# Imports

import gradio as gr

import torch

from transformers import AutoTokenizer, AutoModelForCausalLM, pipeline

import PyPDF2

# SmartSDLC-AI Core Class

class SmartSDLC\_AI:

    def \_\_init\_\_(self):

        self.model\_name = "ibm-granite/granite-3.3-2b-instruct"

        self.tokenizer = None

        self.model = None

        self.pipeline = None

        self.load\_model()

    def load\_model(self):

        try:

            print("🔄 Loading AI model...")

            self.tokenizer = AutoTokenizer.from\_pretrained(self.model\_name, trust\_remote\_code=True)

            self.model = AutoModelForCausalLM.from\_pretrained(

                self.model\_name,

                torch\_dtype=torch.float16,

                device\_map="auto",

                trust\_remote\_code=True,

            )

            self.pipeline = pipeline(

                "text-generation",

                model=self.model,

                tokenizer=self.tokenizer,

                max\_length=1024,

                temperature=0.7,

                do\_sample=True,

                pad\_token\_id=self.tokenizer.eos\_token\_id

            )

            print("✅ AI model loaded.")

        except Exception as e:

            print(f"❌ Error: {e}")

            print("⚠️ Falling back to DialoGPT-medium...")

            fallback\_model = "microsoft/DialoGPT-medium"

            self.tokenizer = AutoTokenizer.from\_pretrained(fallback\_model)

            self.model = AutoModelForCausalLM.from\_pretrained(fallback\_model)

            self.pipeline = pipeline(

                "text-generation",

                model=self.model,

                tokenizer=self.tokenizer,

                max\_length=1024,

                temperature=0.7,

                do\_sample=True,

                pad\_token\_id=self.tokenizer.eos\_token\_id

            )

            print("✅ Fallback model loaded.")

    def analyze\_requirements(self, text):

        prompt = f"You are a software requirement analysis assistant. Analyze the following requirements and list key functionalities, ambiguities, and improvement suggestions:\n\n{text}\n\nResponse:"

        response = self.pipeline(prompt)

        result = response[0]['generated\_text'].split("Response:")[-1].strip()

        return result

    def generate\_code(self, description):

        prompt = f"You are a software code generation assistant. Based on the following description, generate Python code:\n\n{description}\n\nCode:"

        response = self.pipeline(prompt)

        result = response[0]['generated\_text'].split("Code:")[-1].strip()

        return result

# 📑 PDF Text Extraction Function

def extract\_text\_from\_pdf(file\_obj):

    reader = PyPDF2.PdfReader(file\_obj)

    text = ""

    for page in reader.pages:

        text += page.extract\_text() or ""

    return text

# 🖥️ Gradio Interface Builder

def create\_gradio\_interface():

    with gr.Blocks(title="SmartSDLC-AI") as app:

        gr.HTML("<h1 style='text-align:center;'>🛠️ SmartSDLC-AI</h1><p style='text-align:center;'>AI-powered Requirement Analysis & Code Generation Assistant</p>")

        with gr.Tabs():

            # 📄 Requirement Analysis Tab

            with gr.Tab("📄 Requirement Analysis"):

                with gr.Row():

                    pdf\_input = gr.File(label="Upload PDF Requirements")

                    text\_input = gr.Textbox(label="Or Enter Requirements Prompt", lines=6)

                analyze\_btn = gr.Button("Analyze Requirements")

                analysis\_output = gr.Textbox(label="Requirement Analysis Result", lines=12)

            # 💻 Code Generation Tab

            with gr.Tab("💻 Code Generation"):

                code\_desc\_input = gr.Textbox(label="Describe the Functionality for Code Generation", lines=6)

                generate\_code\_btn = gr.Button("Generate Code")

                code\_output = gr.Code(label="Generated Python Code", language="python")

        # Requirement Analysis Function

        def handle\_analysis(pdf\_file, prompt\_text):

            if pdf\_file:

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

            elif prompt\_text.strip():

                text = prompt\_text

            else:

                return "❗ Please upload a PDF or enter requirement text."

            result = smart\_sdlc\_ai.analyze\_requirements(text)

            return result

        analyze\_btn.click(fn=handle\_analysis, inputs=[pdf\_input, text\_input], outputs=analysis\_output)

        # Code Generation Function

        def handle\_code\_generation(desc):

            if not desc.strip():

                return "❗ Please enter a description for code generation."

            result = smart\_sdlc\_ai.generate\_code(desc)

            return result

        generate\_code\_btn.click(fn=handle\_code\_generation, inputs=code\_desc\_input, outputs=code\_output)

        gr.HTML("<p style='text-align:center; color:gray;'>⚙️ Powered by IBM Granite AI | SmartSDLC-AI for Modern Development</p>")

    return app

# 🚀 Run Application

if \_\_name\_\_ == "\_\_main\_\_":

    print("🚀 SmartSDLC-AI Initializing...")

    smart\_sdlc\_ai = SmartSDLC\_AI()

    iface = create\_gradio\_interface()

    print("🌐 Launching SmartSDLC-AI with public link...")

    iface.launch(share=True)