**SMART SDLC – AI Enhanced Software Development Lifecycle**

**Table of Contents**

1. Abstract
2. Objectives
3. System Requirements
4. Implementation
5. Project Files and Modules
6. Appendix – Source Code
7. Output Screenshots
8. Conclusion
9. **Abstract**

The SMART SDLC (Software Development Life Cycle) project is an AI-powered application designed to automate and enhance various phases of software development. It integrates generative AI models with modern frameworks such as FastAPI, Streamlit, LangChain, and IBM Watsonx, providing developers with tools for requirement classification, user story generation, code generation, bug fixing, test case generation, and code summarization.

By streamlining repetitive tasks, this project reduces manual effort, improves productivity, and ensures efficient project delivery.

1. **Objectives**

* To enhance the Software Development Life Cycle (SDLC) using Artificial Intelligence.
* To automate key phases like requirement gathering, analysis, coding, testing, and maintenance.
* To provide developers with an interactive AI assistant for SDLC-related queries.
* To classify raw requirements from PDF documents into structured SDLC tasks.
* To generate AI-powered user stories and executable code.
* To provide bug fixing and code summarization features for faster debugging.
* To enable a simple and user-friendly Streamlit dashboard for developers.

**3. System Requirements**

**Hardware Requirements**

* Processor: Intel i3/i5 or higher
* RAM: Minimum 8 GB
* Hard Disk: 20 GB free space
* GPU: (Optional) CUDA-enabled GPU for faster AI inference

**Software Requirements**

Operating System: Windows / Linux / macOS

Python 3.10+

Libraries/Frameworks:

FastAPI (Backend services)

Streamlit (Frontend dashboard)

IBM Watsonx AI Models

LangChain (Conversation management)

PyMuPDF / PyPDF2 (PDF handling)

Uvicorn (ASGI server)

**4. Implementation**

The project implementation followed a milestone-driven approach:

1. Requirement Gathering & Research → Studied existing SDLC challenges.

2. Model Selection → IBM Watsonx Granite model selected for NLP tasks.

3. Architecture Design → Defined modular architecture (agents, core, models).

4. Backend Development → APIs built with FastAPI, integrated with LangChain.

5. Frontend Development → Created interactive dashboard using Streamlit.

6. Testing & Debugging → Performed functional testing on classification, summarization, and generation modules.

7. Deployment → Launched the system locally and prepared for cloud deployment.

**5. Project Files and Modules**

* app/
* agents/
* context\_agent.py – Handles context-aware responses.
* planner\_agent.py – Assists in planning and task breakdown.
* api/ – API endpoints for SDLC functionalities.
* core/ – Core backend utilities.
* db/ – Stores user session data and task outputs.

**Langchain\_module/**

* conversation\_chain.py – Manages AI conversations.
* conversation\_handler.py – Routes user inputs to correct module.

**models/**

* ai\_task.py – Defines AI-powered SDLC tasks.
* feedback\_model.py – Collects and processes user feedback.
* main.py – Central script integrating backend + frontend.

**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

# Extract text from 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)}"

# Eco tips generator

def eco\_tips\_generator(problem\_keywords):

prompt = f"Generate practical and actionable eco-friendly tips for sustainable living related to: {problem\_keywords}. Provide specific solutions and suggestions:"

return generate\_response(prompt, max\_length=1000)

# Policy summarization

def policy\_summarization(pdf\_file, policy\_text):

# Get text from PDF or direct input

if pdf\_file is not None:

content = extract\_text\_from\_pdf(pdf\_file)

summary\_prompt = f"Summarize the following policy document and extract the most important points, key provisions, and implications:\n\n{content}"

else:

summary\_prompt = f"Summarize the following policy document and extract the most important points, key provisions, and implications:\n\n{policy\_text}"

return generate\_response(summary\_prompt, max\_length=1200)

# Create Gradio interface

with gr.Blocks() as app:

gr.Markdown("# Eco Assistant & Policy Analyzer")

with gr.Tabs():

# Eco Tips Generator tab

with gr.TabItem("Eco Tips Generator"):

with gr.Row():

with gr.Column():

keywords\_input = gr.Textbox(

label="Environmental Problem/Keywords",

placeholder="e.g., plastic, solar, water waste, energy saving...",

lines=3

)

generate\_tips\_btn = gr.Button("Generate Eco Tips")

with gr.Column():

tips\_output = gr.Textbox(label="Sustainable Living Tips", lines=15)

generate\_tips\_btn.click(eco\_tips\_generator, inputs=keywords\_input, outputs=tips\_output)

# Policy Summarization tab

with gr.TabItem("Policy Summarization"):

with gr.Row():

with gr.Column():

pdf\_upload = gr.File(label="Upload Policy PDF", file\_types=[".pdf"])

policy\_text\_input = gr.Textbox(

label="Or paste policy text here",

placeholder="Paste policy document text...",

lines=5

)

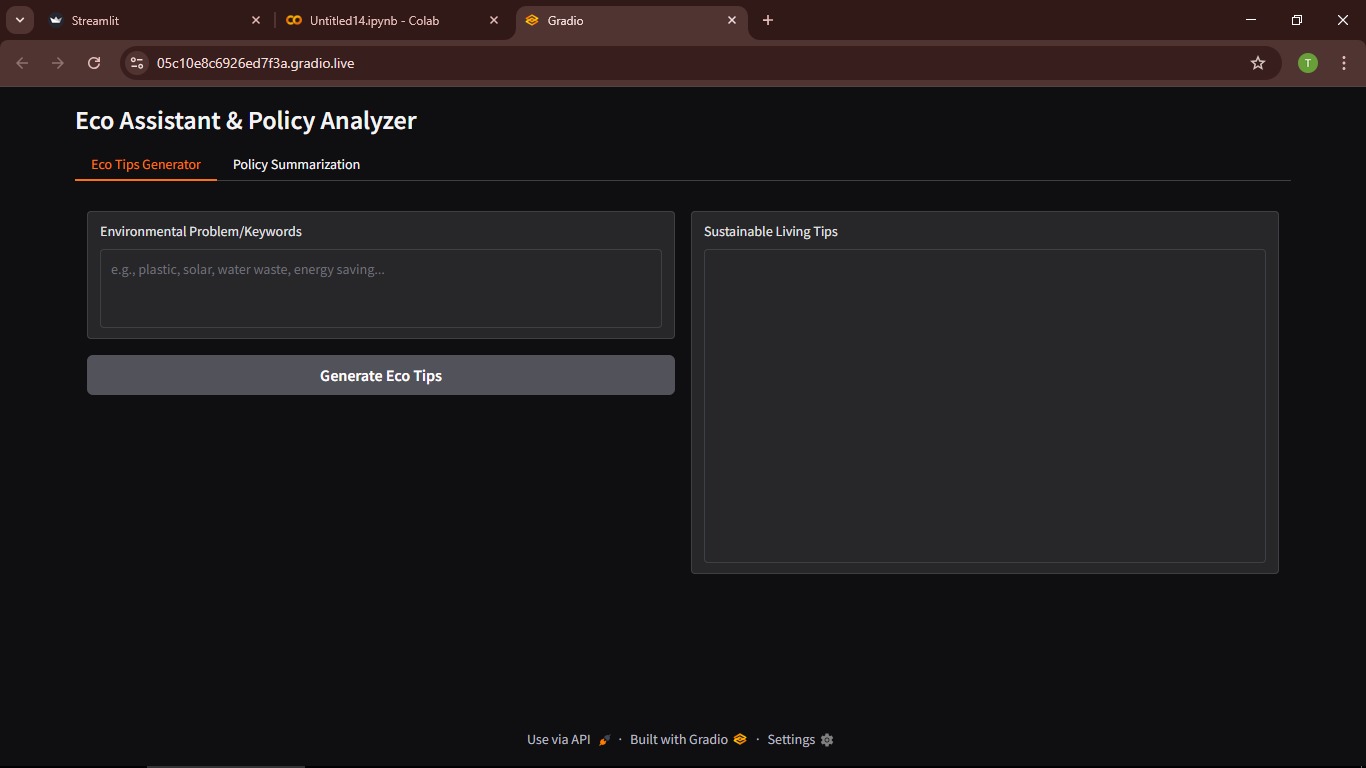
summarize\_btn = gr.Button("Summarize Policy")

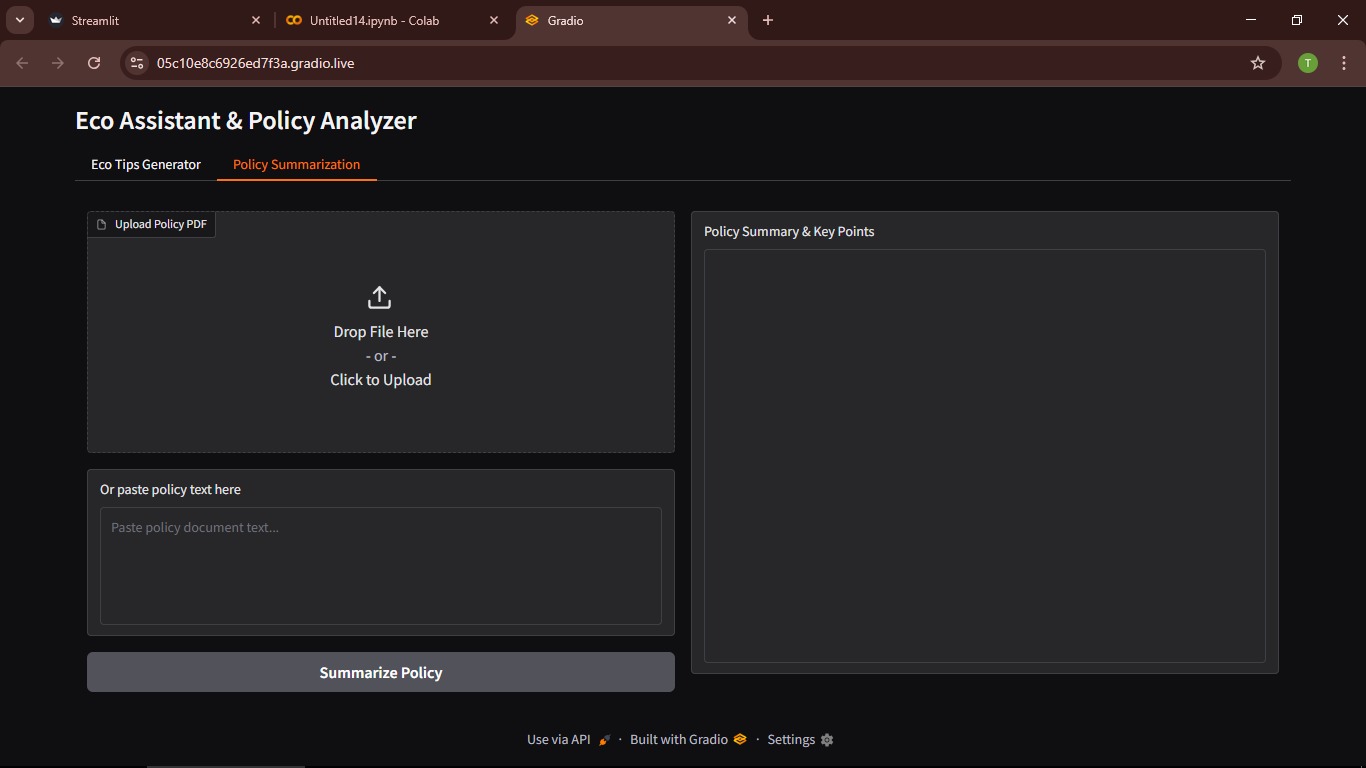
with gr.Column():

summary\_output = gr.Textbox(label="Policy Summary & Key Points", lines=20)

summarize\_btn.click(policy\_summarization, inputs=[pdf\_upload, policy\_text\_input], outputs=summary\_output)

app.launch(share=True)

**7. Output Screenshot**



1. **Conclusion**

The project SMART SDLC demonstrates how AI can transform software development processes.

It provides:

* Faster requirement analysis.
* Automated generation of user stories & code.
* Simplified bug fixing and test generation.
* An intelligent chatbot for learning and clarifying SDLC concepts.

This project reduces developer workload, increases accuracy, and shortens development timelines. With further enhancement, it can be scaled into a full-fledged AI development assistant for real-world software engineering projects.