Project Title: SmartSDLC – AI-Enhanced Software Development Lifecycle.

1.Introduction:

Team ID : NM2025TMID06815

Team Size : 5

Team Leader : SRINITHI R

Team member : SNEHA U

Team member : SWETHA M

Team member : THEERTHANA A

Team member : THIRUVALARSELVI M

Sure! Here's a comprehensive explanation of the **SmartSDLC – AI-Enhanced Software Development Lifecycle** project, covering the following:

1. ✅ Project Overview
2. 🧱 Architecture
3. ⚙️ Setup Instructions
4. 📁 Folder Structure
5. ▶️ Running the Application
6. 📘 API Documentation
7. 🔐 Authentication
8. 👷 User Interface
9. 🧪 Testing

☑ 1. Project Overview

SmartSDLC is a full-stack software development platform powered by **Artificial Intelligence** that assists developers, testers, and project managers throughout the **Software Development Lifecycle (SDLC)**.

It leverages AI to:

- Extract & validate requirements from natural language input
- Generate system design diagrams and boilerplate code
- Auto-generate test cases

- Predict bugs and deployment issues
- Automate CI/CD pipelines and monitoring

🧱 2. Architecture

🧠 SmartSDLC Modular Architecture

```
[Frontend UI - React/Next.js]
         |
         v
[Backend API - Node.js / Python (Flask/FastAPI)]
         |
         +-----------------------------+
         |                             |
[AI Engine - NLP, ML Models, LLMs]     [Database - MongoDB / PostgreSQL]
         |
         v
[AI Modules for Each SDLC Phase]
  ├─ Requirement Analyzer (NLP)
  ├─ Design Generator (LLM + Diagramming Tool)
  ├─ Code Generator (LLM/GPT + Template Engine)
  ├─ Test Generator (AI Test Engine)
  ├─ Deployment Optimizer (DevOps AI)
  └─ Monitoring & Feedback (AIOps)
```

Milestone 1: Model Selection and Architecture

Milestone 1 focuses on selecting the optimal AI models for SmartSDLC, choosing granite-13b-chat-v1 for natural language tasks and granite-20b-code-instruct for code generation. The architecture integrates the backend (FastAPI), frontend (Streamlit), AI layer (Watsonx), and modules (LangChain, GitHub). The development environment is set up with necessary dependencies, API keys, and a modular structure, enabling efficient integration and development in subsequent phases.

Activity 1.1: Research and select the appropriate generative AI model

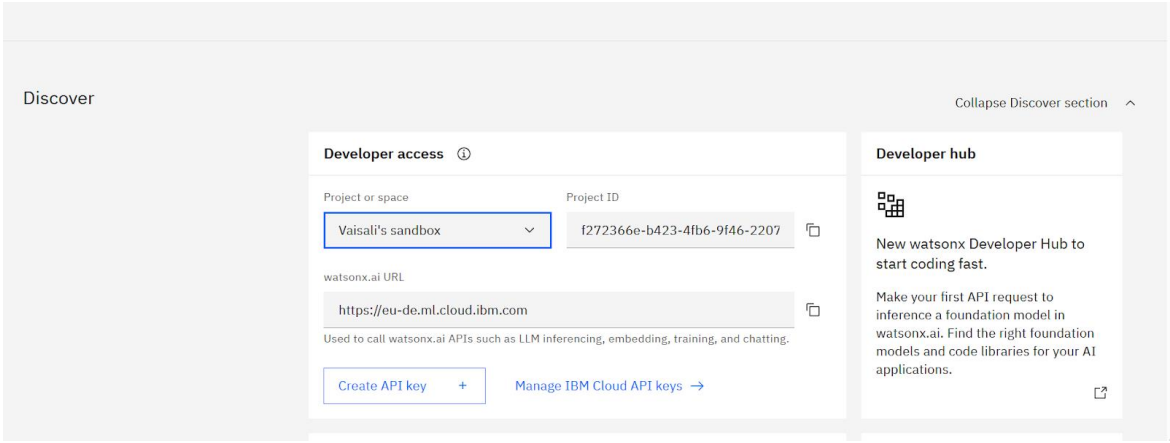
1. **Understand the Project Requirements:**
   Analyze the goals of the SmartSDLC platform, particularly the need for automating SDLC phases like requirement analysis, code generation, test creation, bug fixing, and documentation. This understanding helps narrow down which generative AI models best suit each task.

2. **Explore Available Models:**
   Study IBM's Watsonx Granite model documentation to compare available foundation models. Focus on those capable of handling natural language processing and code synthesis, such as granite-13b-chat-v1 for language understanding and granite-20b-code-instruct for structured code generation and summarization.

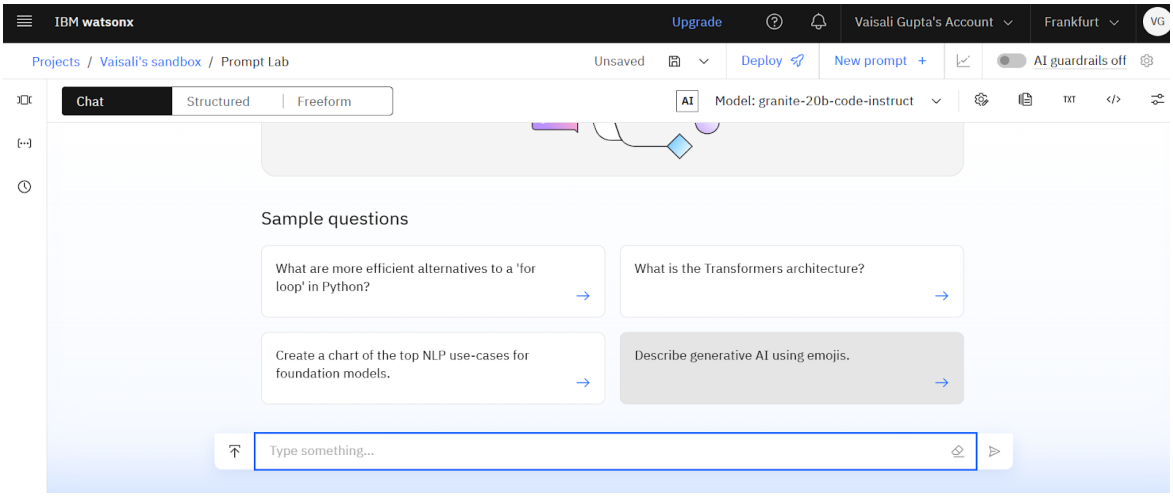
3. **Evaluate Model Performance:**
   Evaluate models based on metrics like accuracy, latency, and quality of generated responses across different SDLC tasks. Use prior benchmarks or run pilot queries (e.g., "generate a function to validate email input") to assess model suitability.

4. **Select Optimal Models:**
   Finalize the two best-fit Granite models—granite-13b-chat-v1 for interacting with natural language prompts and granite-20b-code-instruct for multilingual and functional code outputs. These models will power the core logic of SmartSDLC's automation system.

5. **Generating Watsonx ai API key**



6. **Selecting the model**



Activity 1.2: Define the architecture of the application.

1. **Draft a System Architecture Diagram:**

   Create a high-level visual representation that includes the backend (FastAPI), the frontend (Streamlit), the AI layer (Watsonx APIs), and service modules (LangChain, GitHub integration). Include flow direction between components such as file uploads, AI interactions, and response rendering.

2. **Define Frontend-Backend Interaction:**

Detail how the Streamlit UI captures user inputs (PDF uploads, text prompts, buggy code) and sends them to FastAPI endpoints. Also, document how the backend routes handle logic and return structured outputs like user stories or code blocks.

3. **Backend Service Responsibilities:**
   Define roles of each backend module:

   a. PDF processing and classification logic (using PyMuPDF).
   b. Prompt generation and API communication with Watsonx.
   c. LangChain agent for orchestration.
   d. Auth module for user sessions and hashed logins.
4. **AI Integration Flow:**
   Outline how API calls are made to Watsonx (or optionally LangChain), and how responses are parsed and displayed back to the user. Specify if fallback models or retries will be incorporated.

Activity 1.3: Set up the development environment:

1. **Install Python and Pip:**
   Ensure Python 3.10 is installed for compatibility with Watsonx SDK. Install pip to manage dependencies like FastAPI, Streamlit, and PyMuPDF.
2. **Create a Virtual Environment:**
   Use venv to create isolated environments for backend and frontend:

```
python -m venv myenv
myenv\Scripts\activate
```

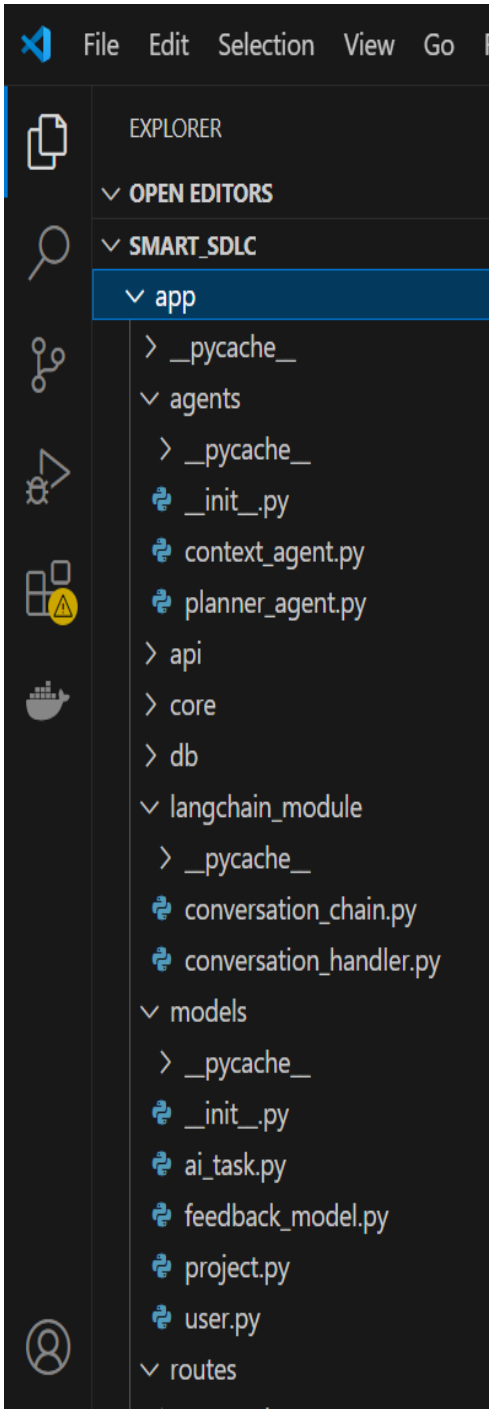
3. **Install Required Libraries:**
   Install all required packages such as fastapi, uvicorn, pymupdf, requests, langchain, streamlit, and ibm-watsonx-ai using pip. Store these in requirements.txt.

4. **Set Up API Keys and Env Files:**
   Generate IBM Cloud API key and store it securely in a .env file. Include other

configs like model IDs and LangChain agent settings.

5. **Organize Project Structure:**
   Set up the initial folder structure for both frontend (smart_sdlc_frontend/) and backend (app/), ensuring proper separation of api/, services/, models/, and utils/ modules for clean development.

file structure



Milestone 2: Core Functionalities Development

Milestone 2 involves building the core AI-driven features of SmartSDLC, including requirement analysis, code generation, test case creation, bug fixing, code summarization, chatbot assistance, and GitHub integration. These functionalities are implemented using Watsonx and LangChain via modular service scripts. The FastAPI backend handles routing, authentication, and smooth API interactions, connecting frontend inputs with AI processing and generating structured outputs.

Activity 2.1: Develop the core functionalities:

This activity focuses on building the logic behind each AI-powered SDLC feature. These functionalities are the heart of SmartSDLC and are implemented using IBM Watsonx and LangChain integration.

1. AI-Powered Requirement Analysis

- **Module:** ai_story_generator.py
- **Description:** Extracts text from uploaded PDFs and uses AI to classify each sentence into SDLC phases (Requirements, Design, Development, etc.). Then it converts relevant sentences into structured user stories to be used in planning or code generation.

```python
@router.post("/upload-pdf")
async def upload_pdf(file: UploadFile = File(...)):
    contents = await file.read()

    with open("temp.pdf", "wb") as f:
        f.write(contents)

    doc = fitz.open("temp.pdf")
    full_text = "".join([page.get_text() for page in doc])
    doc.close()
```

app > services > ai_story_generator.py

```python
# services/ai_story_generator.py

def generate_user_story(requirement: str) -> str:
    # Dummy version — you can replace with Watsonx/OpenAI later
    return f"As a user, I want to {requirement.lower()} so that I can achieve my goal."

```

2. Multilingual Code Generation

- **Module:** code_generator.py

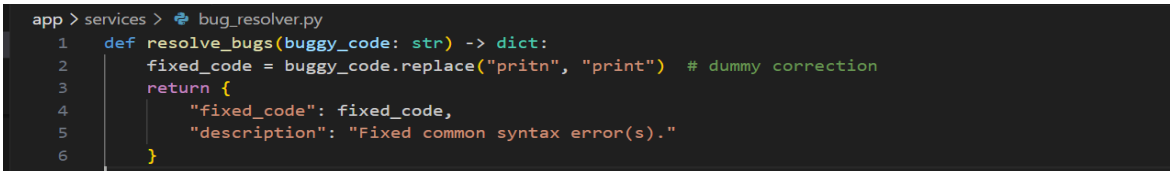
- **Description:** Takes task descriptions or user stories and generates clean, production-ready code in Python or other languages using granite-20b-code-instruct. Ideal for initial development tasks or boilerplate generation.

3. AI-Driven Test Case Generation

- **Module:** Likely part of code_generator.py
- **Description:** Generates unit tests or integration test cases based on the generated or user-provided code. Helps developers quickly validate core logic.

4. Bug Fixing and Code Correction

- **Module:** bug_resolver.py
- **Description:** Takes buggy code snippets and automatically identifies and fixes errors using AI. Returns a clean version of the code along with optional explanations.



5. Code Summarization and Documentation

- **Module:** doc_generator.py
- **Description:** Extracts summaries or generates documentation-style descriptions of code blocks. Helps with code readability and maintainability.

6. Chatbot Assistance

- **Modules:** conversation_handler.py, chat_routes.py
- **Description:** A floating AI chatbot helps users navigate the platform, answer questions, or give suggestions on SDLC tasks. Powered by LangChain and Watsonx integration.

app > langchain_module > conversation_handler.py

```python
from app.langchain_module.conversation_chain import chat_with_granite

# Expanded SDLC-related keywords (case-insensitive)
SDLC_KEYWORDS = [
    "requirement", "user story", "user stories", "design", "architecture",
    "development", "coding", "implementation", "programming", "code",
    "testing", "test case", "test cases", "unit test", "integration test",
    "bug", "debug", "fix", "error", "quality assurance", "qa",
    "deployment", "release", "launch", "production",
    "maintenance", "support", "upgrade", "hotfix", "patch",
    "documentation", "version control", "repository", "git", "jira", "devops",
    "sdlc", "software development life cycle", "software process",
    "agile", "scrum", "waterfall", "kanban", "model", "spiral","SDLC", "v-model"
]

# Check if message is SDLC-related
def is_sdlc_related(message: str) -> bool:
    return any(keyword in message.lower() for keyword in SDLC_KEYWORDS)

# Handle conversation with detailed SDLC guidance
def handle_conversation(message: str) -> str:
    try:
        if not is_sdlc_related(message):
            return (
                "🤖 I specialize in Software Development Life Cycle (SDLC) topics.\n\n"
                "You can ask me about:\n"
                "- 📌 SDLC Phases (Requirements, Design, Development, Testing, Deployment, Maintenance)\n"
                "- 🔁 Methodologies (Agile, Scrum, Waterfall, V-Model, Spiral, Kanban)\n"
                "- 🛠️ Tools (Git, JIRA, Jenkins, Selenium)\n"
```

app > langchain_module > conversation_chain.py

```python
import os
import requests
from dotenv import load_dotenv

# Load environment variables
load_dotenv()

# Fetch credentials from .env
API_KEY = os.getenv("WATSONX_API_KEY")
PROJECT_ID = os.getenv("WATSONX_PROJECT_ID")

# Get IAM Token from WatsonX
def get_iam_token(api_key: str) -> str:
    url = "https://iam.cloud.ibm.com/identity/token"
    headers = {"Content-Type": "application/x-www-form-urlencoded"}
    data = {
        "apikey": api_key,
        "grant_type": "urn:ibm:params:oauth:grant-type:apikey"
    }

    try:
        response = requests.post(url, headers=headers, data=data)
        response.raise_for_status()
        token = response.json()["access_token"]
        print("🔐 Token fetched successfully.")
        return token
    except Exception as e:
        print(f"[❌ ERROR fetching token] {e}")
```

7. Project Feedback and Learning Loop

- **Module:** feedback_routes.py, feedback_model.py

- **Description:** Collects user feedback on AI output and stores it in a database or JSON for improving prompts, models, or usability in future iterations.

app > models > feedback_model.py

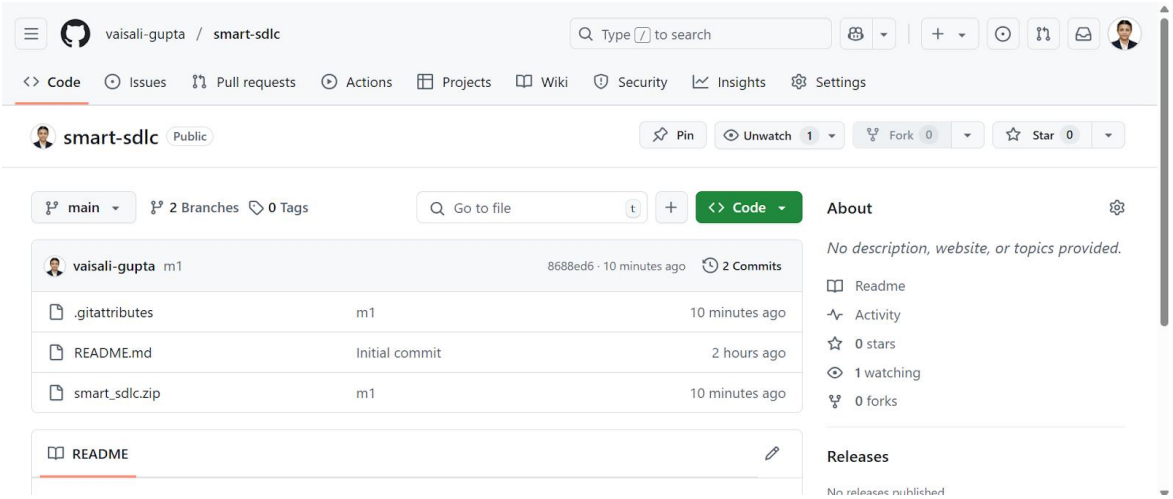
```python
from pydantic import BaseModel

class Feedback(BaseModel):
    name: str = "Anonymous"
    message: str
```

app > routes > feedback_routes.py

```python
from fastapi import APIRouter, HTTPException
from pydantic import BaseModel
import json
import os

router = APIRouter()

# ✅ Path to your feedback file inside the db folder
FEEDBACK_FILE = "app/db/feedback_data.json"

# Create the file if it doesn't exist
if not os.path.exists(FEEDBACK_FILE):
    with open(FEEDBACK_FILE, "w") as f:
        json.dump([], f)

class Feedback(BaseModel):
    username: str
    message: str
    rating: int  # e.g., 1 to 5 stars

@router.post("/feedback")
def collect_feedback(feedback: Feedback):
    try:
        # Load existing feedback
        with open(FEEDBACK_FILE, "r") as f:
            data = json.load(f)
```

8. GitHub Integration

- **Module:** github_service.py
- **Description:** Automates GitHub workflows such as pushing AI-generated code to a repo, opening issues, or syncing generated documentation.



Output: Fast API Backend

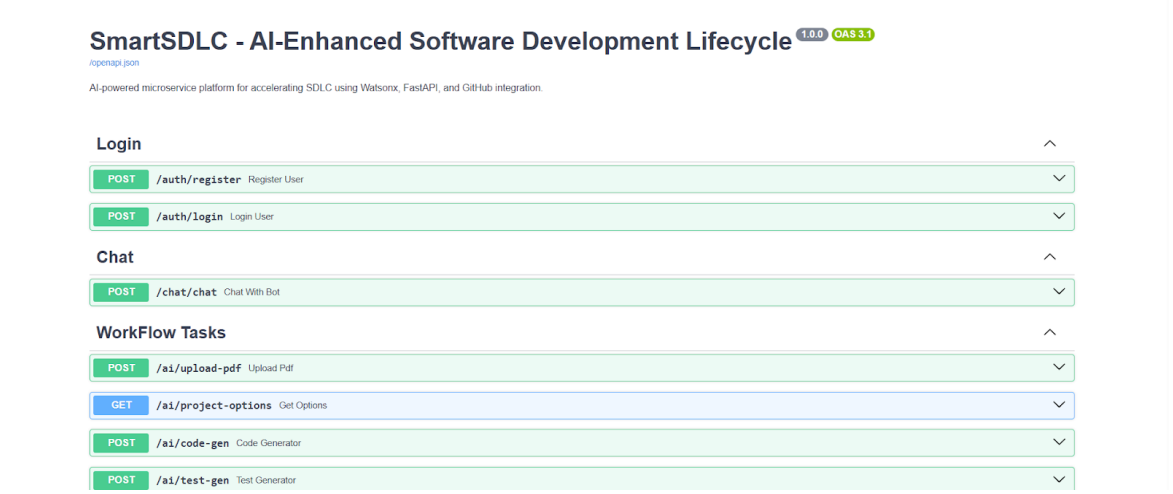


Figure 2: SmartSDLC API Documentation – Explore and interact with AI-powered microservices for login, chat, and workflow automation via FastAPI Swagger UI.

Activity 2.2: Implement the FastAPI backend to manage routing and user input processing, ensuring smooth API interactions.

This activity focuses on the **API layer and routing logic** that connects user requests with the backend services and AI models.

1. Backend Routing

- **Modules:** routes/ai_routes.py, routes/auth_routes.py, routes/chat_routes.py, routes/feedback_routes.py
- **Description:** FastAPI routers handle incoming POST/GET requests from the frontend. Each endpoint corresponds to one feature such as /generate-code, /upload-pdf, /fix-bugs, or /register.

```python
# === AI Services ===
@router.post("/code-gen")
def code_generator(requirements: str = Form(...), language: str = Form("python")):
    return generate_code(requirements, language)

@router.post("/test-gen")
def test_generator(code: str = Form(...), language: str = Form("python")):
    return generate_test(code, language)

@router.post("/fix-bugs")
def bug_fixer(buggy_code: str = Form(...)):
    return fix_bugs(buggy_code)

@router.post("/summarize")
def summarize_code(code: str = Form(...)):
    return summarize_documentation(code)
```

2. User Authentication

- **Modules:** auth_routes.py, security.py, user.py
- **Description:** Manages secure user login and registration with hashed passwords. Each request checks the user's session or credentials before executing the corresponding service.

3. Service Layer Logic

- **Modules:** services/
- **Description:** The service modules encapsulate the core business logic for AI processing. Each route calls the appropriate service function with cleaned user input and returns formatted responses.

4. AI & LangChain Integration

- **Modules:** watsonx_service.py, chat_routes.py
- **Description:** These modules handle interaction with external AI services. Inputs from routes are formatted into prompts, passed to the Watsonx or LangChain model, and the AI responses are processed.

app > services > watsonx_service.py

```python
import os
import requests
import logging
from dotenv import load_dotenv
from app.services.code_generator import generate_code_snippet
from app.services.doc_generator import generate_docstring
from app.services.bug_resolver import resolve_bugs

load_dotenv()

# Watsonx API endpoint
WATSONX_API_URL = "https://eu-de.ml.cloud.ibm.com/ml/v1/text/generation?version=2023-05-29"

# Enable basic logging
logging.basicConfig(level=logging.INFO)

# Function to retrieve a Bearer token from IBM IAM using the API key
def get_iam_token() -> str:
    response = requests.post(
        url="https://iam.cloud.ibm.com/identity/token",
        headers={"Content-Type": "application/x-www-form-urlencoded"},
        data={
            "grant_type": "urn:ibm:params:oauth:grant-type:apikey",
            "apikey": os.getenv("API_KEY"),
        },
    )

    if response.status_code != 200:
        logging.error("Failed to get IAM token: %s", response.text)
```

app > routes > chat_routes.py

```python
# app/routes/chat_routes.py

from fastapi import APIRouter, Form, HTTPException
from app.langchain_module.conversation_handler import handle_conversation

router = APIRouter()

@router.post("/chat")
def chat_with_bot(message: str = Form(...)):
    try:
        response = handle_conversation(message)
        return {"response": response}
    except Exception as e:
        print(f"[❌ ERROR] {e}")
        raise HTTPException(status_code=500, detail="Failed to process message.")

```

Milestone 3: main.py Development

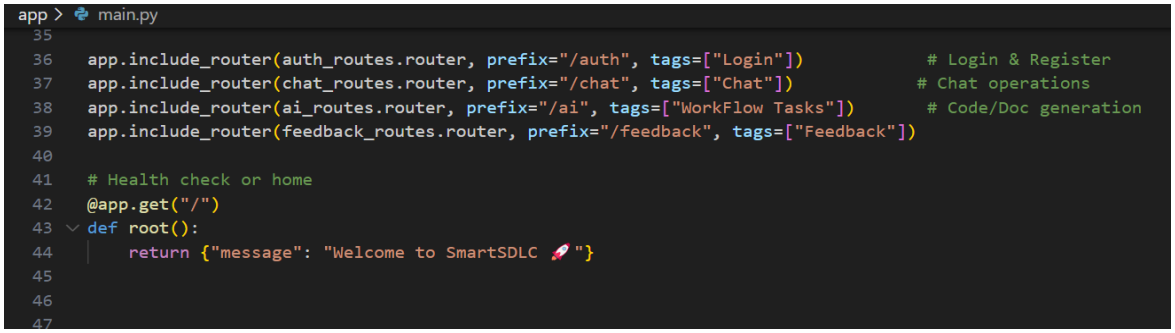
Milestone 3 focuses on creating and organizing the main control center of the application – the main.py file, which powers the Fast API backend in SmartSDLC. This milestone involves linking each core SDLC functionality through clearly defined API

routes, ensuring input/output handling is reliable, and preparing the system for frontend interaction. This modular setup will allow seamless integration of Watsonx and LangChain responses while maintaining a clean API surface for the frontend.

Activity 3.1: Writing the Main Application Logic in main.py

1: Define the Core Routes in main.py

- Import and include routers for each of the SmartSDLC's functional areas: requirement analysis, code generation, testing, summarization, bug fixing, authentication, chatbot, and feedback.
- Use FastAPI's include_router() method to modularize the route definitions and separate concerns across files.
- Example router mounts:
    - /ai: Handles AI-based endpoints like code generation, test creation, bug fixing.
    - /auth: Manages login, registration, and user validation.
    - /chat: Powers the floating chatbot via LangChain.
    - /feedback: Handles submission and storage of feedback.



2: Set Up Route Handling and Middleware

FastAPI middleware is configured using CORSMiddleware to ensure the frontend (e.g., Streamlit) can interact with the backend without cross-origin issues. This is especially useful during development and should be fine-tuned in production.

3: Implement Application Metadata and Root Route

The FastAPI app is created with custom metadata such as a **project title, version**, and **description** to help identify the API on interactive Swagger docs (/docs).

```
# Local imports
from app.core.config import settings
from app.db.init_db import init_db
from app.routes import  chat_routes, ai_routes, auth_routes  , feedback_routes

app = FastAPI(
    title="SmartSDLC - AI-Enhanced Software Development Lifecycle",
    description="AI-powered microservice platform for accelerating SDLC using Watsonx, FastAPI, and GitHub in
    version="1.0.0"
)
```

Milestone 4: Frontend Development

Milestone 4 focuses on creating a **visually appealing** and **user-friendly interface** for SmartSDLC using **Streamlit**. The UI is designed to provide smooth navigation across SDLC functionalities like requirement classification, code generation, bug fixing, and more. This milestone ensures a responsive, consistent experience while leveraging AI outputs effectively, and includes a built-in chatbot for dynamic user interaction.

Activity 4.1: Designing and Developing the User Interface

1: Set Up the Base Streamlit Structure

- Create a main Home.py file that acts as the dashboard and entry point of the app.
- Add a welcoming hero section with a **Lottie animation**, a title, and a tagline.
- Organize features in a grid layout with clean navigation links to modular pages.

2: Design a Responsive Layout Using Streamlit Components

- Use st.columns(), st.container(), st.markdown() for layout control and consistency.
- Apply custom CSS styling for better fonts, backgrounds, and card shadows.
- Design a flexible layout that works well across different screen sizes and resolutions.

3: Create Separate Pages for Each Core Functionality

- Build feature-specific modules under the pages/ directory:
  - Upload_and_Classify.py: Upload PDF and classify requirements.
  - Code_Generator.py: Convert user prompts into working code.
  - Test_Generator.py: Generate test cases.
  - Bug_Fixer.py: Automatically fix buggy code.
  - Code_Summarizer.py: Summarize code into documentation.
  - Feedback.py: Collect user feedback.

- Connect each page to corresponding FastAPI endpoints via api_client.py.

Activity 4.2: Creating Dynamic Interaction with Backend

1: Integrate FastAPI with Streamlit for Real-Time Content

- Use requests.post() or requests.get() inside api_client.py to interact with backend routes like /ai/generate-code, /ai/upload-pdf, etc.
- Ensure user inputs like uploaded files or prompt text are formatted and passed properly.
- Display AI responses using st.code(), st.success(), and markdown blocks for clarity.

2: Embed a Smart Floating Chatbot

- Add a minimal inline chatbot in Home.py using a Streamlit form.
- Use the /chat/chat endpoint to send and receive real-time responses.
- Enhance interaction by showing emoji-based avatars and session memory.

Milestone 5: Deployment

In Milestone 5, the focus is on deploying the **SmartSDLC application locally** using FastAPI for the backend and Streamlit for the frontend. This involves setting up a secure environment, installing dependencies, connecting API keys for Watsonx, and launching the services to simulate a real-world workflow. This milestone helps ensure that the app can run smoothly in a local setup before shifting to cloud platforms or CI/CD pipelines.

Activity 5.1: Preparing the Application for Local Deployment

1: Set Up a Virtual Environment

- Create a Python virtual environment to manage dependencies and avoid conflicts with other projects.
- Activate the environment and install dependencies listed in requirements.txt to ensure all libraries (FastAPI, Streamlit, Watsonx SDK, etc.) are available.

2: Configure Environment Variables

- Set environment variables for sensitive data such as **IBM Watsonx API key**, **model IDs**, and database URLs.
- Create a .env file in your project root to securely store and load these settings during runtime.
    - WATSONX_API_KEY=your_ibm_key_here
    - WATSONX_PROJECT_ID=your_project_id
    - WATSONX_MODEL_ID=granite-20b-code-instruct

These values are loaded using python-dotenv inside your backend (e.g., config.py).

Activity 5.2: Testing and Verifying Local Deployment

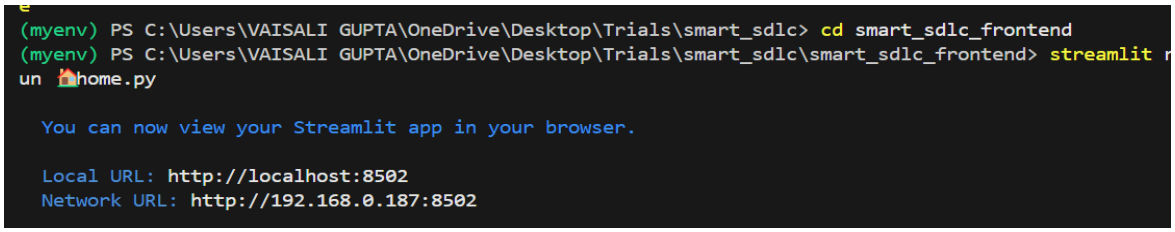
1: Launch and Access the Application Locally

- **Start the FastAPI backend** using Uvicorn: uvicorn app.main:app --reload
- **Run the Streamlit frontend**: streamlit run frontend/Home.py

Open your browser and navigate to:

- **Streamlit UI:** http://localhost:8502
- **FastAPI Swagger Docs:** http://127.0.0.1:8000/docs
- Test each SmartSDLC feature (requirement upload, code generation, bug fixing, chatbot, feedback) to ensure everything is connected and functioning correctly.

- **Run the Web Application**
- Now type “streamlit run ????home.py” command
- Navigate to the localhost where you can view your web page



Milestone 6: Conclusion

The **SmartSDLC** platform represents a significant advancement in the automation of the Software Development Lifecycle by integrating **AI-powered intelligence** into each phase—from requirement analysis to code generation, testing, bug fixing, and documentation. By leveraging cutting-edge technologies like **IBM Watsonx**, **FastAPI**, **LangChain**, and **Streamlit**, the system demonstrates how generative AI can streamline traditional software engineering tasks, reduce manual errors, and accelerate development timelines.

The platform's modular architecture and intuitive interface empower both technical and non-technical users to interact with SDLC tasks efficiently. Features such as requirement classification from PDFs, AI-generated user stories, code generation from natural language, auto test case generation, smart bug fixing, and integrated chat assistance illustrate the power of AI when applied thoughtfully within a development framework.

Overall, SmartSDLC not only improves productivity and accuracy but also sets the foundation for future enhancements like CI/CD integration, team collaboration, version control, and cloud deployment. It is a step toward building **intelligent, developer-friendly ecosystems** that support modern agile development needs with smart automation at its core.

The web application will open in the web browser:

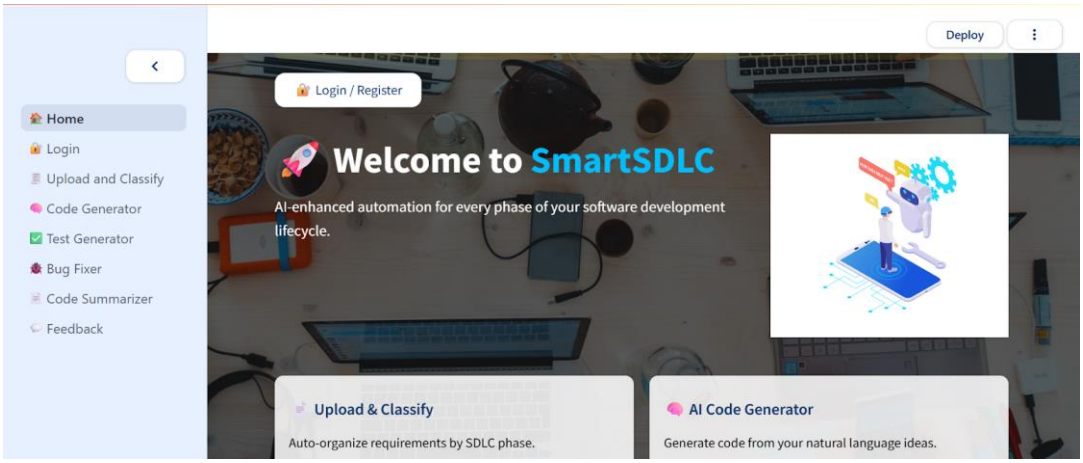


Figure 3: "SmartSDLC Dashboard – Your AI-powered command center for accelerating every phase of the software development lifecycle."

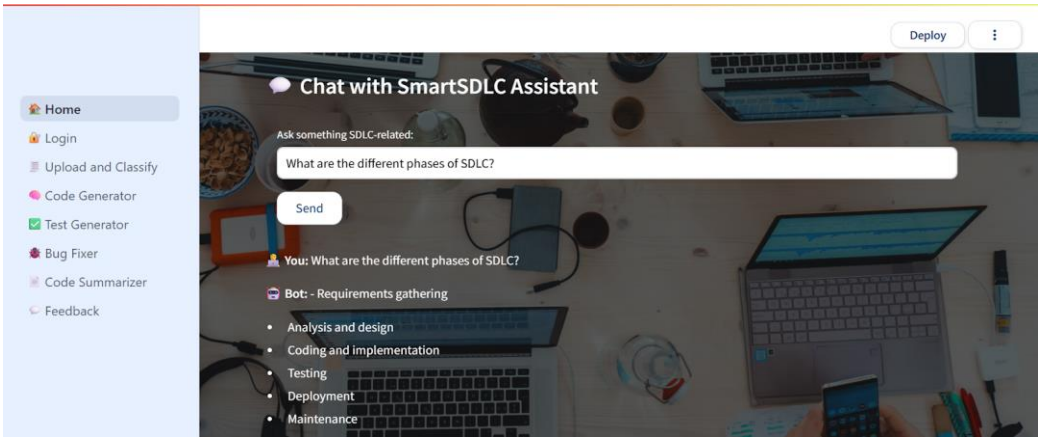


Figure 4: "Interactive Chat with SmartSDLC Assistant – Instantly learn SDLC concepts through AI-powered conversations."

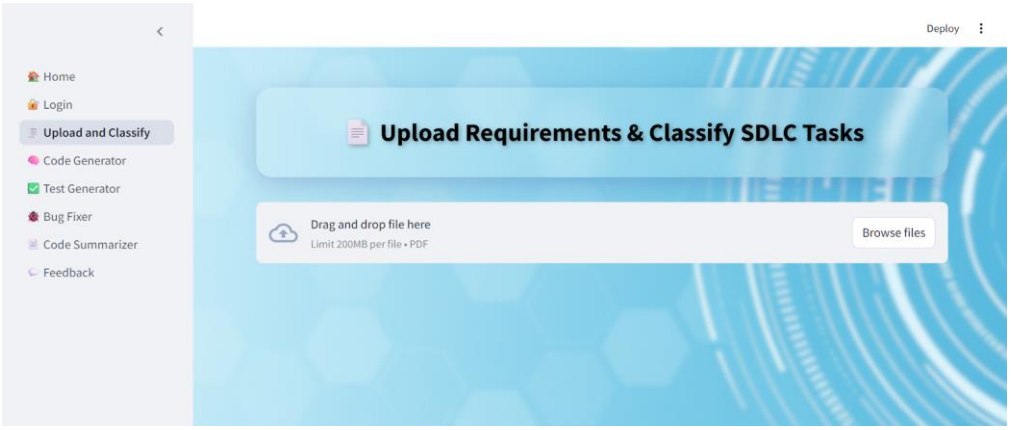


Figure 5: Input: "Upload and Classify – Effortlessly convert raw PDF requirements into structured SDLC tasks using AI."

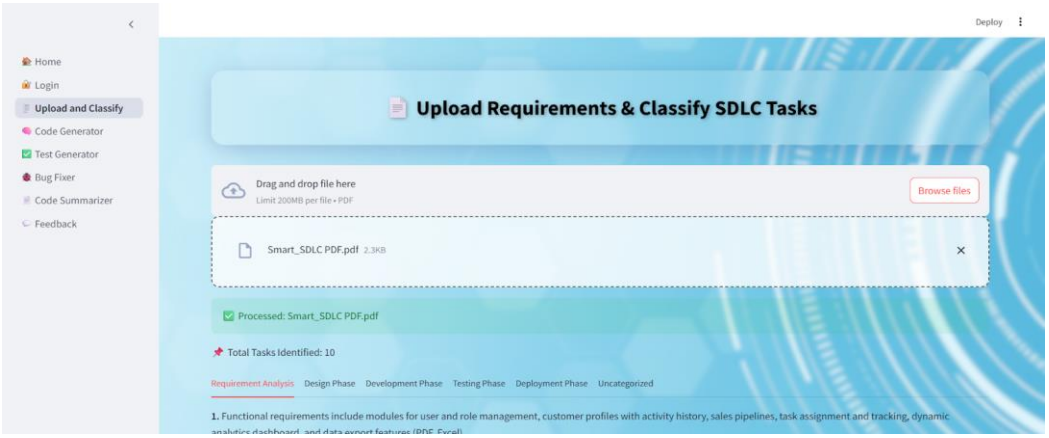


Figure 6: Output: "Smart Requirement Classifier – Instantly extract and categorize software tasks into SDLC phases from your uploaded PDF."

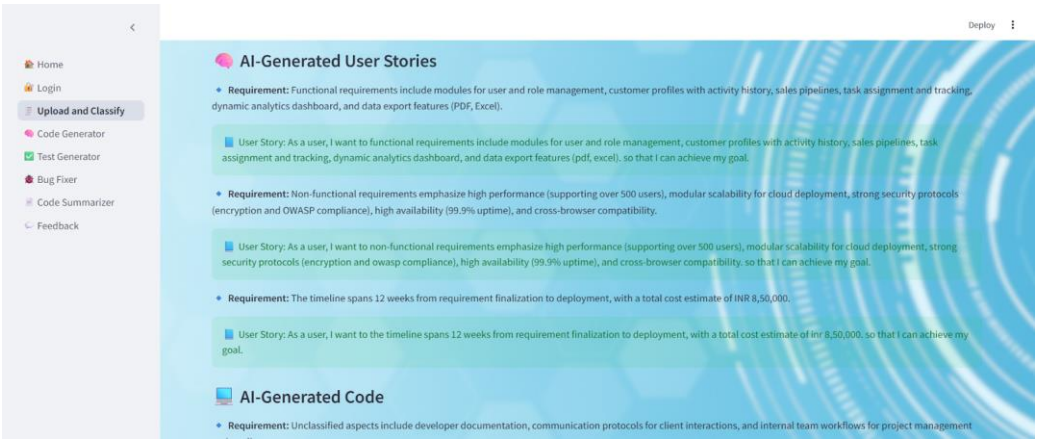


Figure 7: "AI-Generated User Stories – Transform raw requirements into structured user-centric stories for agile development."

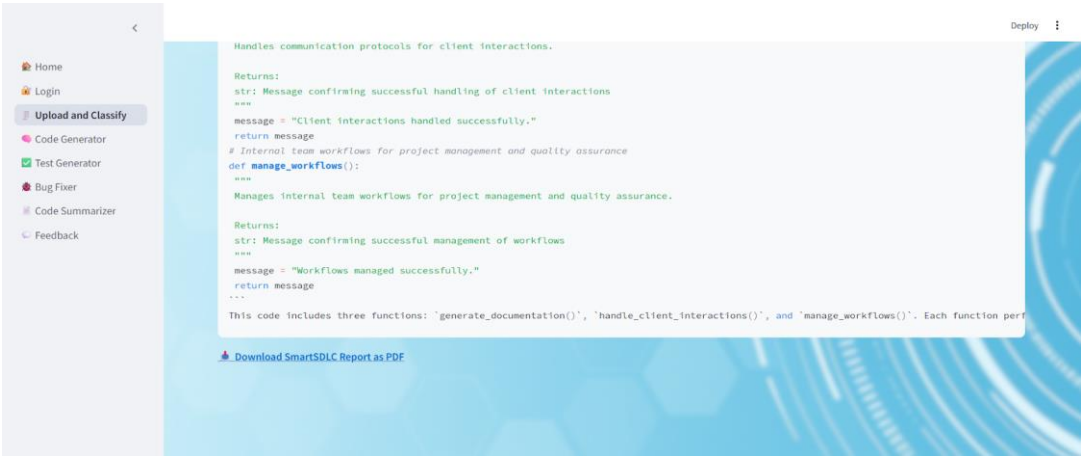


Figure 8: "Auto-Generated Summary with Downloadable Insights – Review AI-explained code and export the SmartSDLC report as a professional PDF."

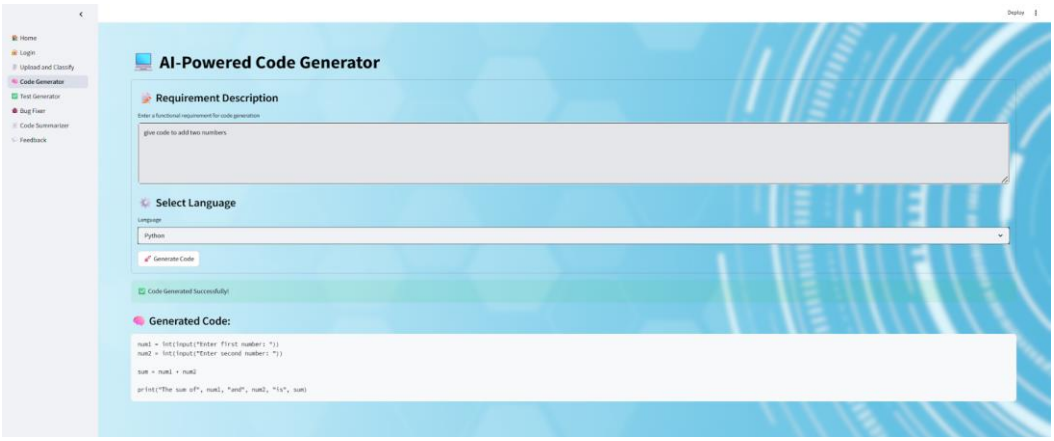


Figure 9: "AI Code Generator – Instantly convert natural language requirements into clean, executable code with just one click."

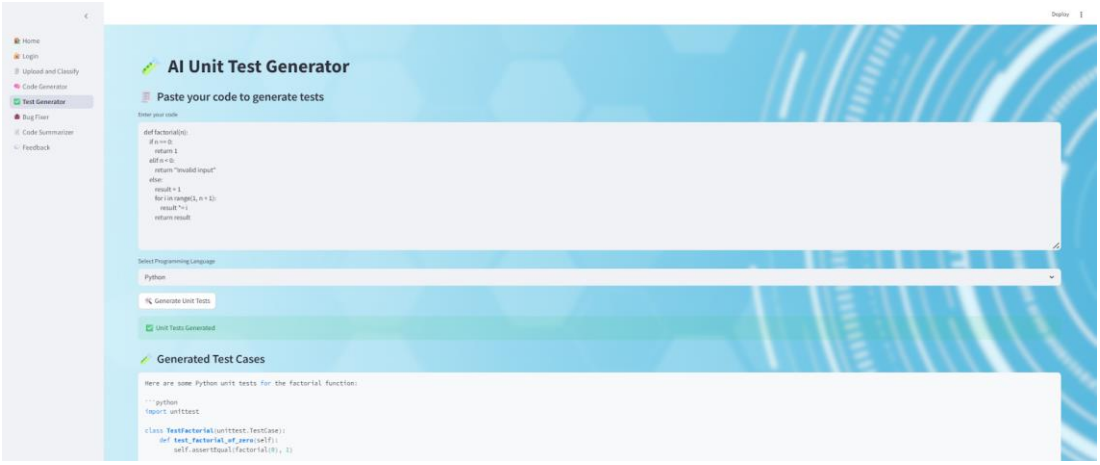


Figure 10: AI-powered tool to auto-generate unit tests from user-provided code.

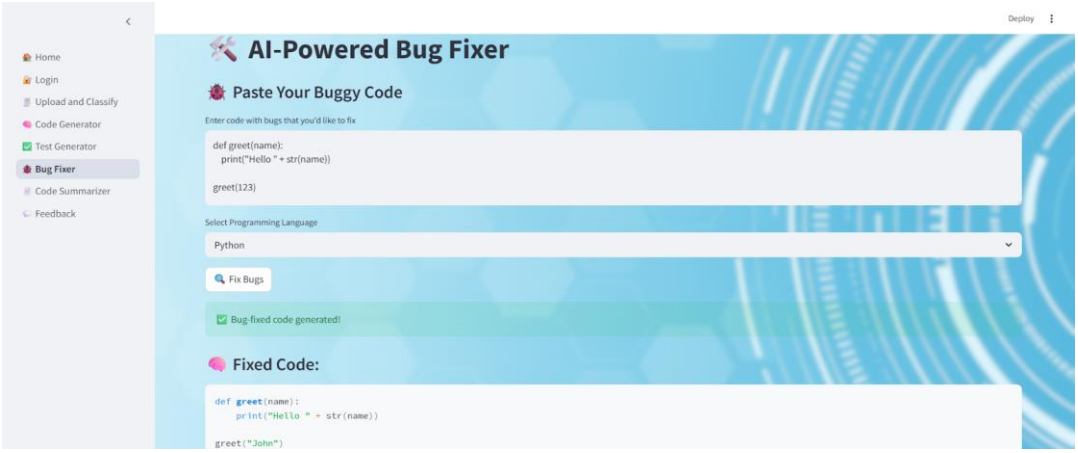


Figure 11: "AI-Powered Bug Fixer – Instantly detect and correct code issues for cleaner, error-free execution."

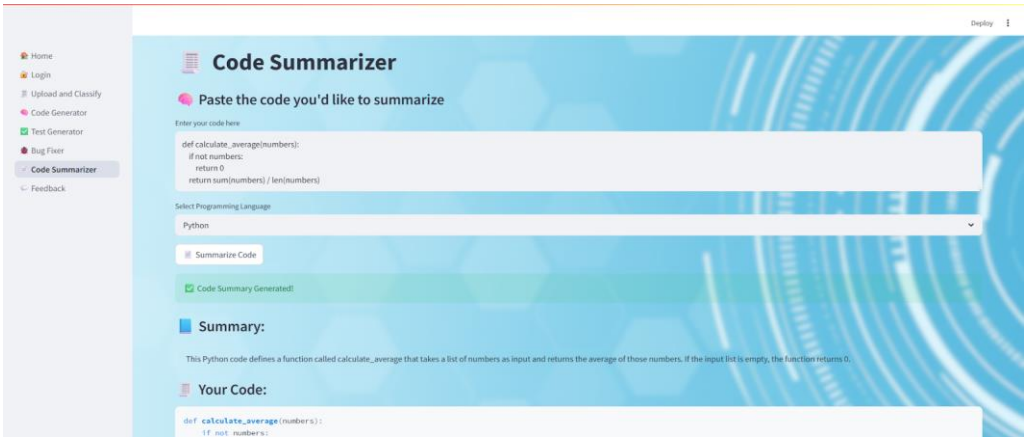


Figure 12: “AI-powered Code Summarizer interface displaying a generated summary for the provided code snippet.”

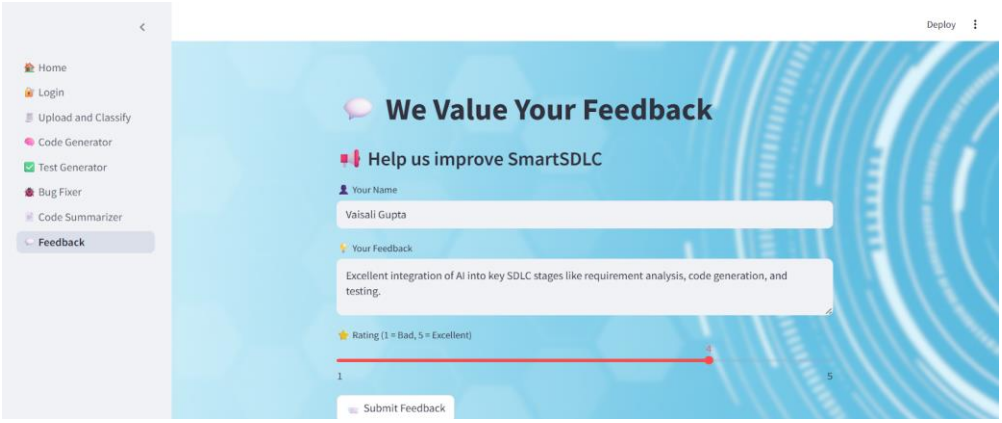


Figure 13: "User submitting feedback on the SmartSDLC system, highlighting effective AI integration across SDLC phases"

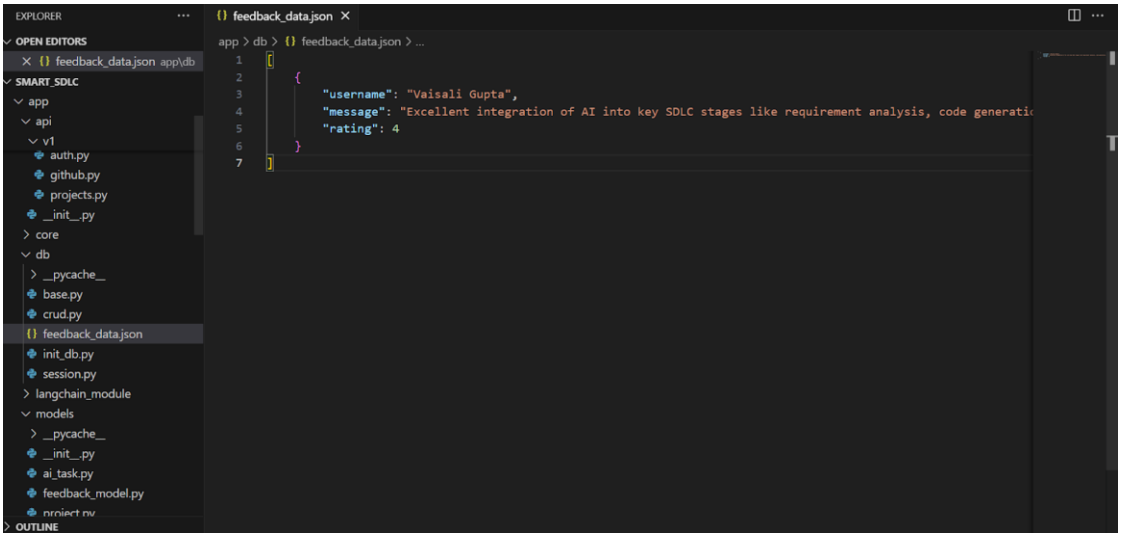


Figure 14:User feedback data stored in feedback_data.json file within the SmartSDLC application's database module.

✨ Tech Stack

- **Frontend**: React.js / Next.js
- **Backend**: Python (FastAPI) or Node.js (Express)
- **AI Models**: OpenAI GPT / BERT / Custom ML models
- **Database**: MongoDB or PostgreSQL
- **CI/CD**: GitHub Actions, Docker
- **Authentication**: JWT-based Auth
- **Testing**: Jest, PyTest, Postman/Newman

⚙️ 3. Setup Instructions

Prerequisites

- Node.js (if frontend/backend is in JS)
- Python 3.9+ (for AI and backend)
- Docker & Docker Compose (optional)
- MongoDB or PostgreSQL installed
- `.env` file with environment variables

Installation Steps

```
# Clone the repository
git clone https://github.com/your-org/smartsdlc.git
cd smartsdlc

# Backend Setup
cd backend
pip install -r requirements.txt

# Frontend Setup
cd ../frontend
npm install
```

AI Model Setup (optional if not using API-based LLMs)

- Download models or connect OpenAI/GPT APIs
- Set API keys in `.env` files

🗂️ 4. Folder Structure

```
smartsdlc/
│
├── backend/
│   ├── app/
│   │   ├── api/                # FastAPI routes
│   │   ├── services/           # Business logic, AI modules
│   │   ├── models/             # Pydantic models or ORM
│   │   ├── utils/              # Helper functions
│   │   └── main.py             # App entrypoint
│   ├── tests/                  # Backend tests
│   └── requirements.txt
```

```
│
├── frontend/
│   ├── components/              # Reusable UI components
│   ├── pages/                   # Pages (Next.js structure)
│   ├── services/                # API calls
│   ├── styles/
│   └── app.js / main.jsx
│
├── docs/                        # API docs, architecture diagrams
├── docker-compose.yml
├── README.md
└── .env.example
```

▶️ 5. Running the Application

Start Backend

```
cd backend
uvicorn app.main:app --reload
```

Or, if using Node.js:

```
cd backend
npm start
```

Start Frontend

```
cd frontend
npm run dev
```

Docker (optional)

```
docker-compose up --build
```

📄 6. API Documentation

API docs are auto-generated using **Swagger** (if FastAPI) or **Postman Collection**.

Swagger (FastAPI)

- Visit: http://localhost:8000/docs

Sample Endpoints

Method	Endpoint	Description
POST	`/api/requirements/analyze`	Extracts requirements from text
POST	`/api/design/generate`	Generates architecture design
POST	`/api/code/generate`	Generates code from design
POST	`/api/test/generate`	Generates test cases
GET	`/api/monitor/status`	Monitors deployed app

🔐 7. Authentication

Uses **JWT-based Auth** to protect routes.

Auth Flow

1. User signs up or logs in.
2. Backend issues JWT token.
3. Token is stored in browser (cookies or localStorage).
4. Token is sent in `Authorization` header for secure API calls.

```
Authorization: Bearer <your_token_here>
```

👨‍💻 8. User Interface

Frontend Features

- 🌐 Dashboard for managing projects
- 📝 Requirement input UI (text or voice)
- 🧩 Visual design generator (UML diagrams)
- 🧠 AI-generated code viewer
- 💉 Test case editor
- 🚀 Deployment status + logs viewer
- 🔒 Login/Register with JWT Auth

Tools

- React + Tailwind CSS
- Chart.js / Mermaid.js for visual diagrams
- Monaco editor for code display

🧪 9. Testing

Backend Testing

- **PyTest** or **unittest**
- Test API routes, services, and AI modules

```
cd backend
pytest
```

Frontend Testing

- **Jest** + **React Testing Library**

```
cd frontend
npm test
```

API Testing

- **Postman** or **Newman** scripts to test endpoints
- Included in `/docs/api/` folder

☑️ Summary

Feature	Description
🧠 AI Assistance	Integrated AI at each SDLC phase
🌐 Full-stack App	Frontend + Backend + DB + AI
📘 API Docs	Auto-generated with Swagger
🔐 Auth	JWT-secured API access
🧪 Testing	Unit + Integration + API

🚀 DevOps Ready Docker + CI/CD pipelines