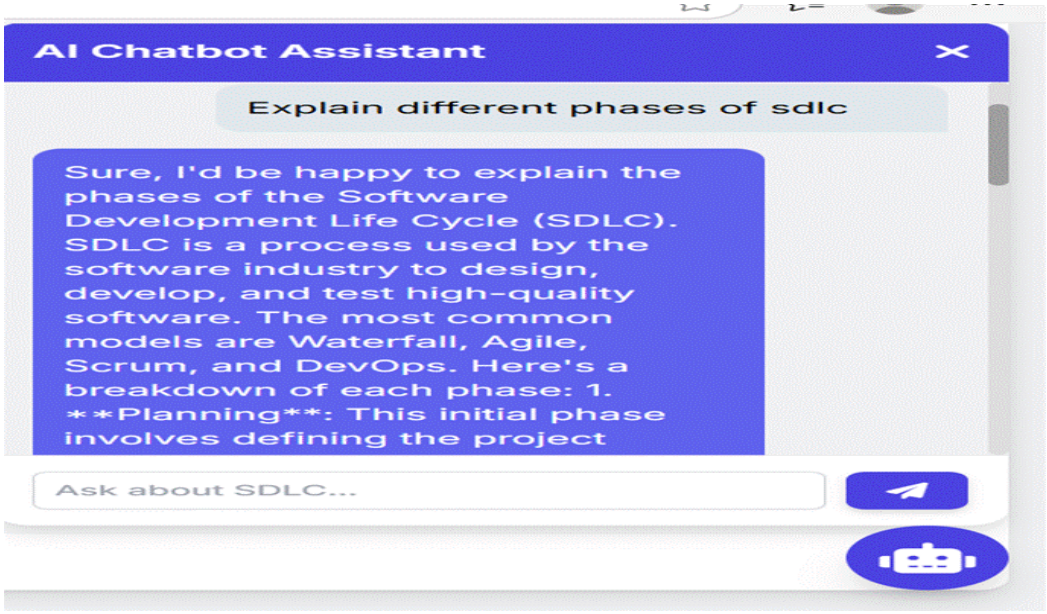
6. Interactive AI Chatbot Assistant:

Input:



Description: A conversational AI assistant available on-demand to answer questions related to the Software Development Lifecycle. It can provide information on methodologies (Agile, Scrum), best practices, specific tools, or general programming concepts, acting as an instant knowledge base.

Output:



10. Testing

- Testing Strategy: The project utilized a combination of functional and performance testing. Functional tests verified the correctness of AI-generated outputs and core feature functionalities (e.g., PDF parsing, code generation accuracy). Performance tests focused on API response times and stability under load.
- Tools Used: Manual testing for UI/UX. Python's requests library for API testing. Basic timing mechanisms for performance measurement. (For comprehensive testing, frameworks like pytest for backend unit/integration tests and browser automation tools for frontend E2E tests would be used.)

11. Screenshots or Demo

● Project Demo Link:
https://drive.google.com/file/d/1xgeYCrtmhoBJwuw1YLYb5UlvFUFGm0yE/view?us p=sharing

12. Known Issues

- AI Model Limitations: The ibm-granite model, while powerful, may occasionally produce "hallucinations" or suboptimal outputs, requiring human review.
- PDF Parsing Variations: Complex PDF layouts or scanned images may result in less accurate text extraction.
- Limited Language Support: Currently, code generation and analysis primarily focus on Python.
- No Persistent Storage: User data and chat history are not persistently stored across sessions in this prototype.
- No User Authentication: The prototype lacks user login/registration and role-based access control.

13. Future Enhancements

- Implement robust user authentication and authorization.
- Integrate with popular IDEs (e.g., VS Code extensions) for seamless workflow.
- Expand AI capabilities to support more programming languages and advanced tasks (e.g., automated refactoring, design pattern suggestions).
- Integrate with CI/CD pipelines and project management tools (e.g., Jira, GitHub Actions).
- Add persistent storage for user preferences, project data, and chat history.
- Develop a more sophisticated UI/UX with advanced code highlighting and interactive elements.
- Explore fine-tuning the AI model on domain-specific codebases for improved accuracy.
- Implement comprehensive logging and monitoring for production deployments.