Smart SDLC – AI-Enhanced Software Development Lifecycle

Generative AI with IBM - Project Report

Submitted by: **Team Leader -** Subbashini G

Team Members – Swathi S, Tamil selvi S, Swetha S, Swetha R

TABLE OF CONTENTS

1.ABSTRACT

The Smart SDLC Platform is a smart tool designed to help automate different stages of software development. It uses advanced AI models like Granite-20B-Code-Instruct and LangChain to understand user inputs such as code, questions, prompts, and PDF files.

Based on these inputs, the platform can generate software requirements, code, test cases, and summaries.

The platform is built using FastAPI and Streamlit, making it fast and easy to use. It also includes a floating chatbot assistant that helps users by answering questions and giving coding support. Tools like PyMuPDF and Uvicorn, along with AI services from IBM Watsonx, are used to make the platform work efficiently.

Overall, SmartSDLC helps developers save time, reduce manual work, and speed up the software development process by turning natural language inputs into useful software components.

2. OBJECTIVES

- ✓ Automate and streamline the Software Development Life Cycle (SDLC) using AI-driven tools.
- ✓ Leverage advanced language models like Granite-20B-Code-Instruct via IBM Watsonx for intelligent code and text processing.
- ✓ Use LangChain to integrate LLM capabilities into the development pipeline.
- ✓ Accept diverse user inputs.
- ✓ Convert inputs into structured outputs
- ✓ Build with FastAPI and Streamlit for a responsive backend and easy-to-use frontend.
- ✓ Include a floating chatbot assistant.

3.SYSTEM REQUIREMENTS

Software Requirements:

- Programming Languages & Tools:
- Python 3.10
- Fast API – For building APIs
- Streamlit – For interactive front-end interface
- Uvicorn – ASGI server for FastAPI
- PyMuPDF (fitz) – For reading PDF documents
- AI & NLP Tools:
- IBM Watsonx AI & Granite Models – For AI model inference
- LangChain – To integrate LLMs and tools
- Git & GitHub –

Hardware Requirements:

- A system capable of running Python 3.10 (Windows/Linux/Mac)
- At least 8 GB RAM (recommended for AI processing)
- Stable internet connection (for API calls and model access)
- Optional: GPU (if running models locally or handling large documents**).**

4.PROJECT WORKFLOW

Environment Setup:

Python 3.10 with FastAPI, Streamlit, LangChain, PyMuPDF, and IBM Granite AI via Watsonx.

Input Module:

Handles PDFs, code snippets, and natural language prompts.

Requirement & User Story Extraction:

AI extracts requirements from PDFs and creates user stories.

Code Generation & Testing:

Converts prompts to code, generates unit tests, and fixes bugs automatically.

Code Summarization:

Provides simple explanations of code functionality.

Report & Interface:

Compiles outputs into downloadable reports; user-friendly web interface with chatbot support.

Feedback & Safety:

Collects user feedback; reminds users to verify AI-generated outputs.

5.PROJECT FILES

- env – Stores API keys and configurations
- App.py – Main application file with Gradio blocks
- requirements.txt – Lists required dependencies

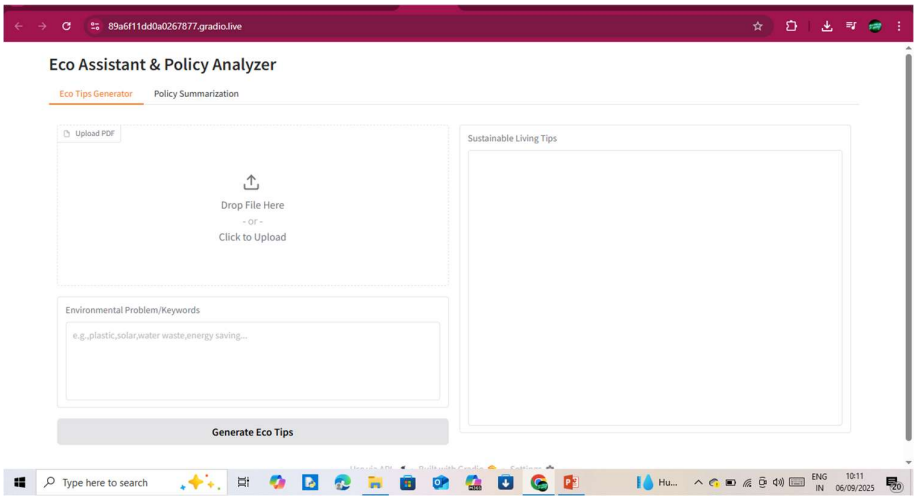
6. APPENDIX: SOURCE CODE

```
import gradio as gr
import torch
from transformers import AutoTokenizer, AutoModelForCausalLM
import PyPDF2
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 = PyPDF2.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)
```

7. OUTPUT SCREENSHOT



Eco Assistant & Policy Analyzer is a user-friendly web tool that helps promote sustainable living and environmental awareness. It features:

- **Eco Tips Generator** – Users can upload documents or enter keywords to get practical tips for eco-friendly living.
- **Policy Summarization** – Simplifies complex environmental policies into easy-to-understand summaries.

Built using **Gradio**, the tool makes environmental guidance and policy insights accessible to everyone.

8.CONCLUSION

SmartSDLC is an AI-powered platform designed to automate and enhance the entire Software Development Lifecycle (SDLC) — from requirements analysis and code generation to testing, bug fixing, and documentation. By leveraging cutting-edge technologies such as IBM Watsonx, FastAPI, LangChain, and Streamlit, it streamlines development processes, minimizes manual errors, and accelerates software delivery.

Featuring a modular architecture and an intuitive user interface, SmartSDLC empowers both technical and non-technical users to manage SDLC tasks effectively. Its key functionalities include AI-driven requirement classification, automated user story creation, code generation from natural language input, test case generation, intelligent bug resolution, and built-in chat assistance.

SmartSDLC significantly improves development productivity and quality, while setting the stage for future enhancements like CI/CD integration, team collaboration tools, version control, and cloud deployment—making it a robust solution for modern, agile development environments.