SmartSDLC AI Enhanced Software Development Lifecycle

1. Introduction

Project Title: SmartSDLC AI Enhanced Software Development Lifecycle
Team Members:

- S. Sameera Jasmine
- D. Rubika
- S. Salini
- M. Rukshana Fathima

2. Project Overview

Purpose:
The purpose of a Sustainable Smart City Assistant is to empower cities and their residents to thrive in a more eco-conscious and connected urban environment. By leveraging AI and real-time data, the assistant helps optimize essential resources like energy, water, and waste, while also guiding sustainable behaviors among citizens through personalized tips and services. For city officials, it serves as a decision-making partner—offering clear insights, forecasting tools, and summarizations of complex policies to support strategic planning.

Features:

- Conversational Interface - Natural language interaction, allows plain language queries.
- Policy Summarization - Converts lengthy documents into actionable summaries.
- Resource Forecasting - Estimates future usage with predictive analytics.
- Eco-Tip Generator - Personalized sustainability advice.
- Citizen Feedback Loop - Collects and analyzes public input.
- KPI Forecasting - Strategic planning support.
- Anomaly Detection - Early warning system for unusual patterns.
- Multimodal Input Support - Handles text, PDFs, CSVs.
- Streamlit or Gradio UI - Provides intuitive dashboards.

3. Architecture

Frontend (Streamlit): Interactive web UI with dashboards, file uploads, chat, feedback, reports.
Backend (FastAPI): REST framework for processing, chat, eco tips, reports, embeddings.
LLM Integration: IBM Watsonx Granite for summaries, tips, and reports.
Vector Search: Pinecone for semantic search on embedded docs.
ML Modules: Forecasting & anomaly detection with Scikit-learn, pandas, matplotlib.

4. Setup Instructions

Prerequisites:

- Python 3.9+
- pip & virtual environment tools
- API keys for IBM Watsonx & Pinecone
- Internet access

Installation:

- Clone repository
- Install requirements.txt
- Create .env and configure
- Run FastAPI server
- Launch Streamlit frontend
- Upload data & interact

5. Folder Structure

- app/ – Backend logic
- app/api/ – API routes
- ui/ – Frontend Streamlit pages
- smart_dashboard.py – Launches dashboard
- granite_llm.py – Handles IBM Watsonx communication
- document_embedder.py – Embeds & stores docs
- kpi_file_forecaster.py – Forecasting trends
- anomaly_file_checker.py – Detect anomalies
- report_generator.py – Generates sustainability reports

6. Running the Application

1. Launch FastAPI server.
2. Run Streamlit dashboard.
3. Navigate via sidebar.
4. Upload docs/CSVs & interact.
5. View reports, summaries, predictions.

7. API Documentation

- POST /chat/ask
- POST /upload-doc
- GET /search-docs
- GET /get-eco-tips
- POST /submit-feedback

8. Authentication

- Token-based (JWT, API keys)
- OAuth2 (IBM Cloud)
- Role-based access
- Planned: sessions & history tracking

9. User Interface

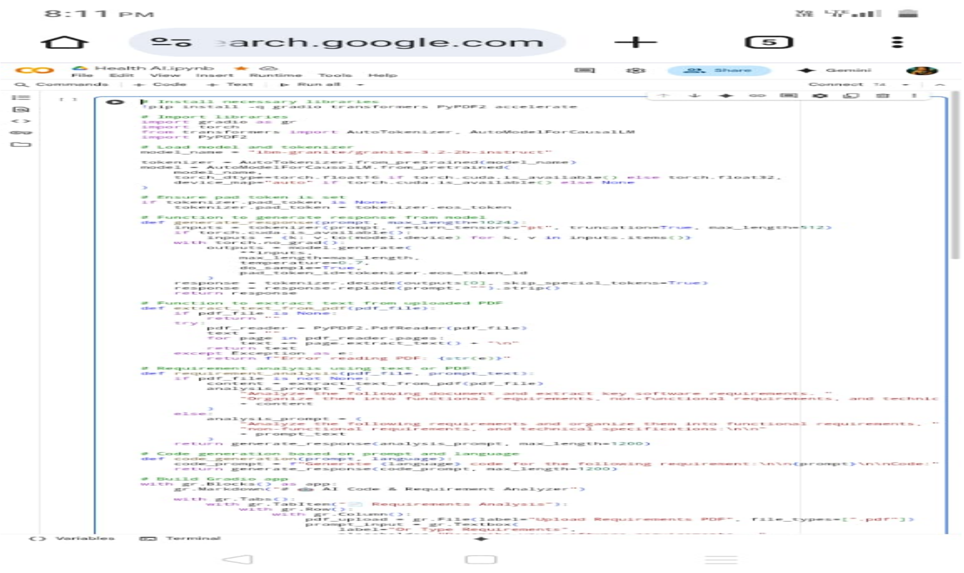
Minimalist & functional, includes:

- Sidebar navigation
- KPI visualizations
- Tabbed layouts (chat, eco tips, forecasting)
- Real-time forms
- PDF report download

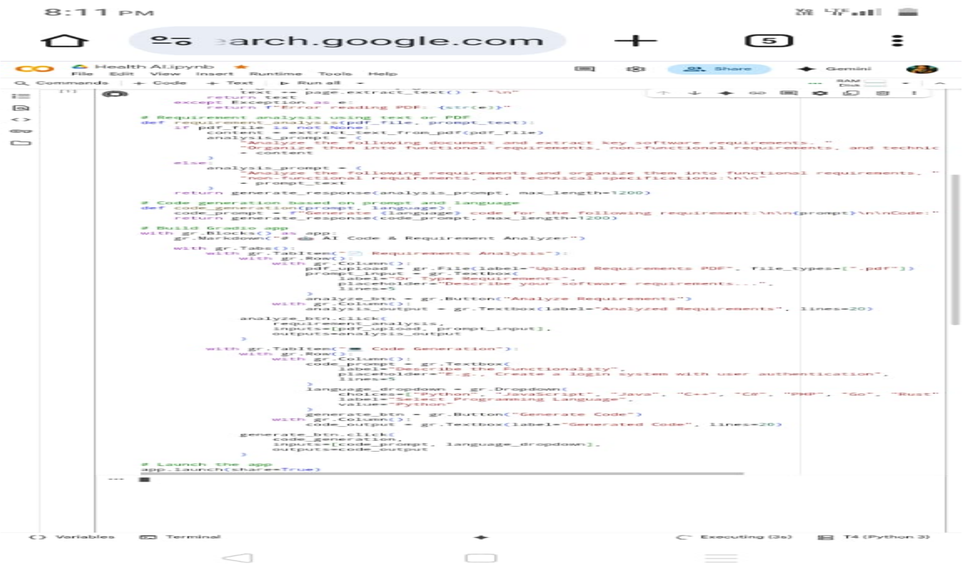
10. Testing

- Unit testing (prompt functions)
- API testing (Swagger, Postman)
- Manual testing (uploads, chat, outputs)
- Edge cases (malformed inputs, large files, invalid keys)

11. Screenshots



```
# Install necessary libraries
!pip install -q gradio transformers PyPDF2 accelerate

# Import libraries
import gradio as gr
import torch
from transformers import AutoTokenizer, AutoModelForCausalLM
import PyPDF2

# 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
)

# Ensure pad token is set
if tokenizer.pad_token is None:
    tokenizer.pad_token = tokenizer.eos_token

# Function to generate response from model
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

# Function to extract text from uploaded PDF
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)}"

# Requirement analysis using text or PDF
def requirement_analysis(pdf_file, prompt_text):
    if pdf_file is not None:
        content = extract_text_from_pdf(pdf_file)
        analysis_prompt = (
            "Analyze the following document and extract key software requirements. "
            "Organize them into functional requirements, non-functional requirements, and technic
            + content
        )
    else:
        analysis_prompt = (
            "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)

# Code generation based on prompt and language
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)

# Build Gradio app
with gr.Blocks() as app:
    gr.Markdown("# 🤖 AI Code & Requirement Analyzer")

    with gr.Tabs():
        with gr.TabItem("📄 Requirements Analysis"):
            with gr.Row():
                with gr.Column():
                    pdf_upload = gr.File(label="Upload Requirements PDF", file_types=[".pdf"])
                    prompt_input = gr.Textbox(
                        label="Or Type Requirements",
```



```
            text += page.extract_text() + "\n"
        return text
    except Exception as e:
        return f"Error reading PDF: {str(e)}"

# Requirement analysis using text or PDF
def requirement_analysis(pdf_file, prompt_text):
    if pdf_file is not None:
        content = extract_text_from_pdf(pdf_file)
        analysis_prompt = (
            "Analyze the following document and extract key software requirements. "
            "Organize them into functional requirements, non-functional requirements, and technic
            + content
        )
    else:
        analysis_prompt = (
            "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)

# Code generation based on prompt and language
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)

# Build Gradio app
with gr.Blocks() as app:
    gr.Markdown("# 🤖 AI Code & Requirement Analyzer")

    with gr.Tabs():
        with gr.TabItem("📄 Requirements Analysis"):
            with gr.Row():
                with gr.Column():
                    pdf_upload = gr.File(label="Upload Requirements PDF", file_types=[".pdf"])
                    prompt_input = gr.Textbox(
                        label="Or Type Requirements",
                        placeholder="Describe your software requirements...",
                        lines=5
                    )
                    analyze_btn = gr.Button("Analyze Requirements")
                with gr.Column():
                    analysis_output = gr.Textbox(label="Analyzed Requirements", 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="Describe the Functionality",
                        placeholder="E.g., Create a login system with user authentication",
                        lines=5
                    )
                    language_dropdown = gr.Dropdown(
                        choices=["Python", "JavaScript", "Java", "C++", "C#", "PHP", "Go", "Rust"
                        label="Select 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
            )

# Launch the app
app.launch(share=True)
```

12. Known Issues

Currently none reported. (To be updated after deployment)

13. Future Enhancements

- Add advanced analytics dashboards
- Expand LLM integration
- Support multilingual input
- Improve authentication & role-based access