Smart SDLC – AI-Enhanced Software Development Lifecycle

Project Documentation

1.Introduction

- Project title: Smart SDLC – AI-Enhanced Software Development Lifecycle
- Team member: Meenatchi K
- Team member: Madhumithra R
- Team member: Abarna L
- Team member: Madhumeetha K

2.Project Overview:

- Purpose:

  The purpose of the Smart SDLC project is to modernize and enhance the traditional software development life cycle by integrating intelligent tools, automation, and data-driven approaches. It aims to streamline each phase of development, from requirement gathering to deployment, by reducing manual effort, improving collaboration, and ensuring higher efficiency. By incorporating smart features such as automated testing, predictive analytics, and adaptive workflows, the project seeks to deliver software faster, with better quality and lower

cost. Ultimately, the Smart SDLC project is designed to make the development process more flexible, reliable, and capable of adapting to changing business needs and technological advancements.

- Features:

Automation of tasks – reduces manual effort in testing, deployment, and documentation.

Smart requirement analysis – identifies gaps, inconsistencies, and suggests improvements.

Continuous Integration and Deployment (CI/CD) – ensures faster and reliable software delivery.

Predictive analytics – forecasts risks, project delays, and potential defects.

Enhanced collaboration – improves communication among developers, testers, and stakeholders.

Flexibility and adaptability – allows easy adjustments to changing requirements.

Automated quality assurance – ensures high-quality outputs with minimal human errors.

Efficient resource utilization – reduces development cost and time.

Real-time monitoring and feedback – tracks progress and improves decision-making.

Data-driven decision support – helps in better planning and execution of each SDLC phase.

3.Architecture

Frontend (Stream lit):

The frontend is built with Stream lit, offering an interactive web UI with
multiple pages including dashboards, file uploads, chat interface, feedback
forms, and report viewers. Navigation is handled through a sidebar using the
stream lit-option-menu library. Each page is modularized for scalability.

Backend (Fast API):

Fast API serves as the backend REST framework that powers API endpoints for
document processing, chat interactions, eco tip generation, report creation,
and vector embedding. It is optimized for asynchronous performance and easy Swagger integration.

LLM Integration (IBM Watsonx Granite):

Granite LLM models from IBM Watsonx are used for natural language
understanding and generation. Prompts are carefully designed to generate
summaries, sustainability tips, and reports.

Vector Search (Pinecone):

Uploaded policy documents are embedded using Sentence Transformers and stored in Pinecone. Semantic search is implemented using cosine similarity to allow users to search documents using natural language queries.

ML Modules (Forecasting and Anomaly Detection):

Lightweight ML models are used for forecasting and anomaly detection using
Scikit-learn. Time-series data is parsed, modeled, and visualized using pandas
and matplotlib.

4. Setup Instructions

Prerequisites:

1. Gradio Framework Knowledge: Gradio Documentation

2. IBM Granite Models (Hugging Face): IBM Granite models

3. Python Programming Proficiency: Python Documentation

4. Version Control with Git: Git Documentation

5. Google Collab's T4 GPU Knowledge: Google collab

Installation Process:

o Clone the repository

o Install dependencies from requirements.txt

o Create a .env file and configure credentials

o Run the backend server using Fast API

o Launch the frontend via Stream lit

o Upload data and interact with the modules

5. Folder Structure

app/ – Contains all Fast API backend logic including routers, models, and

integration modules.

app/api/ – Subdirectory for modular API routes like chat, feedback, report, and

document vectorization.

ui/ – Contains frontend components for Stream lit pages, card layouts, and

form UIs.

smart_dashboard.py – Entry script for launching the main Stream lit

dashboard.

granite_llm.py – Handles all communication with IBM Watsonx Granite model

including summarization and chat.

document_embedder.py – Converts documents to embeddings and stores in

Pinecone.

kpi_file_forecaster.py – Forecasts future energy/water trends using regression.

anomaly_file_checker.py – Flags unusual values in uploaded KPI data.

report_generator.py – Constructs AI-generated sustainability reports.

6. Running the Application

To start the project:

➢ Launch the FastAPI server to expose backend endpoints.

➢ Run the Streamlit dashboard to access the web interface.

➢ Navigate through pages via the sidebar.

➢ Upload documents or CSVs, interact with the chat assistant, and view outputs like reports, summaries, and predictions.

Frontend (Stream lit):

The frontend is built with Stream lit, offering an interactive web UI with multiple pages including dashboards, file uploads, chat interface, feedback forms, and report viewers. Navigation is handled through a sidebar using the stream lit-option-menu library. Each page is modularized for scalability.

Backend (Fast API):

Fast API serves as the backend REST framework that powers API endpoints for document processing, chat interactions, eco tip generation, report creation, and vector embedding. It is optimized for asynchronous performance and easy Swagger integration.

7. API Documentation

Backend APIs available include:

POST /chat/ask – Accepts a user query and responds with an AI-generated message

POST /upload-doc – Uploads and embeds documents in Pinecone

GET /search-docs – Returns semantically similar policies to the input query

GET /get-eco-tips – Provides sustainability tips for selected topics like energy, water, or waste

POST /submit-feedback – Stores citizen feedback for later review or analytics each endpoint is tested and documented in Swagger UI for quick inspection and trial during development.

8. Authentication

1. User Registration & Login

Secure sign-up with username, email, and strong password.

Login with encryption (hashed & salted passwords).

2. Role-Based Access Control (RBAC)

Admin: Full control (manage users, assign roles, monitor all projects).

Project Manager: Assign tasks, track progress, view reports.

Developer/Tester: Access assigned modules, update status, submit code/tests.

Client/Stakeholder: View project progress only.

3. Two-Factor Authentication (2FA) (optional for security)

OTP via email/SMS or authentication apps (like Google Authenticator).

4. Single Sign-On (SSO) Integration (optional advanced feature)

Allows login using Google, GitHub, or corporate accounts.

6. Password Management

Enforce strong password policies.

Forgot password & reset option with secure email/OTP verification.

7. Audit & Logging

Every logi n, logout, and failed attempt is logged.

Admin can review authentication logs for security.

9. User Interface

The interface is minimalist and functional, focusing on accessibility for non-technical users. It includes:

Sidebar with navigation

KPI visualizations with summary cards

Tabbed layouts for chat, eco tips, and forecasting

Real-time form handling

PDF report download capability

The design prioritizes clarity, speed, and user guidance with help texts and intuitive flows.

10. Testing

Testing was done in multiple phases:

Unit Testing: For prompt engineering functions and utility scripts

API Testing: Via Swagger UI, Postman, and test scripts

Manual Testing: For file uploads, chat responses, and output consistency

Edge Case Handling: Malformed inputs, large files, invalid API keys

Each function was validated to ensure reliability in both offline and AP

11.Screen shots

PROGRAM:

```
import gradio as gr
import torch
from transformers import AutoTokenizer, AutoModelForCausalLM
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(
```

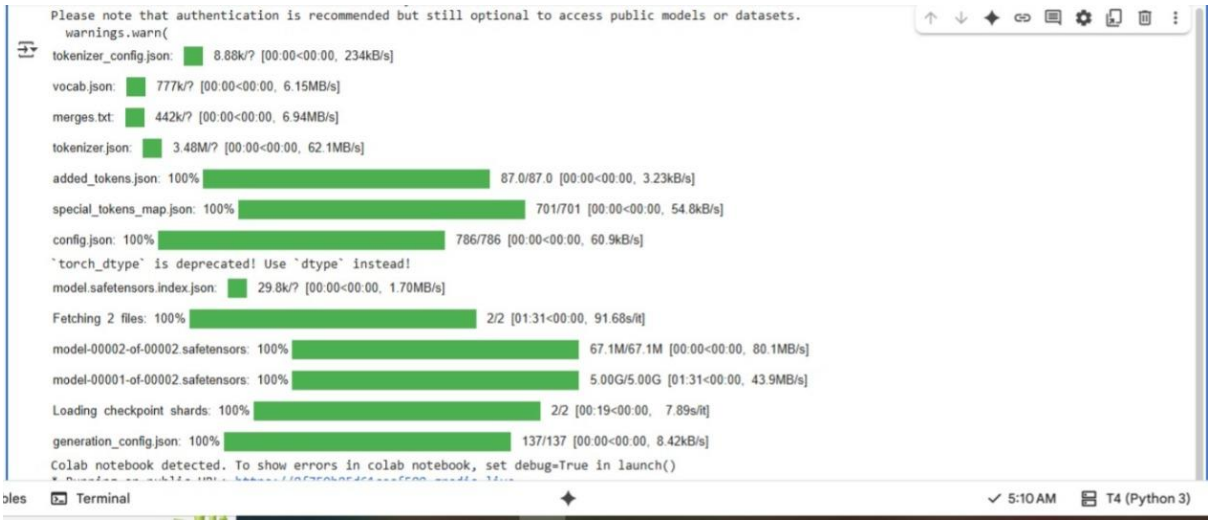
```
            **inputs,
            max_length=max_length,
            temperature=0.7,
            do_sample=True,
            pad_token_id=tokenizer.eos_token_id
        )

    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
```

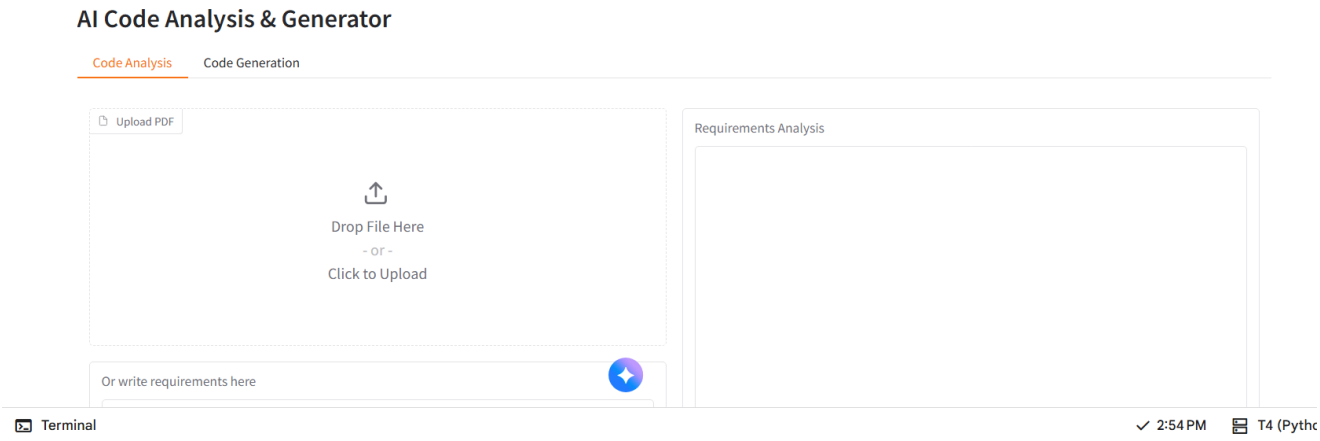
```
    if pdf_file is not None:
        content = extract_text_from_pdf(pdf_file)
        analysis_prompt = f"Analyze the following document and extract key software requirements. Organize them into functional requirements, non-functional
    else:
        analysis_prompt = f"Analyze the following requirements and organize them into functional requirements, non-functional requirements, and technical sp

    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
                    )
```

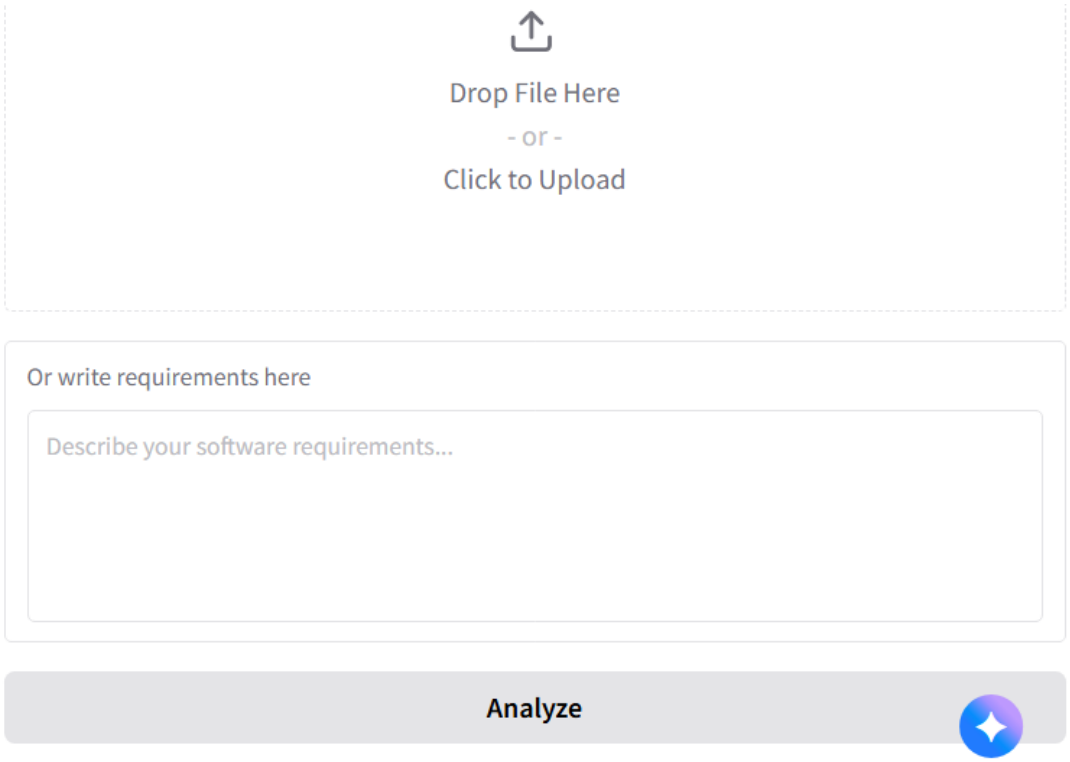
```
            )
            analyze_btn = gr.Button("Analyze")

        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 = 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)
```

OUTPUT:


Please note that authentication is recommended but still optional to access public models or datasets.
warnings.warn(
tokenizer_config.json: 8.88k/? [00:00<00:00, 234kB/s]
vocab.json: 777k/? [00:00<00:00, 6.15MB/s]
merges.txt: 442k/? [00:00<00:00, 6.94MB/s]
tokenizer.json: 3.48M/? [00:00<00:00, 62.1MB/s]
added_tokens.json: 100% 87.0/87.0 [00:00<00:00, 3.23kB/s]
special_tokens_map.json: 100% 701/701 [00:00<00:00, 54.8kB/s]
config.json: 100% 786/786 [00:00<00:00, 60.9kB/s]
`torch_dtype` is deprecated! Use `dtype` instead!
model.safetensors.index.json: 29.8k/? [00:00<00:00, 1.70MB/s]
Fetching 2 files: 100% 2/2 [01:31<00:00, 91.68s/it]
model-00002-of-00002.safetensors: 100% 67.1M/67.1M [00:00<00:00, 80.1MB/s]
model-00001-of-00002.safetensors: 100% 5.00G/5.00G [01:31<00:00, 43.9MB/s]
Loading checkpoint shards: 100% 2/2 [00:19<00:00, 7.89s/it]
generation_config.json: 100% 137/137 [00:00<00:00, 8.42kB/s]
Colab notebook detected. To show errors in colab notebook, set debug=True in launch()
Terminal
5:10 AM
T4 (Python 3)


AI Code Analysis & Generator
Code Analysis
Code Generation
Upload PDF
Drop File Here
- or -
Click to Upload
Requirements Analysis
Or write requirements here
Terminal
2:54 PM
T4 (Pytho



12. Known issues

1. Complexity in Automation

Integrating smart tools like AI for testing, deployment, or requirement analysis can be complex.There may be a steep learning curve for team members.

2. Data Dependency

Smart SDLC relies heavily on accurate and up-to-date project data.

Poor data quality can lead to wrong predictions, inaccurate progress tracking, or flawed automation.

3. Integration Challenges

Integrating the Smart SDLC system with existing tools (JIRA, Git, CI/CD pipelines) may face compatibility issues.

Legacy systems might not support automated features.

4. High Initial Cost

Implementing smart features like AI-based testing, analytics dashboards, and automation can be expensive initially.

5. Security Risks

Storing sensitive project data and using cloud-based tools can introduce security vulnerabilities.

Unauthorized access to project intelligence could be critical.

6. Over-Reliance on Automation

Teams might rely too much on AI predictions, potentially ignoring human judgment and creativity.

Could lead to missed requirements or design flaws.

7. Change Management

Employees may resist transitioning to a smart SDLC approach.

Training and adaptation time is needed for smooth adoption.

8. Maintenance and Updates

Continuous updates and maintenance are required to keep AI models, analytics, and automated processes accurate.

9. Scalability Concerns

As project size grows, the system might slow down if not designed for high scalability.

Automation may not handle very large or complex projects efficiently.

10. False Positives in Predictions

AI/ML modules predicting bugs, delays, or resource allocation may sometimes give inaccurate suggestions.

Misleading insights can affect decision-making.

13. Future Enhancement

1. AI-Powered Requirement Analysis

Use AI to automatically analyze requirement documents. Detect ambiguities, inconsistencies, and missing requirements.

Suggest improvements to make requirement gathering faster and more accurate.

2. Predictive Project Management

Implement ML models to predict project timelines, budget overruns, and resource allocation.

Provide early warnings for potential delays or risks.

3. Automated Testing and QA

Introduce AI-based test case generation and execution.

Automate regression, integration, and performance testing.

Use smart bug prediction systems to detect high-risk areas.

4. Intelligent Resource Allocation

Smart algorithms to assign tasks based on skill, availability, and past performance.

Optimize team productivity and reduce workload imbalance.

5. Continuous Learning System

Implement a feedback loop where past project data is analyzed to improve future predictions and automation.

Adaptive learning for better decision-making in subsequent projects.

6. Cloud Integration & Collaboration

Integrate with cloud platforms for remote collaboration.

Enable real-time monitoring, task updates, and team communication.

7. Enhanced Security & Compliance

Introduce smart security checks and automated compliance auditing.

Ensure sensitive project data is secure and regulatory standards are met.

8. Advanced Reporting & Analytics

Interactive dashboards with KPIs, project health metrics, and predictive insights.

Customizable reports for managers, clients, and team members.

9. Chatbot Assistance

Implement a project assistant chatbot to answer queries, suggest solutions, and provide guidance.Reduces time spent searching for documentation or project status updates.

10. Integration with DevOps

Seamlessly integrate smart SDLC with CI/CD pipelines. Automate deployment, monitoring, and rollback processes with intelligent decision-making.