SMARTSDLC – AI-ENHANCED SOFTWARE DEVELOPMENT LIFECYCLE

The **SmartSDLC** project is an AI-enhanced platform designed to automate and streamline the **Software Development Lifecycle (SDLC)** using advanced technologies like **IBM Watsonx**, **FastAPI**, **LangChain**, and **Streamlit**. It integrates **generative AI** to handle key SDLC phases, including requirement analysis, code generation, test case creation, bug fixing, and documentation. The platform features a **user-friendly interface** that allows users to upload PDFs, generate structured requirements, and transform natural language prompts into functional code. It also includes an **AI-powered chatbot** for real-time assistance and support. The backend, built with **FastAPI**, efficiently processes API requests, while the frontend, developed using **Streamlit**, offers a visually appealing and interactive dashboard. The system is modular, scalable, and secure, featuring a robust authentication mechanism and seamless integration between AI models and user inputs. Deployment is streamlined with local hosting via **Uvicorn** and **Streamlit**, with comprehensive API documentation available through **Swagger UI**. By leveraging AI in a practical and efficient manner, SmartSDLC significantly reduces manual workload, accelerates development processes, and enhances software quality, making it an invaluable tool for developers and project teams.

Scenario 1: Requirement Upload and Classification

Requirement Upload and Classification, the platform simplifies the complex task of requirement gathering by allowing users to upload PDF documents containing raw, unstructured text. The backend extracts content using PyMuPDF and leverages IBM Watsonx's Granite-20B AI model to classify each sentence into specific SDLC phases such as Requirements, Design, Development, Testing, or Deployment. These classified inputs are then transformed into structured user stories, enabling clear planning and traceability. The frontend displays this output in an organized, readable format grouped by phase, significantly improving clarity and saving manual effort.

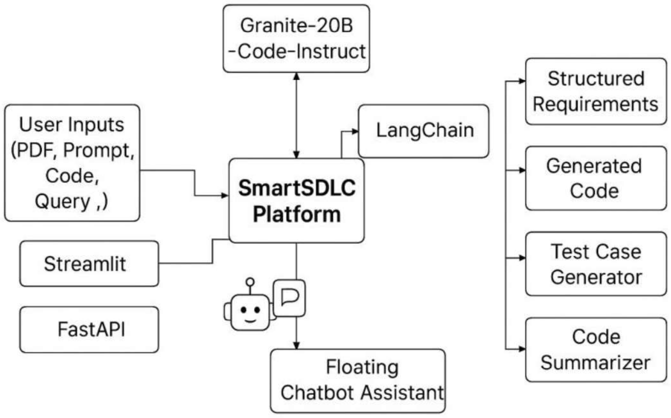
Scenario 2: AI Code Generator

AI Code Generator, addresses the development phase, where developers can input natural language prompts or structured user stories. These prompts are sent to the Watsonx model, which generates contextually relevant, production-ready code. This reduces the time needed for boilerplate or prototype creation and enhances coding efficiency. The code is presented in a clean, syntax-highlighted format on the frontend, ready for use or further enhancement.

Scenario 3: Bug Fixer

Bug Fixer, the platform supports debugging by accepting code snippets in languages such as Python or JavaScript. Upon receiving the buggy code, the Watsonx AI analyzes it for both syntactical and logical errors and returns an optimized version. This not only assists developers in identifying mistakes without extensive manual reviews but also provides immediate, corrected code directly in the frontend for comparison.

Architecture



Pre-requisites

- **Python 3.10 : https://www.python.org/downloads/release/python-3100/**
- **FastAPI : https://fastapi.tiangolo.com/**
- **Streamlit : https://docs.streamlit.io/**
- **IBM Watsonx AI & Granite Models:https://www.ibm.com/products/watsonx-ai/foundation-model**
- **LangChain :https://www.langchain.com/**
- **Uvicorn: https://www.uvicorn.org/**
- **PyMuPDF (fitz): https://pymupdf.readthedocs.io/en/latest/**
- **Git & GitHub: https://www.w3schools.com/git/git_intro.asp?remote=github**
- **Frontend Libraries**

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

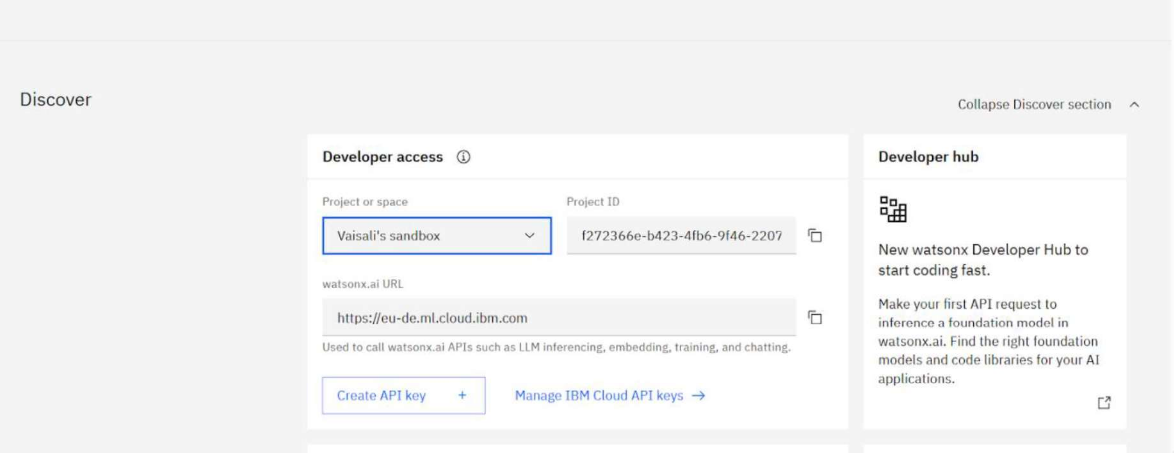
1. **Understand the Project Requirements:**
   Analyze the goals of the SmartSDLC platform, particularly the need for automating SDLC phases like requirement analysis, code generation, test creation, bug fixing, and documentation. This understanding helps narrow down which generative AI models best suit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



Project or space: Vaisali's sandbox

Project ID: f272366e-b423-4fb6-9f46-2207

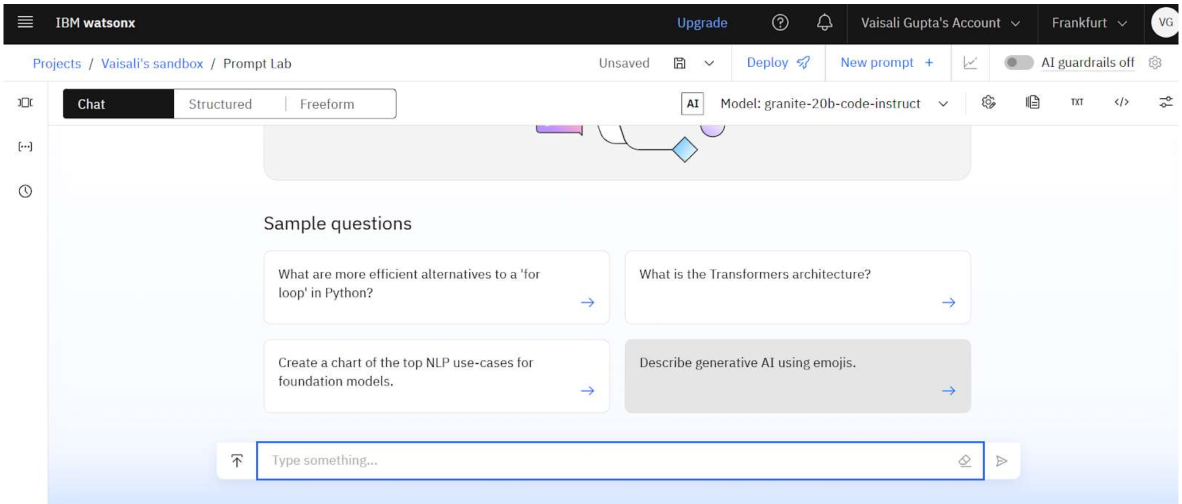
watsonx.ai URL: https://eu-de.ml.cloud.ibm.com

Used to call watsonx.ai APIs such as LLM inferencing, embedding, training, and chatting.

New watsonx Developer Hub to start coding fast.

Make your first API request to inference a foundation model in watsonx.ai. Find the right foundation models and code libraries for your AI applications.

6. **Selecting the model**



Model: granite-20b-code-instruct

Sample questions

What are more efficient alternatives to a 'for loop' in Python?

What is the Transformers architecture?

Create a chart of the top NLP use-cases for foundation models.

Describe generative AI using emojis.

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:
   1. PDF processing and classification logic (using PyMuPDF).
   2. Prompt generation and API communication with Watsonx.
   3. LangChain agent for orchestration.
   4. 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:

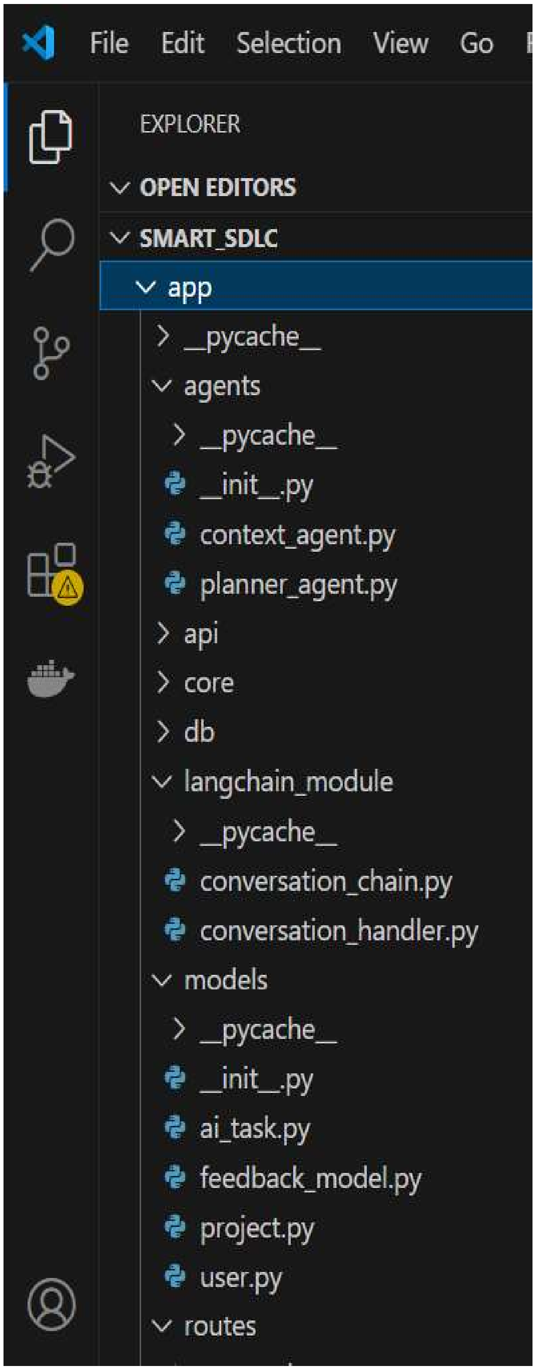
   python -m venv myenv
   myenv\Scripts\activate

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



```
EXPLORER
∨ OPEN EDITORS
∨ SMART_SDLC
  ∨ app
    > __pycache__
    ∨ agents
      > __pycache__
      __init__.py
      context_agent.py
      planner_agent.py
    > api
    > core
    > db
    ∨ langchain_module
      > __pycache__
      conversation_chain.py
      conversation_handler.py
    ∨ models
      > __pycache__
      __init__.py
      ai_task.py
      feedback_model.py
      project.py
      user.py
    ∨ routes
```

Milestone 2: Core Functionalities Development

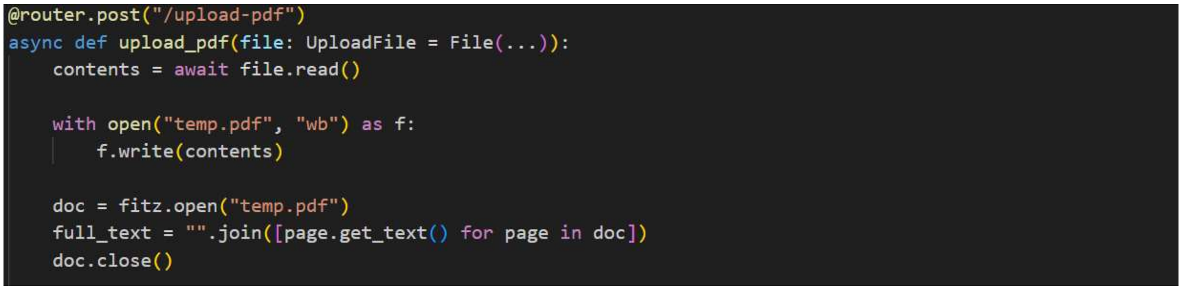
Milestone 2 involves building the core AI-driven features of SmartSDLC, including requirement analysis, code generation, test case creation, bug fixing, code summarization, chatbot assistance, and GitHub integration. These functionalities are implemented using Watsonx and LangChain via modular service 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:

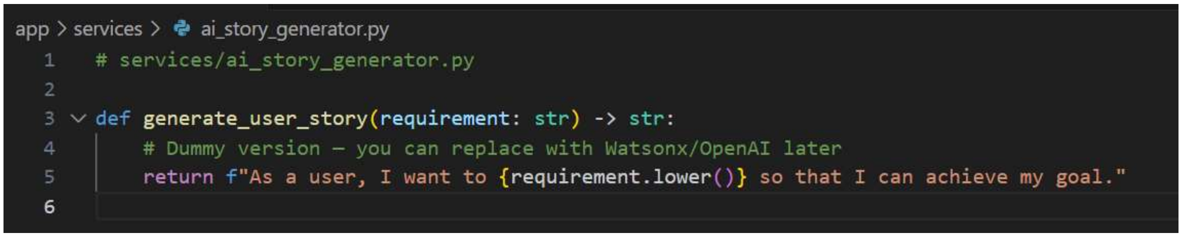
This activity focuses on building the logic behind each AI-powered SDLC feature. These functionalities are the heart of SmartSDLC and are implemented using IBM Watsonx and LangChain integration.

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.



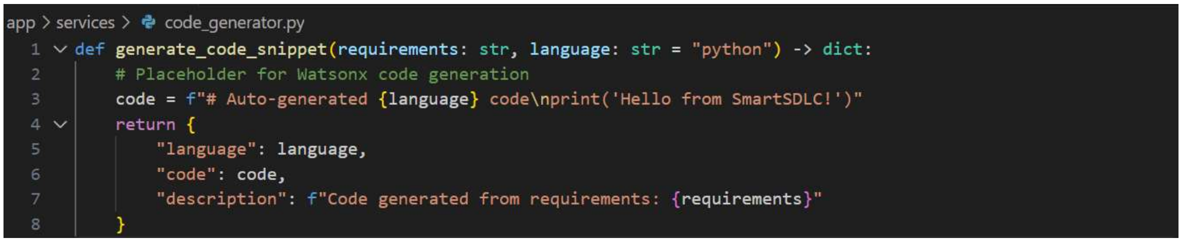
```
@router.post("/upload-pdf")
async def upload_pdf(file: UploadFile = File(...)):
    contents = await file.read()

    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()
```



```
# 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.



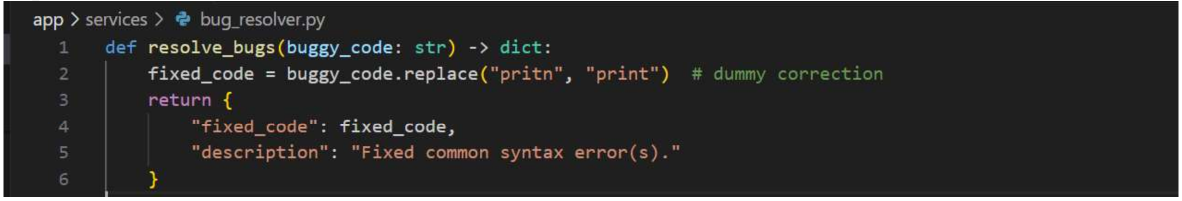
```
def generate_code_snippet(requirements: str, language: str = "python") -> dict:
    # Placeholder for Watsonx code generation
    code = f"# Auto-generated {language} code\nprint('Hello from SmartSDLC!')"
    return {
        "language": language,
        "code": code,
        "description": f"Code generated from requirements: {requirements}"
    }
```

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

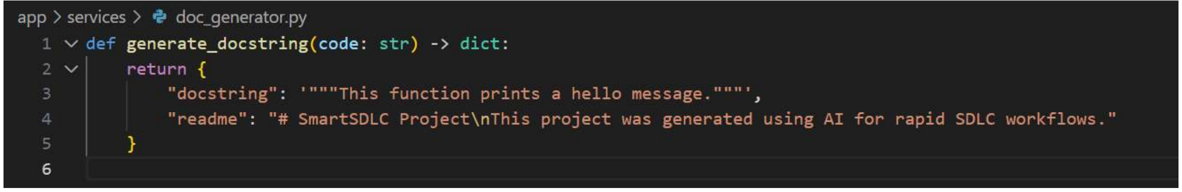
- **Module:** bug_resolver.py
- **Description:** Takes buggy code snippets and automatically identifies and fixes errors using AI. Returns a clean version of the code along with optional explanations.



```python
def resolve_bugs(buggy_code: str) -> dict:
    fixed_code = buggy_code.replace("pritn", "print")  # dummy correction
    return {
        "fixed_code": fixed_code,
        "description": "Fixed common syntax error(s)."
    }
```

5. Code Summarization and Documentation

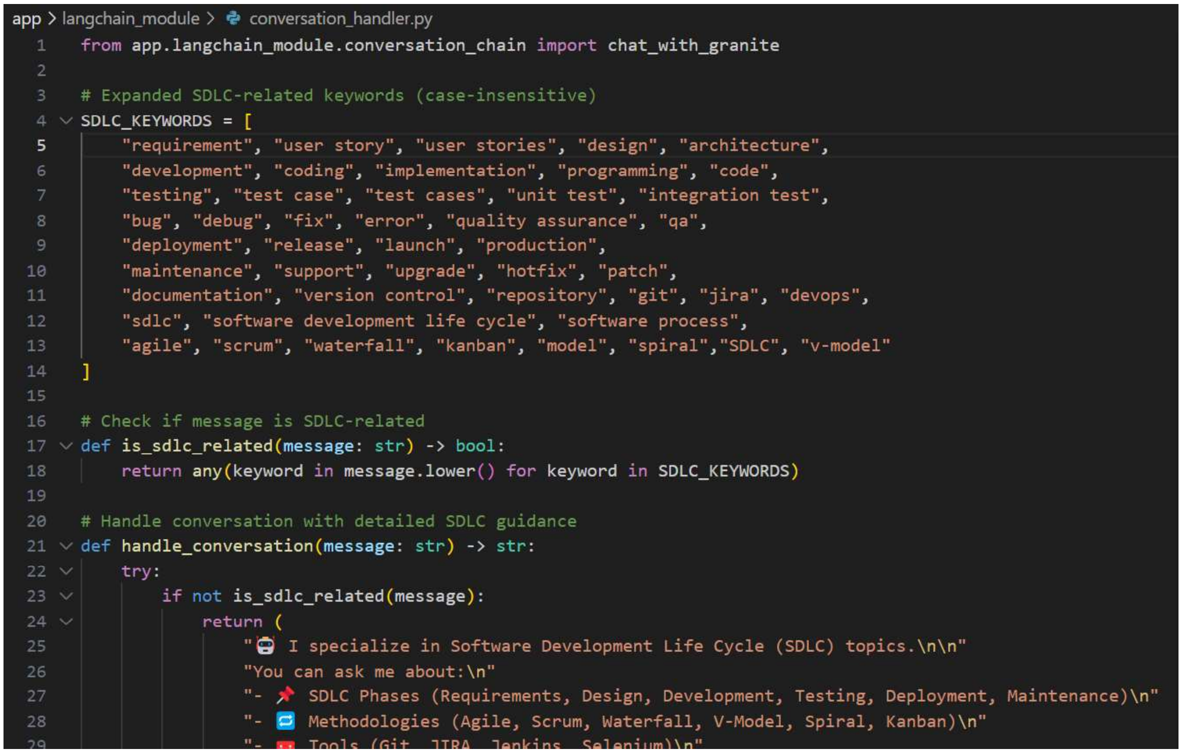
- **Module:** doc_generator.py
- **Description:** Extracts summaries or generates documentation-style descriptions of code blocks. Helps with code readability and maintainability.



```python
def generate_docstring(code: str) -> dict:
    return {
        "docstring": '"""This function prints a hello message."""',
        "readme": "# SmartSDLC Project\nThis project was generated using AI for rapid SDLC workflows."
    }

```

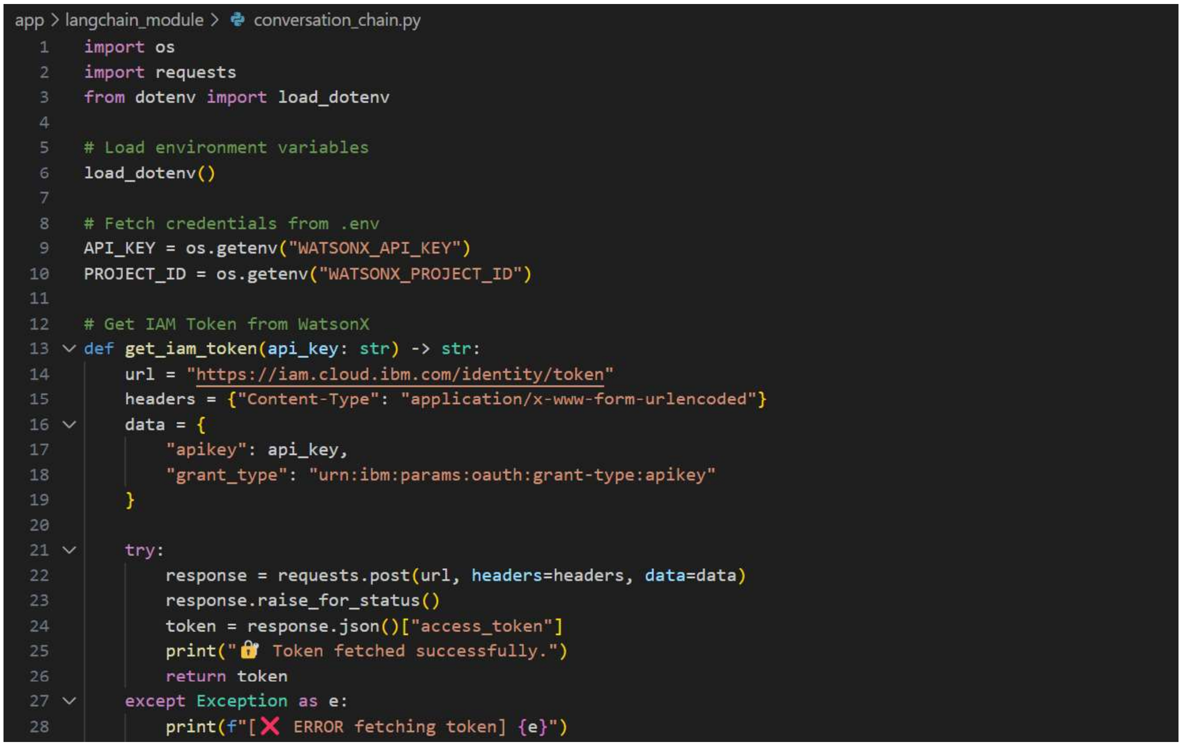
6. Chatbot Assistance

- **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"
```

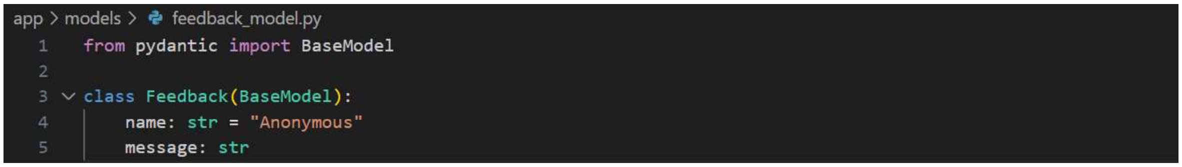


app > langchain_module > conversation_chain.py

```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}")
```

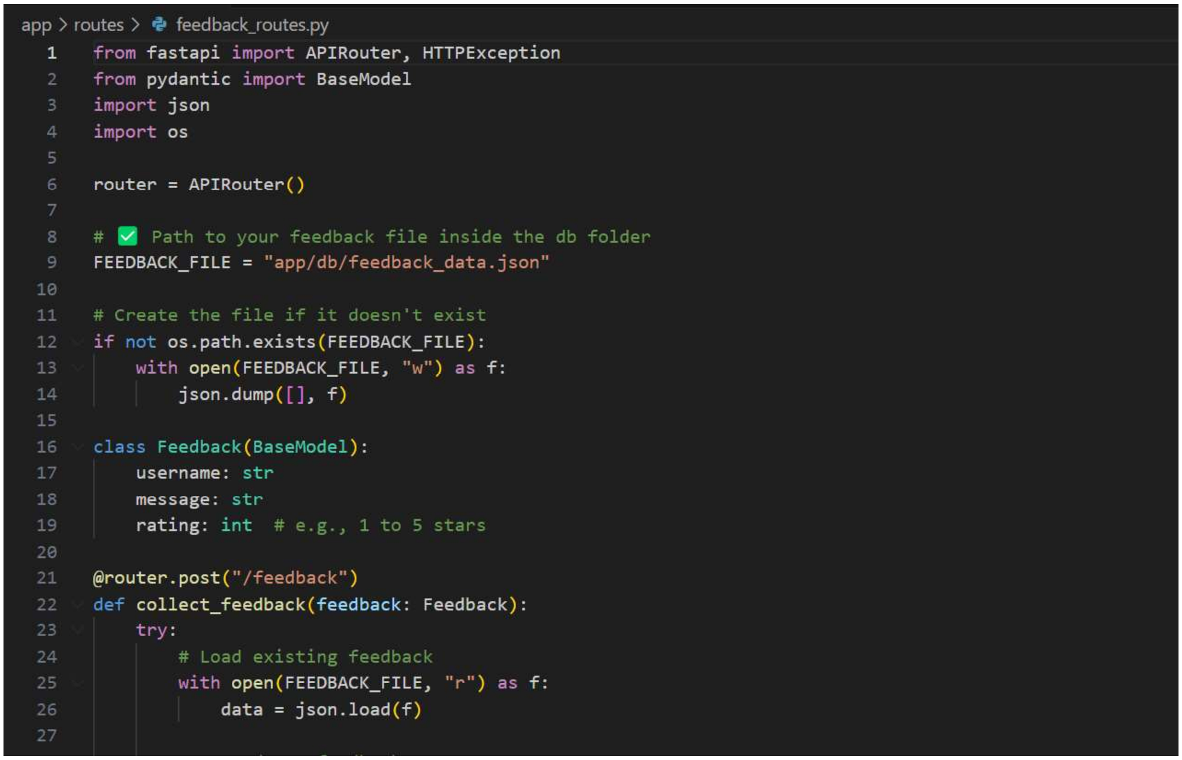
7. Project Feedback and Learning Loop

- **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
```



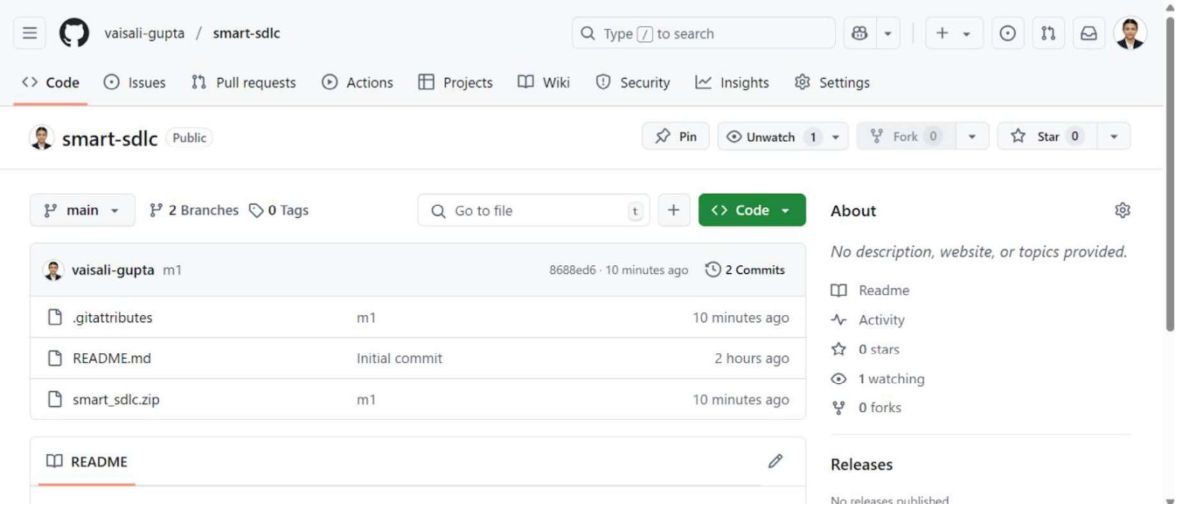
app > routes > feedback_routes.py

```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

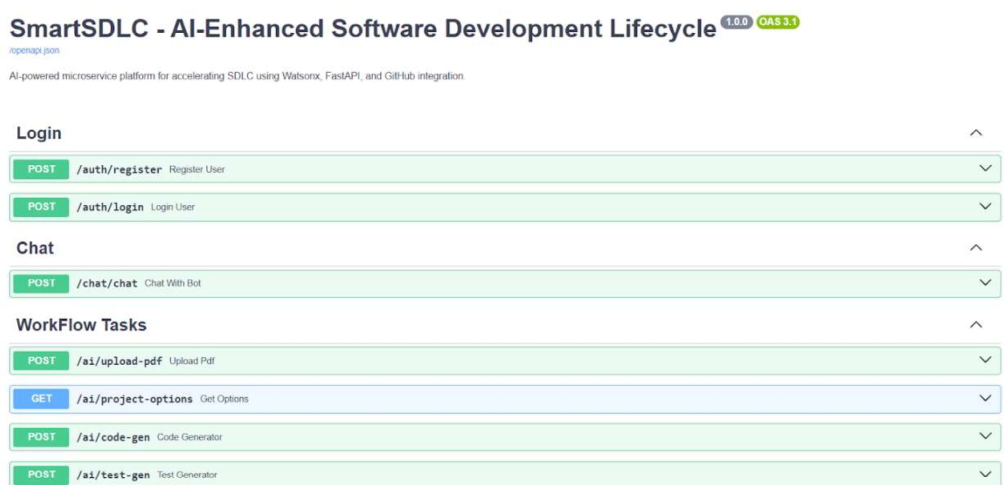
- **Description:** Automates GitHub workflows such as pushing AI-generated code to a repo, opening issues, or syncing generated documentation.



vaisali-gupta m1 8688ed6 · 10 minutes ago 2 Commits

.gitattributes	m1	10 minutes ago
README.md	Initial commit	2 hours ago
smart_sdlc.zip	m1	10 minutes ago

Output: Fast API Backend


SmartSDLC - AI-Enhanced Software Development Lifecycle 1.0.0 OAS 3.1

/openapi.json

AI-powered microservice platform for accelerating SDLC using Watsonx, FastAPI, and GitHub integration

Login
POST /auth/register Register User
POST /auth/login Login User
Chat
POST /chat/chat Chat With Bot
WorkFlow Tasks
POST /ai/upload-pdf Upload Pdf
GET /ai/project-options Get Options
POST /ai/code-gen Code Generator
POST /ai/test-gen Test Generator

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

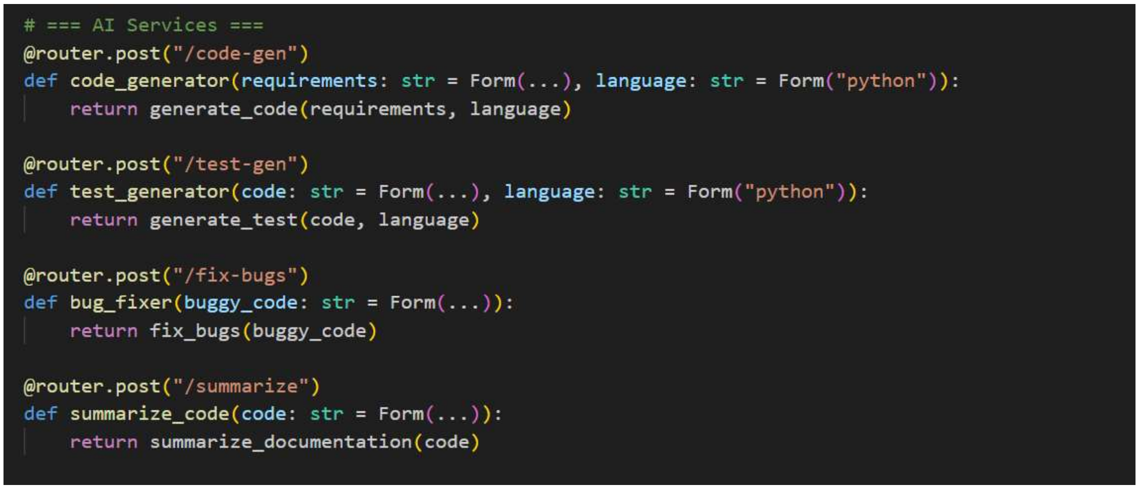
Activity 2.2: Implement the FastAPI backend to manage routing and user input processing, ensuring smooth API interactions.

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.



```
# === 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

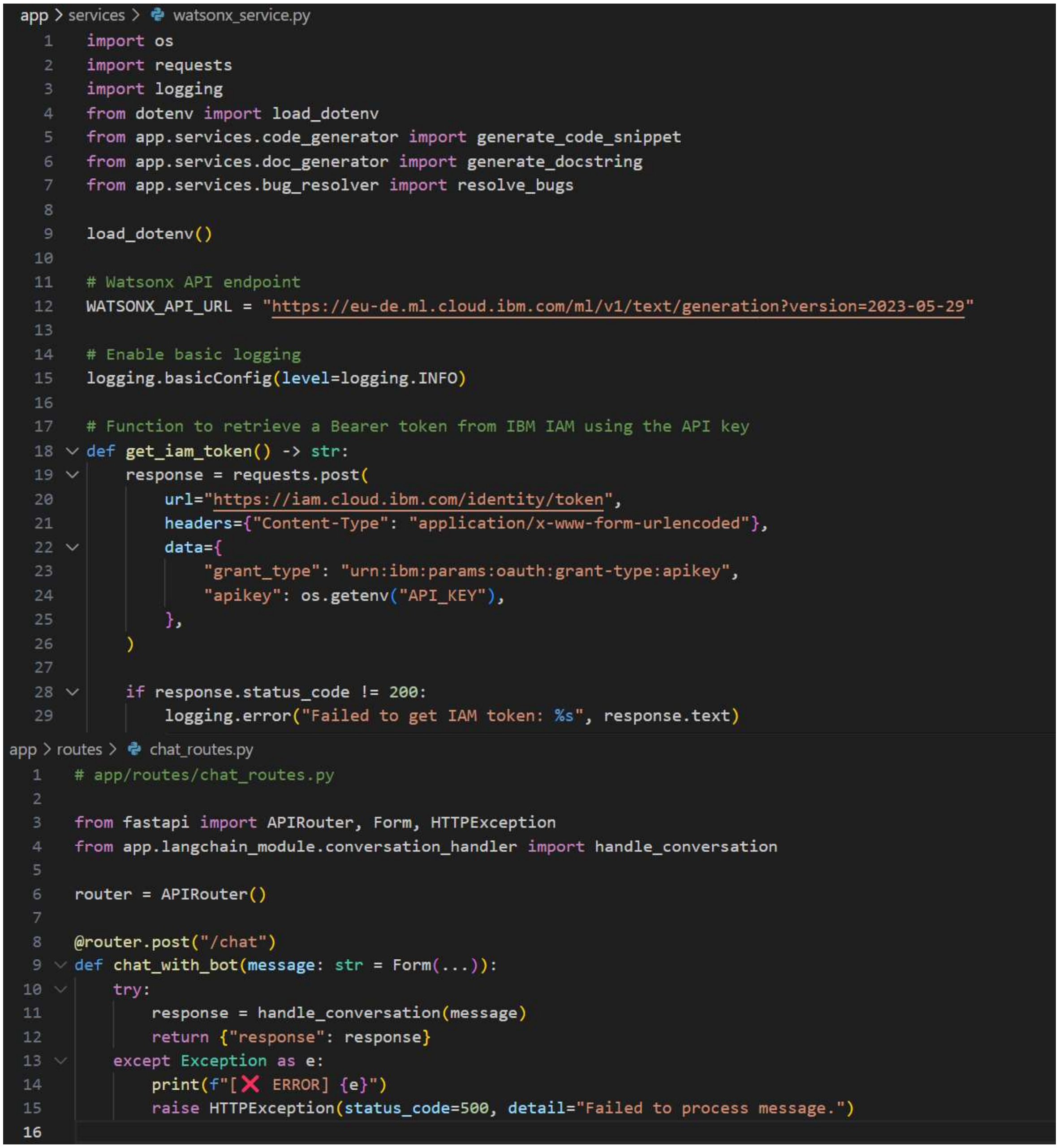
- **Modules:** auth_routes.py, security.py, user.py
- **Description: Manages secure user login and registration with hashed passwords. Each request checks the user's session or credentials before executing the corresponding service.**

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.



```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)
```

```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.")

```

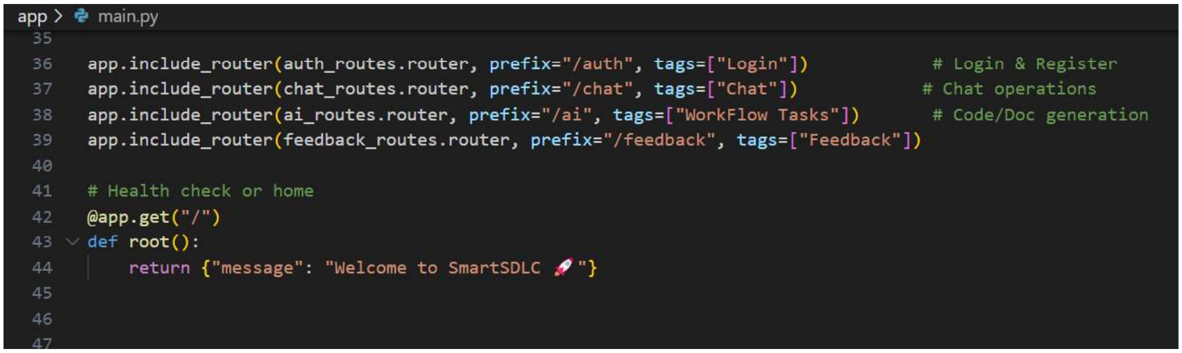
Milestone 3: main.py Development

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

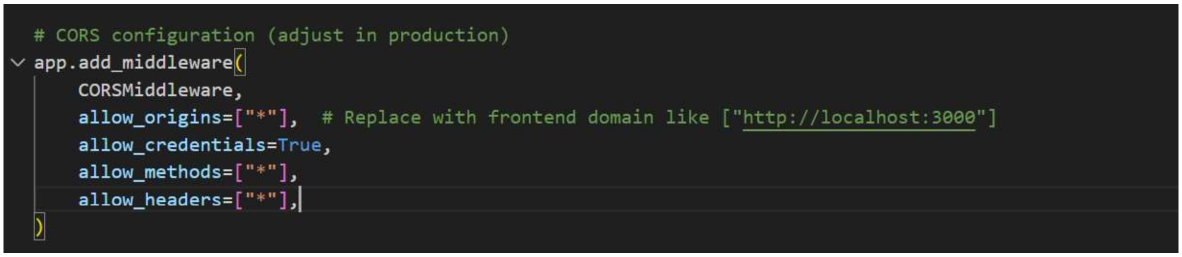
- Import and include routers for each of the SmartSDLC's functional areas: requirement analysis, code generation, testing, summarization, bug fixing, authentication, chatbot, and feedback.
- Use FastAPI's include_router() method to modularize the route definitions and separate concerns across files.
- 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.



```python

app.include_router(auth_routes.router, prefix="/auth", tags=["Login"])            # Login & Register
app.include_router(chat_routes.router, prefix="/chat", tags=["Chat"])             # Chat operations
app.include_router(ai_routes.router, prefix="/ai", tags=["WorkFlow Tasks"])       # Code/Doc generation
app.include_router(feedback_routes.router, prefix="/feedback", tags=["Feedback"])

# Health check or home
@app.get("/")
def root():
    return {"message": "Welcome to SmartSDLC 🚀"}



```

2: Set Up Route Handling and Middleware

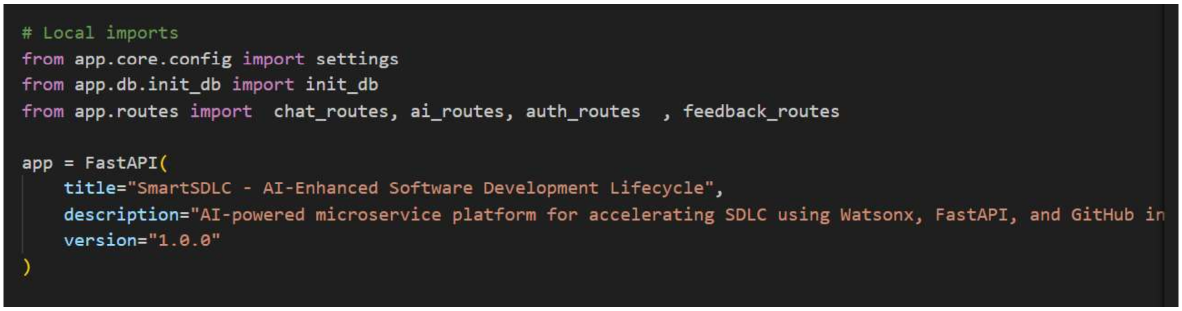
FastAPI middleware is configured using CORSMiddleware to ensure the frontend (e.g., Streamlit) can interact with the backend without cross-origin issues. This is especially useful during development and should be fine-tuned in production.



```python
# CORS configuration (adjust in production)
app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],  # Replace with frontend domain like ["http://localhost:3000"]
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)
```

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).



```python
# 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

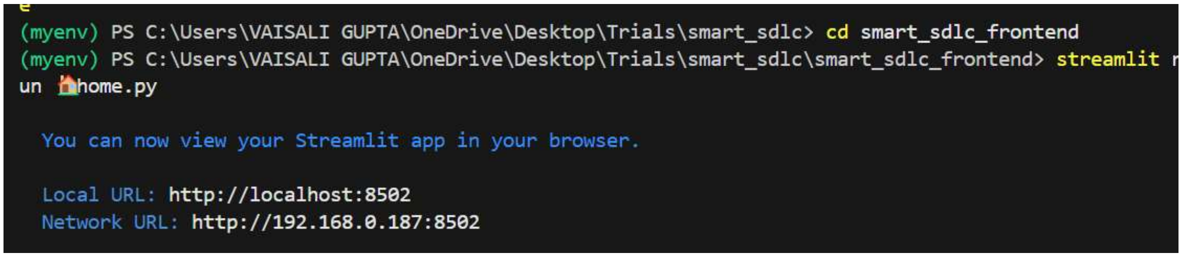
1: Launch and Access the Application Locally

- **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



```
e
(myenv) PS C:\Users\VAISALI GUPTA\OneDrive\Desktop\Trials\smart_sdlc> cd smart_sdlc_frontend
(myenv) PS C:\Users\VAISALI GUPTA\OneDrive\Desktop\Trials\smart_sdlc\smart_sdlc_frontend> streamlit r
un 🏠home.py

  You can now view your Streamlit app in your browser.

  Local URL: http://localhost:8502
  Network URL: http://192.168.0.187:8502
```

Milestone 6: Conclusion

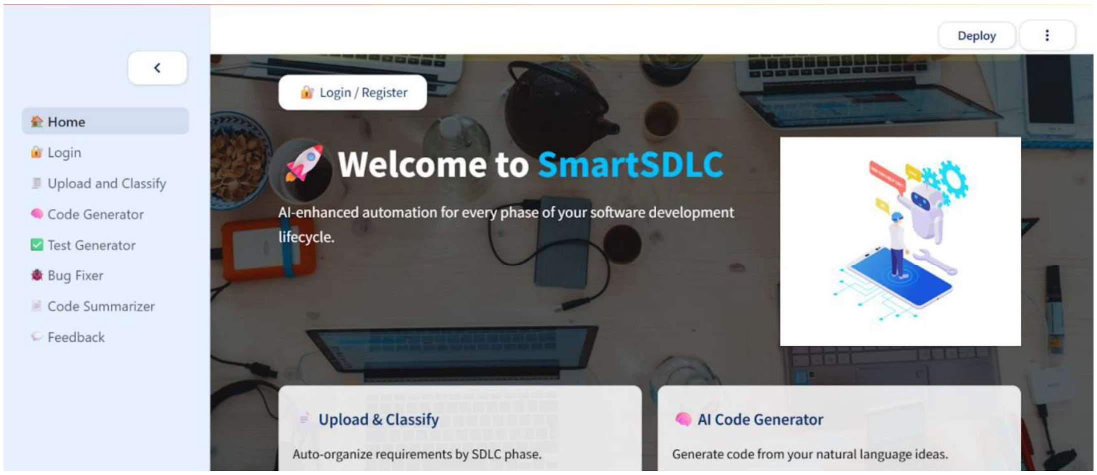
The **SmartSDLC** platform represents a significant advancement in the automation of the Software Development Lifecycle by integrating **AI-powered intelligence** into each phase—from requirement analysis to code generation, testing, bug fixing, and documentation. By leveraging cutting-edge technologies 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:



Welcome to SmartSDLC

AI-enhanced automation for every phase of your software development lifecycle.

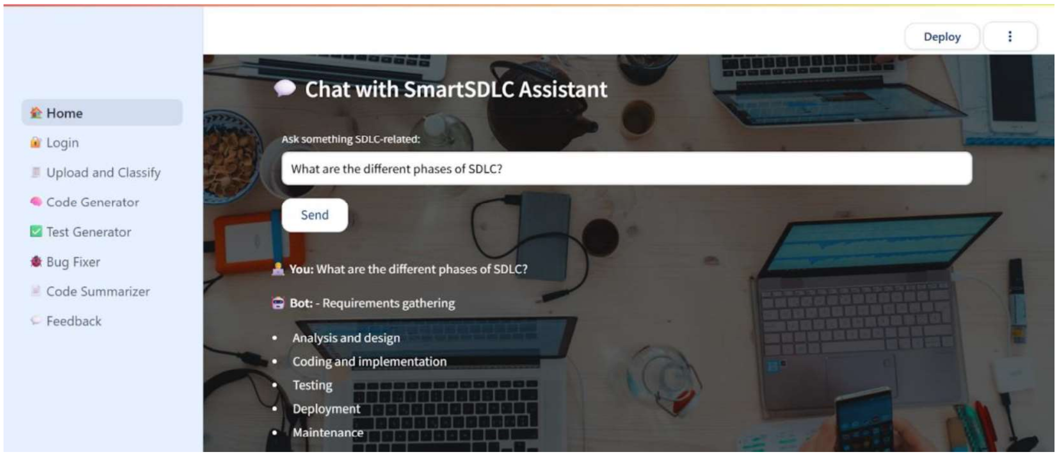
Upload & Classify

Auto-organize requirements by SDLC phase.

AI Code Generator

Generate code from your natural language ideas.

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



Chat with SmartSDLC Assistant

Ask something SDLC-related:

What are the different phases of SDLC?

You: What are the different phases of SDLC?

Bot: - Requirements gathering

- Analysis and design
- Coding and implementation
- Testing
- Deployment
- Maintenance

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

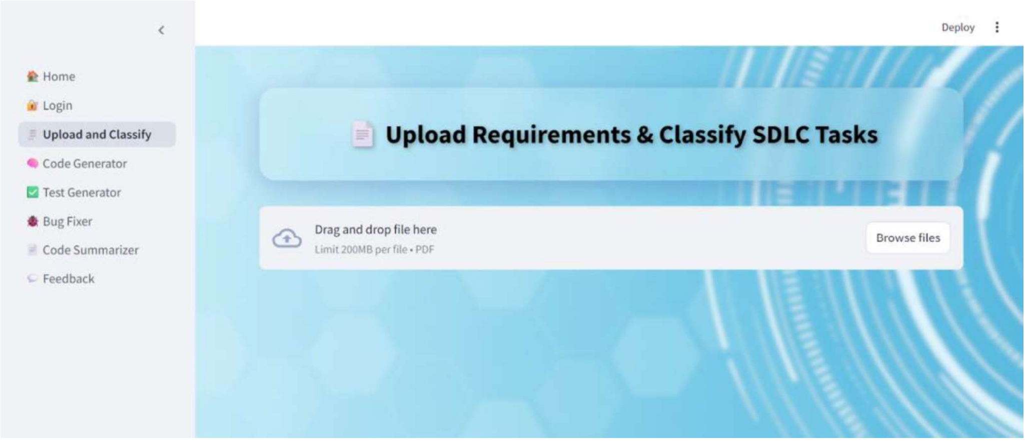
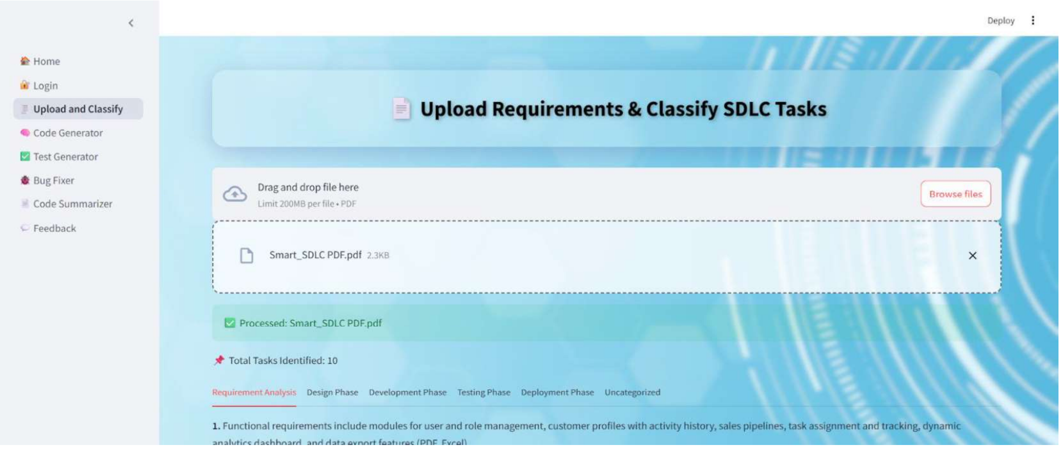


Figure 5: Input: "Upload and Classify – Effortlessly convert raw PDF requirements into structured SDLC tasks using AI."

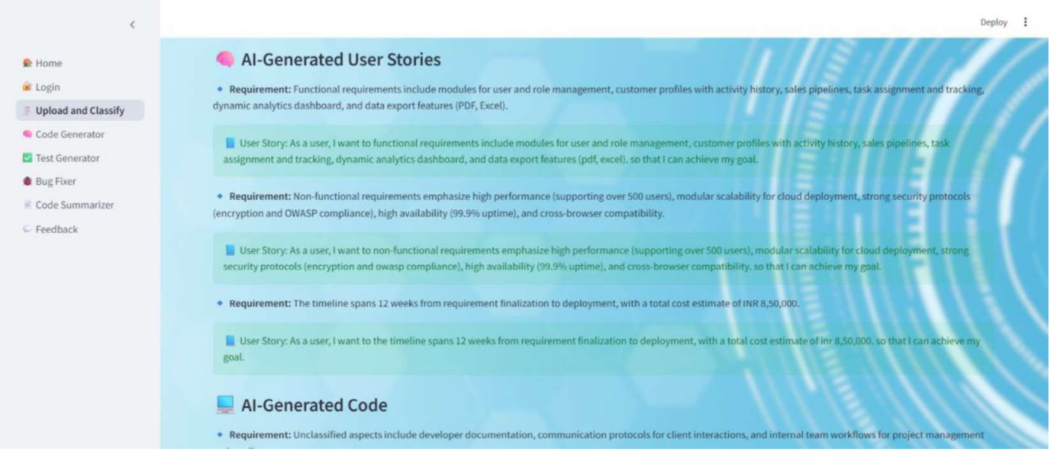


☑ Processed: Smart_SDLC PDF.pdf

📌 Total Tasks Identified: 10

1. Functional requirements include modules for user and role management, customer profiles with activity history, sales pipelines, task assignment and tracking, dynamic analytics dashboard, and data export features (PDF, Excel).

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



AI-Generated User Stories

- **Requirement:** Functional requirements include modules for user and role management, customer profiles with activity history, sales pipelines, task assignment and tracking, dynamic analytics dashboard, and data export features (PDF, Excel).

User Story: As a user, I want to functional requirements include modules for user and role management, customer profiles with activity history, sales pipelines, task assignment and tracking, dynamic analytics dashboard, and data export features (pdf, excel). so that I can achieve my goal.

- **Requirement:** Non-functional requirements emphasize high performance (supporting over 500 users), modular scalability for cloud deployment, strong security protocols (encryption and OWASP compliance), high availability (99.9% uptime), and cross-browser compatibility.

User Story: As a user, I want to non-functional requirements emphasize high performance (supporting over 500 users), modular scalability for cloud deployment, strong security protocols (encryption and owasp compliance), high availability (99.9% uptime), and cross-browser compatibility. so that I can achieve my goal.

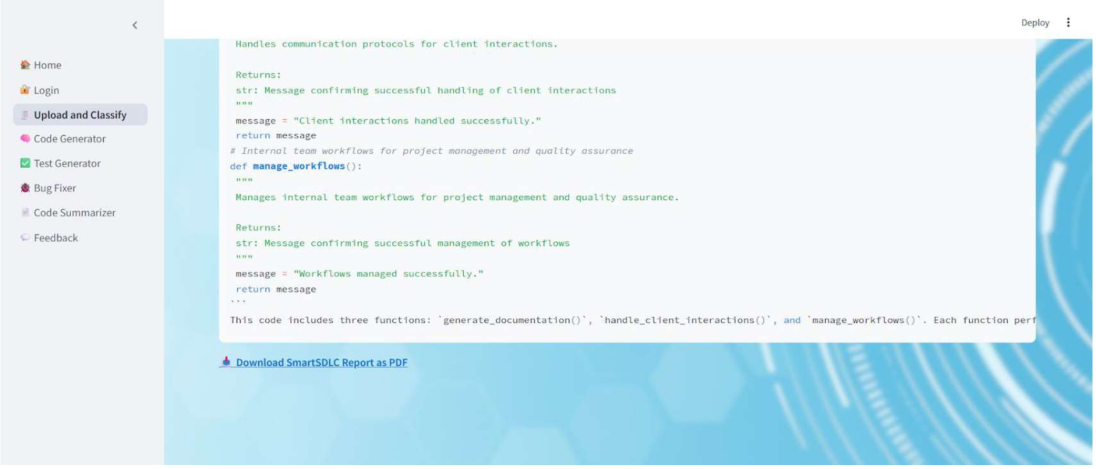
- **Requirement:** The timeline spans 12 weeks from requirement finalization to deployment, with a total cost estimate of INR 8,50,000.

User Story: As a user, I want to the timeline spans 12 weeks from requirement finalization to deployment, with a total cost estimate of inr 8,50,000. so that I can achieve my goal.

AI-Generated Code

- **Requirement:** Unclassified aspects include developer documentation, communication protocols for client interactions, and internal team workflows for project management

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

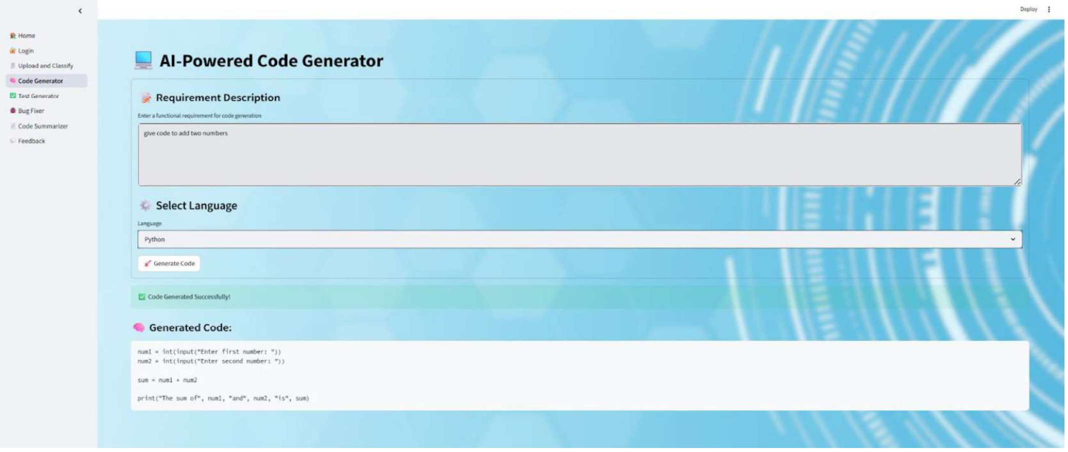


```
 Handles communication protocols for client interactions.

 Returns:
 str: Message confirming successful handling of client interactions
 """
 message = "Client interactions handled successfully."
 return message
# Internal team workflows for project management and quality assurance
def manage_workflows():
 """
 Manages internal team workflows for project management and quality assurance.

 Returns:
 str: Message confirming successful management of workflows
 """
 message = "Workflows managed successfully."
 return message
```

This code includes three functions: `generate_documentation()`, `handle_client_interactions()`, and `manage_workflows()`. Each function perf

Download SmartSDLC Report as PDF

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



AI-Powered Code Generator

Requirement Description

Enter a functional requirement for code generation

give code to add two numbers

Select Language

Language

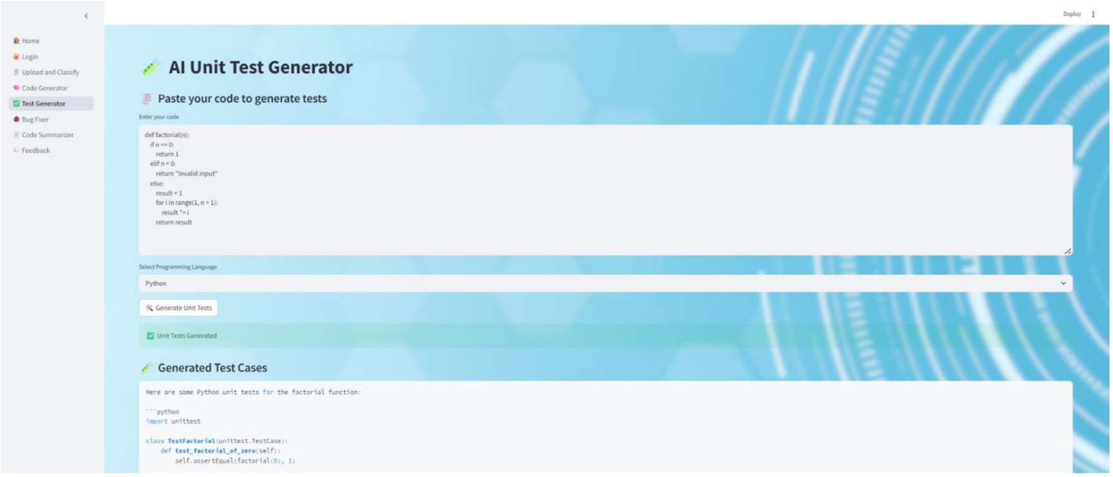
Python

☑ Code Generated Successfully!

Generated Code:

```
num1 = int(input("Enter first number: "))
num2 = int(input("Enter second number: "))

sum = num1 + num2

print("The sum of", num1, "and", num2, "is", sum)
```

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



AI Unit Test Generator

Paste your code to generate tests

Enter your code

```
def factorial(n):
  if n == 0:
    return 1
  elif n < 0:
    return "Invalid input"
  else:
    result = 1
    for i in range(1, n + 1):
      result *= i
    return result
```

Select Programming Language

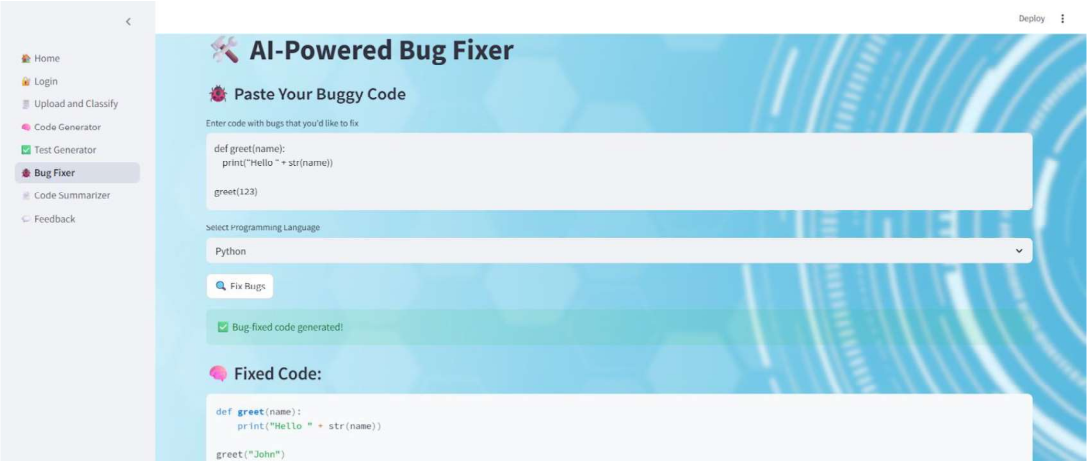
Python

☑ Unit Tests Generated

Generated Test Cases

~~~
Here are some Python unit tests for the factorial function:

```python
import unittest

class TestFactorial(unittest.TestCase):
    def test_factorial_of_zero(self):
        self.assertEqual(factorial(0), 1)
~~~

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



AI-Powered Bug Fixer

Paste Your Buggy Code

Enter code with bugs that you'd like to fix

```
def greet(name):
  print("Hello " + str(name))

greet(123)
```

Select Programming Language

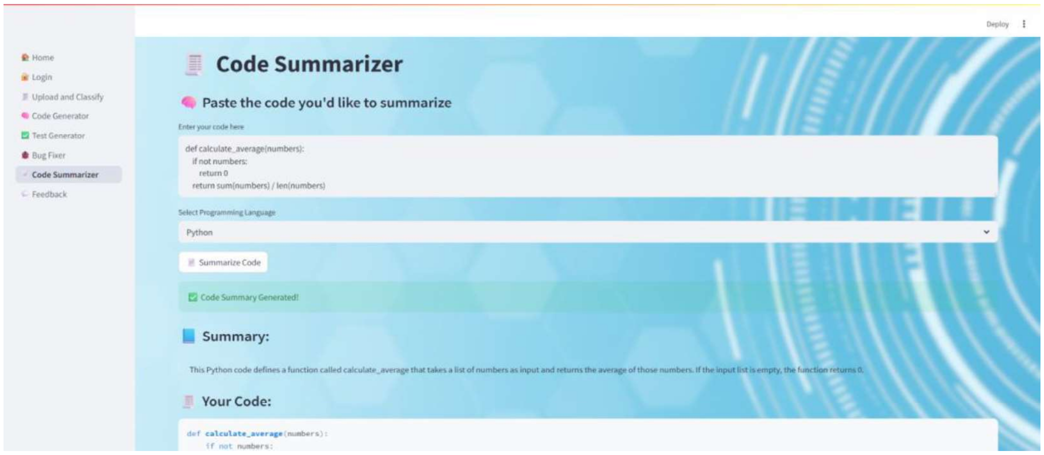
Python

☑ Bug-fixed code generated!

Fixed Code:

```
def greet(name):
    print("Hello " + str(name))

greet("John")
```

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



Code Summarizer

Paste the code you'd like to summarize

Enter your code here

```
def calculate_average(numbers):
  if not numbers:
    return 0
  return sum(numbers) / len(numbers)
```

Select Programming Language

Python

Code Summary Generated!

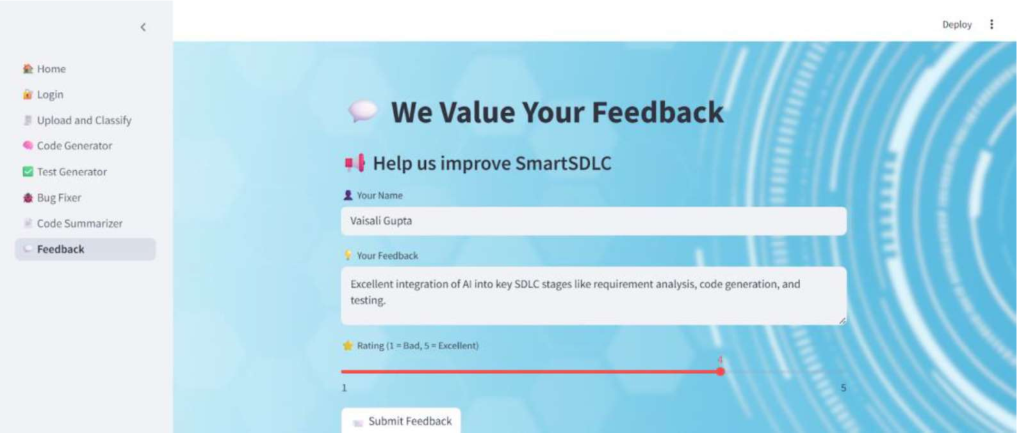
Summary:

This Python code defines a function called calculate_average that takes a list of numbers as input and returns the average of those numbers. If the input list is empty, the function returns 0.

Your Code:

```
def calculate_average(numbers):
    if not numbers:
```

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



We Value Your Feedback

Help us improve SmartSDLC

Your Name

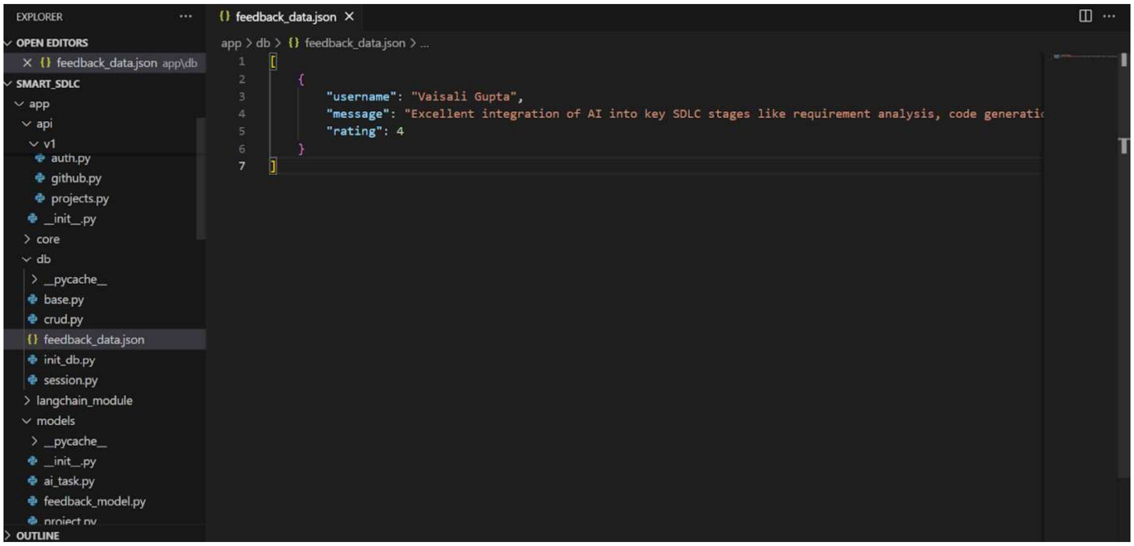
Vaisali Gupta

Your Feedback

Excellent integration of AI into key SDLC stages like requirement analysis, code generation, and testing.

Rating (1 = Bad, 5 = Excellent)

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



```json
[
    {
        "username": "Vaisali Gupta",
        "message": "Excellent integration of AI into key SDLC stages like requirement analysis, code generati
        "rating": 4
    }
]
```

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