

SmartSDLC -AI-Enhanced Software Development Lifecycle

Generative AI with IBM

1. Introduction

- **Project Title:** SmartSDLC – AI-Enhanced Software Development Lifecycle
 - **Team Members:**
 - S. Iniyaval
 - R. Deva Dharshini
 - R. Atchaya
 - U. Aafrin
-

2. Project Overview

Purpose:

SmartSDLC aims to integrate AI across the entire software development lifecycle to optimize productivity, improve software quality, and enhance decision-making. By leveraging **Generative AI with IBM Watsonx Granite LLMs**, the framework supports developers, managers, and stakeholders with real-time insights, policy summarization, anomaly detection, and predictive analytics.

Features:

- Conversational Interface (natural language queries)
- Policy & Document Summarization
- Resource & KPI Forecasting
- Eco-Tip Generator (sustainability insights)
- Feedback Loop for stakeholders

- Anomaly Detection (early warnings)
 - Multimodal Input Support (text, PDFs, CSVs)
 - Streamlit/Gradio UI for interaction
-

3. Architecture

- **Frontend (Streamlit):** Interactive dashboards, chat, reports, feedback forms
 - **Backend (FastAPI):** Handles APIs for chat, summarization, embedding, forecasting
 - **LLM Integration (IBM Watsonx Granite):** Provides natural language understanding, summarization, and recommendations
 - **Vector Search (Pinecone):** Embedding + semantic search of documents
 - **ML Modules:** Forecasting & anomaly detection (Scikit-learn, Pandas, Matplotlib)
-

4. Setup Instructions

Prerequisites:

- Python 3.9+
- pip & venv tools
- IBM Watsonx API keys
- Pinecone API key
- Internet connection

Installation Steps:

1. Clone repository

2. Install dependencies (requirements.txt)
 3. Create .env file for credentials
 4. Run backend with FastAPI
 5. Launch frontend with Streamlit
 6. Upload documents and interact
-

5. Folder Structure

- app/ – FastAPI backend logic
 - app/api/ – Modular API routes (chat, feedback, report, embeddings)
 - ui/ – Streamlit pages and layouts
 - smart_dashboard.py – Main dashboard entry point
 - granite_llm.py – Watsonx LLM integration
 - document_embedder.py – Embedding & Pinecone storage
 - kpi_file_forecaster.py – KPI forecasting module
 - anomaly_file_checker.py – Anomaly detection module
 - report_generator.py – AI-generated reports
-

6. Running the Application

1. Start FastAPI server
2. Run Streamlit dashboard
3. Navigate via sidebar
4. Upload docs/CSVs
5. Interact with assistant (chat, reports, predictions)

6. Get outputs in real-time

7. API Documentation

- POST /chat/ask – AI-generated Q&A
 - POST /upload-doc – Document embedding
 - GET /search-docs – Semantic document search
 - GET /get-eco-tips – AI-driven sustainability tips
 - POST /submit-feedback – Feedback storage
-

8. Authentication

- Token-based (JWT / API keys)
 - OAuth2 (IBM Cloud)
 - Role-based access (admin, citizen, researcher)
 - Planned: sessions + history tracking
-

9. User Interface

- Sidebar navigation
 - KPI visualizations with summary cards
 - Tabbed chat, eco-tips, forecasting views
 - Real-time forms
 - Downloadable PDF reports
-

10. Testing

- **Unit Testing:** prompt functions & utilities

- **API Testing:** Swagger UI & Postman
- **Manual Testing:** file uploads, chat, predictions
- **Edge Cases:** large files, malformed inputs, invalid keys

11. Screenshots

The screenshots show the AI Code Analysis & Generator web application interface. The top screenshot displays the 'Code Analysis' tab, which includes a file upload area with a 'Drop File Here' prompt and a 'Click to Upload' button. Below this is a text input field for requirements, containing the text 'operating system'. To the right, the 'Requirements Analysis' section shows a list of functional requirements for a Windows 10 Pro application. The bottom screenshot displays the 'Code Generation' tab, which includes a text input field for code requirements, containing the text 'weather app using API'. Below this is a dropdown menu for the programming language, set to 'Python'. To the right, the 'Generated Code' section shows a Python code snippet for retrieving weather data using the OpenWeatherMap API.

AI Code Analysis & Generator

Code Analysis | Code Generation

Upload PDF

Drop File Here
- OR -
Click to Upload

Or write requirements here

operating system

Analyze

Requirements Analysis

: Windows 10 Pro

1. Application must support multi-threading for efficient processing of large datasets.
2. Application must integrate with a corporate Active Directory for user authentication and access control.
3. Application must implement a secure communication channel using SSL/TLS for data transmission.
4. Application must provide a user-friendly graphical user interface (GUI) for interaction.
5. Application must maintain a log of all user activities for auditing purposes.
6. Application must support real-time data visualization and updates.
7. Application must be capable of handling and processing data from various sources, including but not limited to CSV, JSON, and XML.

Functional Requirements:

1. F: The application shall support multi-threading for efficient processing of large datasets.
2. F: The application shall integrate with a corporate Active Directory for user authentication and access control.
3. F: The application shall implement a secure communication channel using SSL/TLS for data transmission.
4. F: The application shall provide a user-friendly graphical user interface (GUI) for interaction.
5. F: The application shall maintain a log of all user activities for auditing purposes.
6. F: The application shall support real-time data visualization and updates.
7. F: The application shall be capable of handling and processing data from various sources, including but not limited to CSV, JSON, and XML.

Use via API | Built with Gradio | Settings

AI Code Analysis & Generator

Code Analysis | Code Generation

Code Requirements

weather app using API

Programming Language

Python

Generate Code

Generated Code

```
python
import requests

def get_weather(city, api_key):
    """
    Retrieve weather data for a given city using OpenWeatherMap API.

    :param city: str, name of the city
    :param api_key: str, API key from OpenWeatherMap
    :return: dict, weather data
    """
    base_url = "http://api.openweathermap.org/data/2.5/weather"
    params = {
        'q': city,
        'appid': api_key,
        'units': 'metric' # Use metric units (Celsius)
    }
    response = requests.get(base_url, params=params)

    if response.status_code == 200:
        return response.json()
```

Use via API | Built with Gradio | Settings

12. Known Issues

- Limited scalability for very large datasets
 - Latency in high-volume document embedding
 - No offline mode (requires IBM Watsonx API access)
-

13. Future Enhancements

- Advanced security & role management
- Session management & user history
- Deeper Watsonx model fine-tuning
- Expanded forecasting models
- Enterprise/government deployment