SmartSDLC – AI-Enhanced Software Development Lifecycle

Project Documentation

1. Introduction

Project Title: SmartSDLC – AI-Enhanced Software Development Lifecycle

Team Members:

- I Srinithi
- V srimathi
- A Sheerin
- D Pavithra
- P Madhan babu

2. Project Overview

Purpose

The purpose of the SmartSDLC project is to create an intelligent, automated, and streamlined platform for managing the Software Development Lifecycle (SDLC). In traditional software development, multiple phases such as requirement analysis, design, coding, testing, debugging, and documentation involve significant manual effort. These steps are often time-consuming, repetitive, and error-prone, leading to delays, miscommunication, and inconsistencies in software quality. SmartSDLC addresses these challenges by integrating Artificial Intelligence (AI) and automation technologies into the development process. Its primary aim is to reduce manual intervention in repetitive tasks so that developers and project teams can focus on innovation, decision-making, and creative problem-solving rather than routine work. By leveraging IBM Watsonx for natural language processing, LangChain for orchestration, FastAPI for scalable backend services, and Streamlit for an interactive frontend interface, SmartSDLC ensures a smooth, automated workflow across all SDLC stages. The platform not only accelerates requirement gathering and classification but also generates structured user stories, produces production-ready code, fixes bugs instantly, and provides automated documentation.

Features

1. Requirement Upload and Classification

Users can upload PDFs with unstructured requirements. PyMuPDF extracts text, and Watsonx classifies it into SDLC phases like Requirements, Design, or Testing. The results are displayed as structured user stories for clarity and traceability.

2. AI Code Generator

Developers can input natural language prompts or user stories to generate production-ready code. Watsonx ensures accurate and efficient output, which is displayed in a syntax-highlighted format for immediate use.

3. Bug Fixer

Buggy code in Python or JavaScript can be submitted for AI analysis. The system detects syntax and logic errors, returning corrected code with a side-by-side comparison of original and fixed versions.

4. AI-Powered Chatbot Assistance

A built-in chatbot provides real-time support across SDLC stages. It answers questions, clarifies requirements, and guides users, making the platform interactive and beginner-friendly.

5. Automated Documentation

SmartSDLC auto-generates documentation from requirements, code, and bug fixes. This ensures consistent, up-to-date records that save time and improve collaboration among team members.

6. Policy Summarization

Long technical or compliance documents are summarized by AI. This feature delivers concise versions of policies, helping teams quickly understand key points without reading every detail.

7. Modular and Scalable Architecture

The system is modular, allowing each feature to work independently yet integrate seamlessly. Its design supports easy scaling and the addition of new functionalities.

8. Secure Authentication and Access Control

Access is protected through token-based authentication. Role-based access can also be added to ensure data confidentiality and controlled permissions for different users.

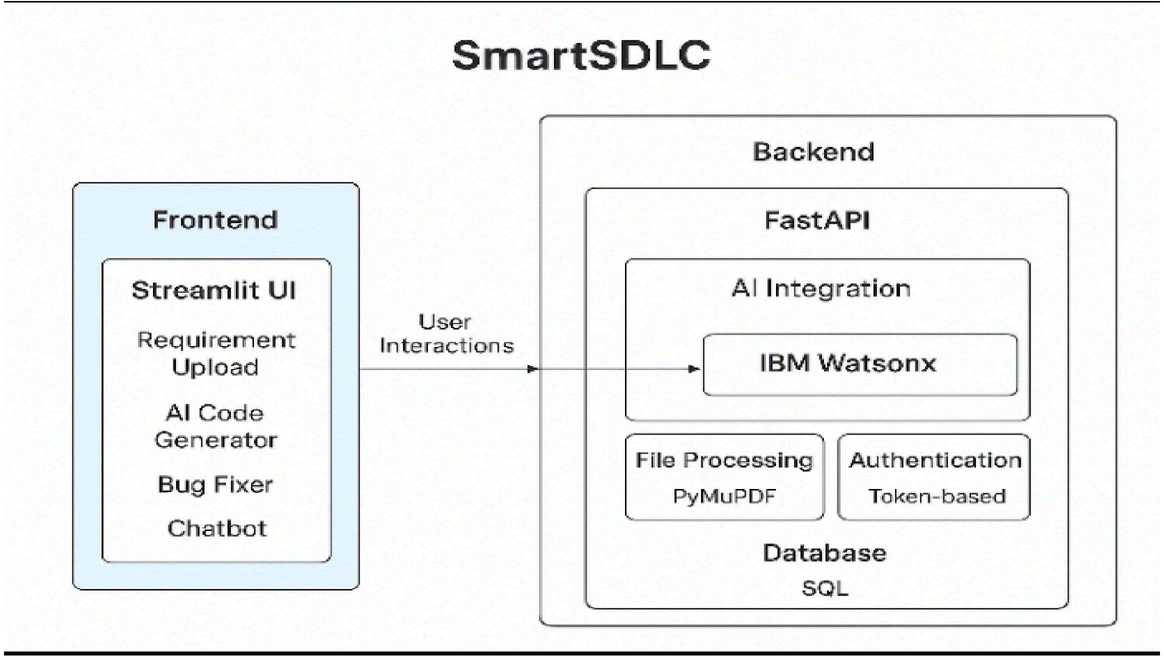
9. Interactive User Interface

The Streamlit dashboard offers an intuitive, organized interface. Users can upload files, generate code, and fix bugs with ease, supported by syntax highlighting and navigation tabs.

10. Seamless Deployment and Documentation

The backend uses FastAPI with Swagger UI for API documentation. Deployment is simplified with Uvicorn and Streamlit, and can be scaled further using Docker containers.

3. Architecture



1. Frontend (Streamlit UI): Provides an intuitive web dashboard where users upload requirements, view generated code, and interact with the chatbot. It ensures smooth navigation with tabs and syntax highlighting.

2. Backend (FastAPI Services): Handles request processing, file parsing, AI model interaction, and data exchange. FastAPI offers speed, scalability, and built-in API documentation via Swagger UI.

3. AI Integration (Watsonx): Core intelligence of the system that powers classification, code generation, bug fixing, and summarization. It ensures accuracy, adaptability, and consistency across tasks.

4. File Processing (PyMuPDF): Extracts and cleans text from uploaded PDF requirement documents. This forms the input for AI models to classify and convert into structured user stories.

5. Authentication Layer: Manages secure access using token-based authentication. Provides a foundation for role-based access control and data protection in multi-user scenarios.

6. Database Layer (Optional): Can be integrated for storing requirements, generated code, bug fixes, and chat history. Supports traceability and collaboration in enterprise settings.

7. Deployment (Uvicorn + Docker): Uvicorn runs FastAPI for efficient request handling. Docker ensures portable, scalable deployment across different environments and teams.

8. Documentation (Swagger + Auto-Docs): APIs are auto-documented with Swagger UI. SmartSDLC also generates user-friendly project documentation to maintain transparency and reduce manual effort.

4. Setup Instructions

Prerequisites

- Python 3.10 or above
- FastAPI
- Streamlit
- Uvicorn
- PyMuPDF (fitz)
- IBM Watsonx AI SDK
- LangChain
- Git & GitHub

Installation Process

1. Clone the repository:

```
git clone https://github.com/<username>/SmartSDLC.git
```

```
cd SmartSDLC
```

2. Create a virtual environment:

```
python -m venv venv
source venv/bin/activate (Linux/Mac)
venv\Scripts\activate (Windows)
```

3. Install dependencies:

```
pip install -r requirements.txt
```

4. Configure IBM Watsonx credentials in environment variables or .env file.

5. Folder Structure

1. backend/ – Contains all backend logic built with FastAPI. This includes API endpoints for requirement classification, code generation, bug fixing, and communication with the AI model. It also manages authentication and API documentation.

2. frontend/ – Includes the Streamlit-based user interface. This folder holds files for the dashboard, forms for requirement uploads, code generation panels, and chatbot integration.

3. docs/ – Stores documentation files, reports, and any reference material that explains the system's design and usage. This section is useful for both developers and end-users.

4. samples/ – Contains sample PDF requirement documents, test cases, and code snippets used for demonstrating the system's functionality.

5. requirements.txt – A text file listing all the Python libraries and dependencies required to run the project.

6. README.md – The project's main overview file that gives a quick introduction, setup guide, and usage instructions.

6. Run the Application

1. Start Backend

Launch the FastAPI backend service. API endpoints will be available, and Swagger UI can be used for testing.

2. Start Frontend

Launch the Streamlit dashboard.Open the local link in a browser to access the interface.

3. Use SmartSDLC

Upload PDF requirements. Generate code from prompts. Fix bugs and use the AI chatbot for guidance.

7. API Documentation

The APIs are exposed using FastAPI with interactive documentation via Swagger UI.yEndpoints Overview:

- POST /requirements/upload → Upload PDF and extract classified requirements

- POST /code/generate → Generate code from natural language or user stories

- POST /bugfix/analyze → Analyze and return corrected code

- GET /health → Check service health

8. Authentication

SmartSDLC uses token-based authentication to secure access.

Users must provide a valid API key or token to interact with backend services.

Role-based access can be added to differentiate permissions for developers, testers, or project managers.

Authentication ensures that sensitive data, such as uploaded requirements and generated code, remains secure.

9. User Interface

The frontend is built with Streamlit, providing an intuitive and interactive dashboard.

Users can upload PDF requirements, view structured outputs, and generate code.

The dashboard also allows users to fix buggy code and interact with the AI-powered chatbot.

Navigation is simple, with organized tabs and syntax-highlighted code display for clarity.

10. Testing

Testing ensures the reliability and accuracy of the SmartSDLC platform.

Unit Tests verify individual components like requirement classification, code generation, and bug fixing.

Integration Tests ensure smooth communication between frontend, backend, and AI services.

Performance Tests measure speed and responsiveness, especially for large PDFs or code snippets.

Testing can be performed using frameworks like Pytest, and results help improve platform stability.

11.Coding

```
# -*- coding: utf-8 -*-

"""smartsdlc

 AIAutomatically generated by Colab.

Original file is located at

   https://colab.research.google.com/drive/1CBsApoiBx6iDavgiGPSSb25hVWEKmHYT

"""

import gradio as gr

import torch
```

```
from transformers import AutoTokenizer, AutoModelForCausalLM
import io
# Load model and tokenizer
model_name = "ibm-granite/granite-3.2-2b-instruct"
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModelForCausalLM.from_pretrained(
    model_name,
    torch_dtype=torch.float16 if torch.cuda.is_available() else torch.float32,
    device_map="auto" if torch.cuda.is_available() else None
)
if tokenizer.pad_token is None:
    tokenizer.pad_token = tokenizer.eos_token

def generate_response(prompt, max_length=1024):
    inputs = tokenizer(prompt, return_tensors="pt", truncation=True, max_length=512)
    if torch.cuda.is_available():
        inputs = {k: v.to(model.device) for k, v in inputs.items()}
    with torch.no_grad():
        outputs = model.generate(
            **inputs,
            max_length=max_length,
            temperature=0.7,
            do_sample=True,
            pad_token_id=tokenizer.eos_token_id
        )
    response = tokenizer.decode(outputs[0], skip_special_tokens=True)

    response = response.replace(prompt, "").strip()

    return response

def extract_text_from_pdf(pdf_file):

    if pdf_file is None:

        return ""

    try:

        pdf_reader = PdfReader(pdf_file)

        text = ""

        for page in pdf_reader.pages:

            text += page.extract_text() + "\n"

        return text

    except Exception as e:

        return f"Error reading PDF: {str(e)}"

def requirement_analysis(pdf_file, prompt_text):

    # Get text from PDF or prompt

    if pdf_file is not None:

        content = extract_text_from_pdf(pdf_file)

        analysis_prompt = f"Analyze the following document and extract key software 
requirements. Organize them into functional requirements, non-functional requirements, 
and technical specifications:\n\n{content}"

    else:

        analysis_prompt = f"Analyze the following requirements and organize them into 
functional requirements, non-functional requirements, and technical 
specifications:\n\n{prompt_text}"

    return generate_response(analysis_prompt, max_length=1200)
```

```
def code_generation(prompt, language):
    code_prompt = f"Generate {language} code for the following
requirement:\n\n{prompt}\n\nCode:"
    return generate_response(code_prompt, max_length=1200)
# Create Gradio interface
with gr.Blocks() as app:
    gr.Markdown("# AI Code Analysis & Generator")
    with gr.Tabs():
        with gr.TabItem("Code Analysis"):
            with gr.Row():
                with gr.Column():
                    pdf_upload = gr.File(label="Upload PDF", file_types=[".pdf"])
                    prompt_input = gr.Textbox(
                        label="Or write requirements here",
                        placeholder="Describe your software requirements...",
                        lines=5
                    )
                    analyze_btn = gr.Button("Analyze")
                with gr.Column():
                    analysis_output = gr.Textbox(label="Requirements Analysis", lines=20)
            analyze_btn.click(requirement_analysis, inputs=[pdf_upload, prompt_input],
outputs=analysis_output)
        with gr.TabItem("Code Generation"):
            with gr.Row():
                with gr.Column():
```

```
                code_prompt = gr.Textbox(
                    label="Code Requirements",
                    placeholder="Describe what code you want to generate...",
                    lines=5
                )
                language_dropdown = gr.Dropdown(
                    choices=["Python", "JavaScript", "Java", "C++", "C#", "PHP", "Go", "Rust"],
                    label="Programming Language",
                    value="Python"
                )
                generate_btn = gr.Button("Generate Code")

            with gr.Column():
                code_output = gr.Textbox(label="Generated Code", lines=20)
        generate_btn.click(code_generation, inputs=[code_prompt, language_dropdown],
outputs=code_output)
app.launch(share=True)
```

12.known issues

- IBM Watsonx API latency may cause delays.
- Limited to Python and JavaScript in bug fixing.
- Streamlit responsiveness may vary with large PDFs.
- No offline mode.

13. Future Enhancements

- Support for more programming languages.
- Enhanced bug detection with detailed explanations.
- Project management features like task tracking.
- Multi-user support with role-based access.

- Cloud deployment with Docker/Kubernetes.
- Advanced analytics and reporting dashboards.