Smart SDLC – AI Enhanced software Development Lifecycle

Generative AI with IBM

1.INTRODUCTION:

Smart SDLC – AI Enhanced software development life cycle

Team Members

- R.priyadharshini.
- M.Vishnupriya.
- S.Dharani.

2.PROJECT OVERVIEW :

The Smart SDLC is a next-generation approach to software development that integrate Artificial Intelligence (AI) tools and automation across every phase of the Software Development Lifecycle (SDLC). Unlike traditional SDLC models (Waterfall, Agile, DevOps), the Smart SDLC leverages AI to optimize planning, reduce human error, accelerate delivery, and ensure higher quality outcomes.

- Conversation Interface

The Conversational Interface makes SDLC "Smart" by allowing developers, testers, and clients to interact with AI naturally—like chatting—while AI handles documentation, coding, testing, and deployment.

- Policy summarization

Policies in Smart SDLC ensure that AI helps in development responsibly—protecting data, maintaining quality, and always keeping humans in control.

- Resources Forecasting

Resources forecasting in Smart SDLC uses AI to smartly plan people, time, money, tools, and infrastructure so that software projects finish on time, within budget, and with the right team support.

- Eco – Tip Generator

Eco-Tip generation in Smart SDLC means AI gives smart, green-friendly suggestions at every stage of development, making software projects more sustainable and energy-efficient.

- Citizen Feedback loop

Citizen Feedback Loop in Smart SDLC means AI collects, analyzes, and applies user feedback continuously, making software smarter, faster, and more user-focused.

- KPI Fore casting

KPI Forecasting in Smart SDLC means AI predicts project success by tracking time, cost, quality, and user satisfaction, so teams can fix problems before they happen.

- Anomaly Detection

Anomaly Detection in Smart SDLC means AI continuously watches for unusual errors or risks across all phases, helping teams fix problems quickly and keep software reliable.

- Multimodal Input support

In short: Multimodal Input Support in Smart SDLC means users can interact with the AI system through text, voice, images, or sketches—making software development smarter, easier, and more accessible to everyone.

- Streamlit to Gradio UI

In short: Streamlit to Gradio UI in Smart SDLC means using Streamlit for dashboards and project tracking, and Gradio for AI-powered conversational interfaces—making the whole development lifecycle interactive, visual, and user-friendly.

3.ARCHITECTURE:

The front-end architecture of Smart SDLC provides an AI-powered, multimodal, user-friendly interface (dashboards + chatbot + feedback forms) that connects people with the Smart SDLC system smoothly.

The back-end architecture of Smart SDLC is the AI-powered engine with databases, logic, and cloud infrastructure that processes inputs, runs AI models, and delivers results to the front end.

- LLM integration

LLM integration in Smart SDLC connects AI models to the development process, making the system conversational, intelligent, and capable of automating tasks like requirement analysis, coding, testing, and feedback handling.

- Vector sector

Vector Sector Architecture in Smart SDLC means storing project knowledge as vectors, so the AI can recall past requirements, code, bugs, and feedback—making development faster, smarter, and more accurate.

- ML Modules

ML Modules in Smart SDLC are specialized AI components (like mini-experts) for requirements, coding, testing, forecasting, and feedback—working together to make the development process faster, smarter, and more reliable.

4. SETUP INSTRUCTION:

Prerequisites :

- Install Python 3.8+ for AI/ML modules.
- Install Streamlit / Gradio for front-end UI.
- Set up TensorFlow / PyTorch for ML models.
- Install LangChain / Transformers for LLM integration.
- Configure a Vector Database (FAISS, Pinecone, or Weaviate).

Installation process:

- Install Python 3.8+ on your system.
- Install required libraries using pip install -r requirements.txt.
- Set up virtual environment for project dependencies.
- Install Streamlit / Gradio for front-end UI.
- Configure and install TensorFlow / PyTorch for ML models.

5. FOLDER STRUCTURE:

Frontend → UI

Backend → APIs & services

Ml_modules → AI/ML smart modules

Llm_integration → Large Language Model setup

Vector_store → Vector database & embeddings

Tests → Quality check

Configs → Settings

Docs → Documentation

6. Running the Application :

To start the project:

- Open terminal and clone the project from GitHub.
- Navigate to the project folder using cd Smart-SDLC-AI.
- Create and activate a virtual environment.
- Install dependencies using pip install -r requirements.txt.
- Set up database and vector store (initialize configs).
- Configure API keys (OpenAI/LLM, database, etc.).
- Start backend server using python backend/api/main.py.
- Run frontend UI using streamlit run frontend/app.py or gradio app.py.

Frontend (Stream lit):

Enter frontend folder → Run Streamlit → Open browser → Use Smart SDLC UI.

Backend (Fast API):

Enter backend folder → Run FastAPI server → Open API docs → Backend ready for Smart SDLC UI integration.

7. API DOCUMENTATION:

These backend APIs allow the frontend and other modules to interact with Smart SDLC AI features like requirements analysis, code generation, testing, anomaly detection, KPI/resource forecasting, eco-tips, and citizen feedback.

8. AUTHENTICATION:

- Prevents unauthorized access to sensitive project data.
- Protects AI models, code, and feedback data.
- Ensures role-specific access for better control.
- Builds trust among users and stakeholders.

Authentication in Smart SDLC ensures that only verified users can access the system, with secure login, tokens, and role-based access to protect sensitive data and AI operations.

9.USER INTERFACE :

- Makes AI features accessible to non-technical users.
- Provides real-time updates and visual insights.
- Supports collaboration across roles.
- Improves usability, efficiency, and decision-making.

The User Interface in Smart SDLC is a smart, interactive dashboard and chatbot system that lets all users interact with AI modules, track progress, submit feedback, and collaborate easily.

10.TESTING:

- Testing in Smart SDLC uses AI to automatically generate tests, detect defects, monitor performance, and provide feedback—ensuring high-quality, reliable software.
- Smart SDLC uses AI and automation to enhance the traditional software development lifecycle (SDLC). It integrates features like LLM, ML modules, anomaly detection, KPI/resource forecasting, multimodal input, and citizen feedback to make development smarter, faster, and more reliable.

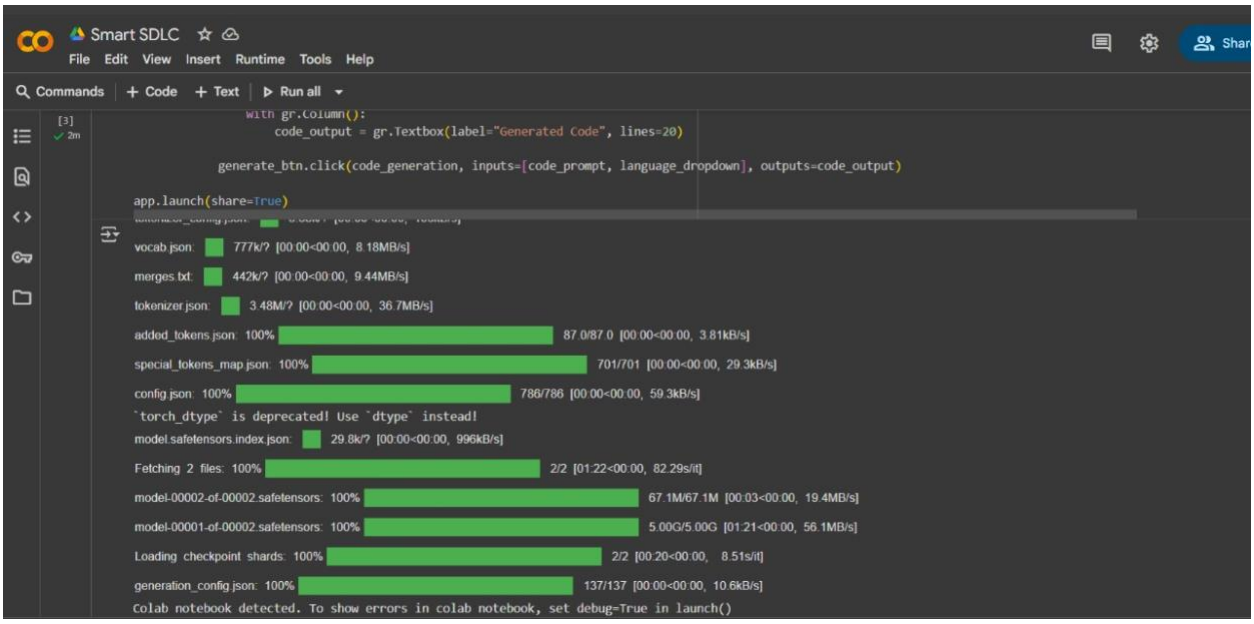
11.KNOWN ISSUES:

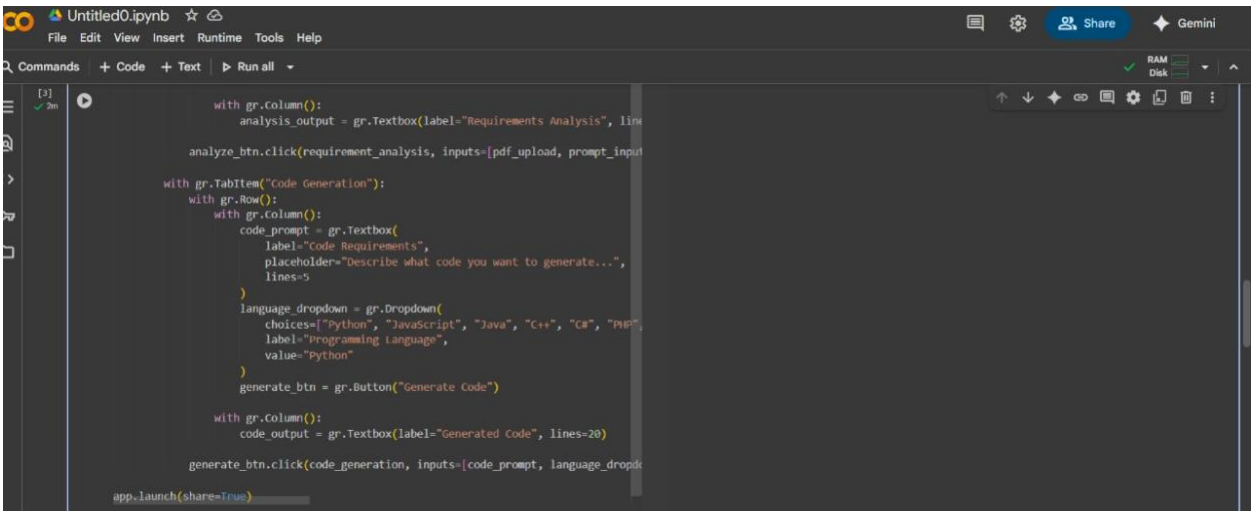
The Smart SDLC platform, while highly advanced, has a few known issues. Large queries to the LLM may sometimes take longer to respond, causing minor delays in user interaction. AI-generated code or suggestions can occasionally be incomplete or require manual corrections. Multimodal input, such as voice commands or image uploads, may not always work perfectly, especially in noisy environments or with unclear images. There are rare instances where the vector store fails to retrieve relevant past project data, affecting the context-aware responses of the AI. The automated testing and anomaly detection modules can sometimes produce false positives, flagging normal behavior as issues. Additionally, extensive use of LLM APIs may reach request limits, and some users may face access restrictions due to misconfigured roles in the role-based access control system. Lastly, system performance is dependent on stable cloud infrastructure and internet connectivity. Despite these limitations, the team is continuously monitoring and improving the system to enhance reliability and efficiency.

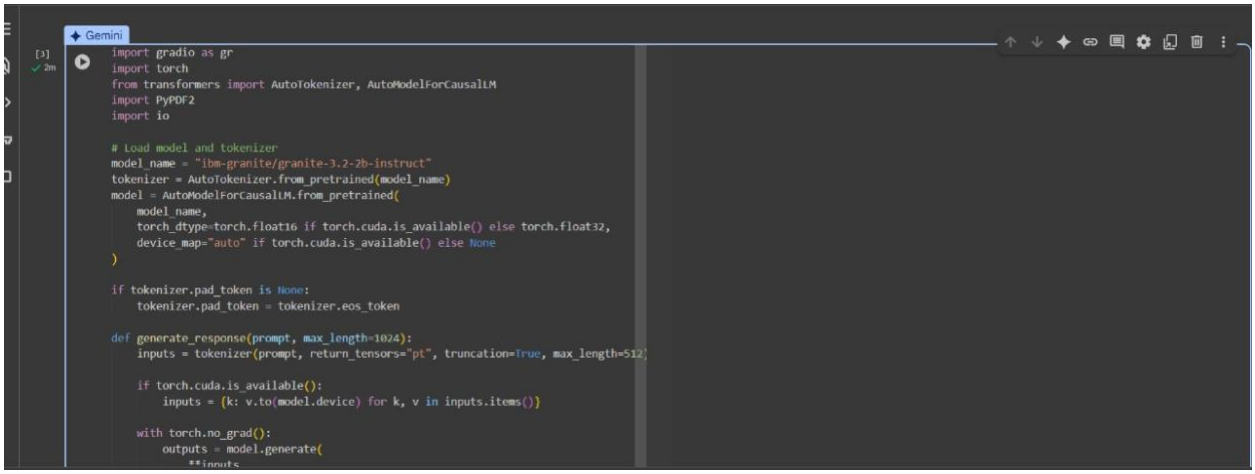
12. FUTURE ENHANCEMENT:

In the future, Smart SDLC can be enhanced to become even more intelligent and user-friendly. One possible improvement is integrating more advanced LLMs and domain-specific AI models to provide faster and more accurate code generation, requirement analysis, and testing suggestions. The system can also support real-time collaboration, allowing multiple developers, testers, and managers to work together seamlessly on the same project. Expanding multimodal input capabilities, such as improved voice recognition, sketch-to-code conversion, and video input analysis, can make the platform more accessible. Additionally, predictive analytics for project risk, cost, and resource management can be strengthened with historical data and machine learning algorithms. The platform can also integrate automated deployment pipelines and DevOps tools, allowing end-to-end automation from coding to production. Finally, enhancing security, role management, and cloud scalability will make Smart SDLC more robust, reliable, and suitable for larger enterprise projects.

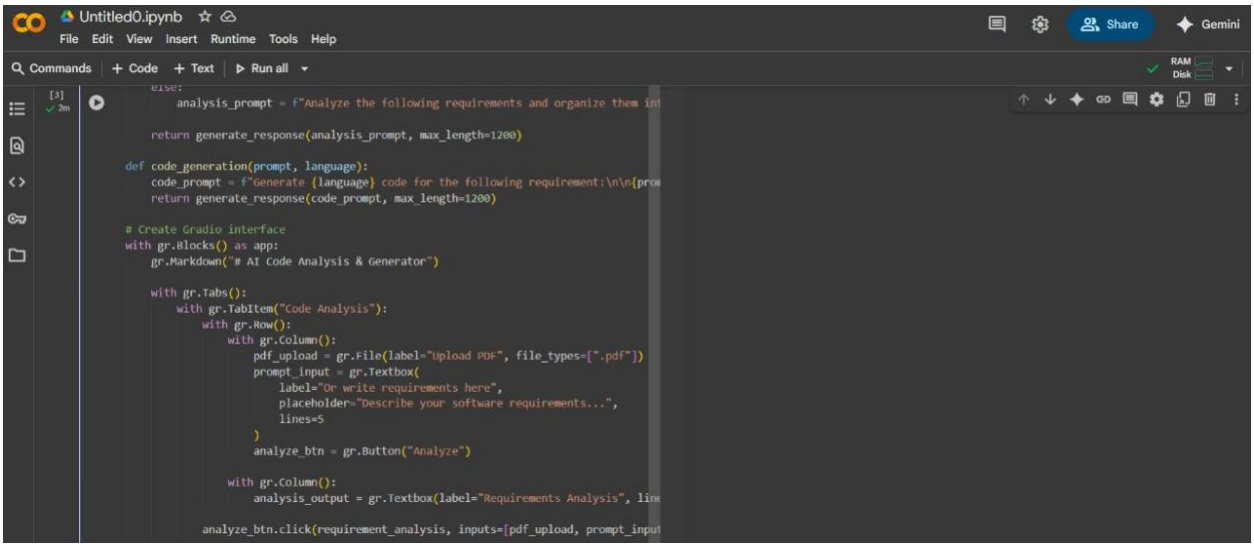
13.PROJECT SCREENSHOT


Smart SDLC
File Edit View Insert Runtime Tools Help
Commands + Code + Text Run all
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)
vocab.json: 777k/? [00:00<00:00, 8.18MB/s]
merges.txt: 442k/? [00:00<00:00, 9.44MB/s]
tokenizer.json: 3.48M/? [00:00<00:00, 36.7MB/s]
added_tokens.json: 100% 87.0/87.0 [00:00<00:00, 3.81kB/s]
special_tokens_map.json: 100% 701/701 [00:00<00:00, 29.3kB/s]
config.json: 100% 786/786 [00:00<00:00, 59.3kB/s]
`torch_dtype` is deprecated! Use `dtype` instead!
model.safetensors.index.json: 29.8k/? [00:00<00:00, 996kB/s]
Fetching 2 files: 100% 2/2 [01:22<00:00, 82.29s/it]
model-00002-of-00002.safetensors: 100% 67.1M/67.1M [00:03<00:00, 19.4MB/s]
model-00001-of-00002.safetensors: 100% 5.00G/5.00G [01:21<00:00, 56.1MB/s]
Loading checkpoint shards: 100% 2/2 [00:20<00:00, 8.51s/it]
generation_config.json: 100% 137/137 [00:00<00:00, 10.6kB/s]
Colab notebook detected. To show errors in colab notebook, set debug=True in launch()

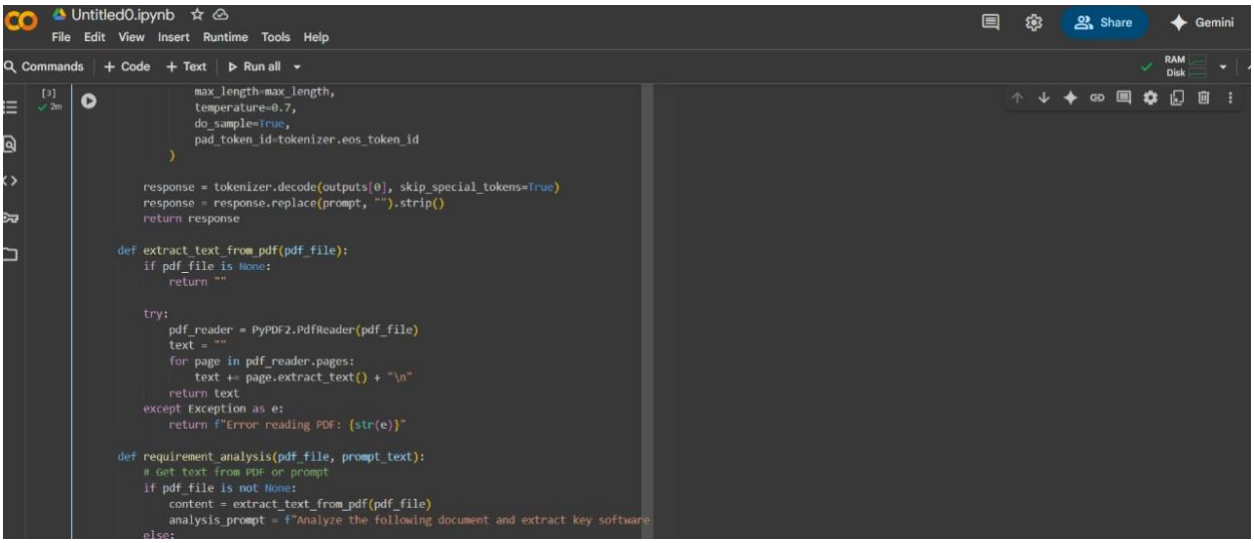

Untitled0.ipynb
File Edit View Insert Runtime Tools Help
Share Gemini
Commands + Code + Text Run all
RAM Disk
with gr.Column():
analysis_output = gr.Textbox(label="Requirements Analysis", line
analyze_btn.click(requirement_analysis, inputs=[pdf_upload, prompt_input
with gr.TabItem("Code Generation"):
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",
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_dropdo
app.launch(share=True)



```
Gemini
[3]
2m
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
```



```
Untitled0.ipynb
File Edit View Insert Runtime Tools Help
Share Gemini
Commands + Code + Text Run all
RAM Disk
[3]
2m
        else:
            analysis_prompt = f"Analyze the following requirements and organize them int

        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{pro
        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", lin

                analyze_btn.click(requirement_analysis, inputs=[pdf_upload, prompt_input
```



```
Untitled0.ipynb
File Edit View Insert Runtime Tools Help
Share Gemini
Commands + Code + Text Run all
RAM Disk
[3]
2m
                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
        else:
```

Thank you…..