SmartSDLC–AI-Enhanced Software Development Lifecycle

Project Description:

SmartSDLC is a full-stack, AI-powered platform that redefines the traditional Software Development Lifecycle (SDLC) by automating key stages using advanced Natural Language Processing (NLP) and Generative AI technologies.

It is not just a tool — it's an intelligent ecosystem that allows teams to convert unstructured requirements into code, test cases, and documentation instantly, thereby minimizing manual intervention, enhancing accuracy, and accelerating the delivery pipeline.

Scenarios:

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.

Scenario 4: Test Case Generator

Test Case Generator focuses on quality assurance, where users provide functional code or a requirement, and the AI generates suitable test cases. These are structured using familiar testing frameworks like unittest or pytest, enabling seamless validation and automation of software testing. This module eliminates the need for manual test writing, ensuring consistency and completeness in test coverage.

Scenario 5: Code Summarizer

Code Summarizer, the platform aids documentation by accepting any source code snippet or module and generating a human-readable explanation. Watsonx analyzes the logic and purpose of the code, then summarizes its function and use cases. This feature is especially helpful for onboarding new developers or for maintaining long-term documentation, making complex codebases easier to understand.

Scenario 6: Floating AI Chatbot Assistant

Floating AI Chatbot Assistant provides real-time, conversational support throughout the application. Integrated using LangChain, the chatbot responds intelligently to user queries about the SDLC, such as "How do I write a unit test?" or "What is requirement analysis?". The backend handles prompt routing based on keyword detection, and the frontend presents the response in a styled chat interface, offering an intuitive and interactive help system.

Collectively, these scenarios demonstrate how SmartSDLC intelligently automates core development tasks, enhances team collaboration, and empowers both technical and non-technical users to efficiently engage with the software development process.

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

Activity 1: Model Selection and Architecture

- **Activity 1.1:** Research and select the appropriate generative AI model
- **Activity 1.2:** Define the architecture of the application.
- **Activity 1.2:** Set up the development environment.

Activity 2: Core Functionalities Development

- **Activity 2.1:** Develop the core functionalities.
- **Activity 2.2:** Implement the FastAPI backend to manage routing and user input processing, ensuring smooth API interactions.

Activity 3: main.py Development

- **Activity 3.1:** Writing the Main Application Logic in main.py
- **Activity 3.2:** Create prompting strategies for the IBM Granite model to generate high-quality educational content across multiple languages.

Activity 4: Frontend Development

- **Activity 4.1:** Designing and Developing the User Interface
- **Activity 4.2**: Creating Dynamic Interaction with Backend

Activity 5: Deployment

- **Activity 5.1:** Preparing the Application for Local Deployment
- **Activity 5.2 Testing and Verifying Local Deployment**

Milestone 1: Model Selection and Architecture

The content outlines a comprehensive three-part activity for developing a SmartSDLC platform powered by IBM's Watsonx generative AI models. It begins with a research phase to select appropriate AI models for automating various software development lifecycle tasks, focusing on evaluating IBM's Granite models for both natural language processing and code generation capabilities. The second activity involves defining the system architecture, detailing the interaction between the Streamlit frontend and FastAPI backend, and establishing the AI integration workflow through Watsonx APIs and LangChain orchestration. The final activity provides step-by-step instructions for setting up the development environment, including installing dependencies, creating virtual environments, securing API keys, and organizing the project structure for efficient development across frontend and backend components.

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.

Activity1.2: 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:

- PDF processing and classification logic (using PyMuPDF).
- Prompt generation and API communication with Watsonx.
- LangChain agent for orchestration.
- Auth module for user sessions and hashed logins.

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.

Activity1.3: 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.

Milestone 2: Core Functionalities Development

Activity 2.1: Develop 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.

Activity 2.2: Implement the Fast API 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.

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.

Milestone 3: main.py Development

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:

1. /ai: Handles AI-based endpoints like code generation, test creation, bug fixing.
2. /auth: Manages login, registration, and user validation.
3. /chat: Powers the floating chatbot via LangChain.
4. /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).

Milestone 4: Frontend Development

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

1. 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.

2. Connect each page to corresponding FastAPI endpoints via api_client.

Feedback.py: Collect user feedback.

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

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:8501
- 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.