SmartSDLC – AI

TEAM ID - (NM2025TMID01900)

1.INTRODUCTION

SMART SDLC refers to a modern, structured approach to software development that integrates the principles of SMART (Specific, Measurable, Achievable, Relevant, and Time-bound) with the traditional **Software Development Life Cycle (SDLC)**. This model aims to enhance the efficiency, clarity, and success of software projects by aligning goals and deliverables at every stage of development.

• Project title : Smart SDLC

❖ G PRAVEEN(TL)	– (A0834E5D760A1599CB54EF13C1D4EC1B)
❖ M MATHAN	– (82BBF3977FA81D7B62A489F0899ADFE5)
❖ S RAHUL	– (9DAD5236C8F8931859C417FE47813C0C)
❖ H SANJAY	– (82BBF3977FA81D7B62A489F0899ADFE5)

2. PROJECT OVERVIEW

The SmartSDLC project is an AI-enhanced platform designed to automate and streamline the Software Development Lifecycle (SDLC) using advanced technologies like IBM Watsonx, FastAPI, LangChain, and Streamlit. It integrates generative AI to handle key SDLC phases, including requirement analysis, code generation, test case creation, bug fixing, and documentation. The platform features a user-friendly interface that allows users to upload PDFs, generate structured requirements, and transform natural language prompts into functional code. It also includes an AI-powered chatbot for real-time assistance and support. The backend, built with FastAPI, efficiently processes API requests, while the frontend, developed using Streamlit, offers a visually appealing and interactive dashboard. The system is modular, scalable, and secure, featuring a robust authentication mechanism and seamless integration between AI models and user inputs. Deployment is streamlined with local hosting via Uvicorn and Streamlit, with comprehensive API documentation available through Swagger UI. By leveraging AI in a practical and efficient manner, SmartSDLC significantly reduces manual workload, accelerates development processes, and enhances software quality, making it an invaluable tool for developers and project teams.

3. SMART(SDLC) ARCHITECTURE

Frontend Components

These handle user interaction, visualization, and assistance:

- User Inputs:
  - PDF uploads, Prompts, Code snippets, Queries.
- Streamlit:
  - Provides a user-friendly interface for input and interaction.
- Floating Chatbot Assistant:
  - Offers real-time help and guidance.
  - Provides an interactive conversational interface for queries and support.

Backend Components

These manage processing, orchestration, and AI-driven automation:

- SmartSDLC Platform (Core Engine)
  - Central processing hub that connects frontend inputs with AI models and tools.
- FastAPI
  - Backend API framework for handling requests and communication.
- LangChain
  - Orchestration framework that connects the SmartSDLC platform with the AI model.
- Granite-20B-Code-Instruct
  - Large language model responsible for:
    - Code generation
    - Requirement analysis
    - Test case creation
    - Code summarization
- Outputs Generated:
  - Structured Requirements

- Generated Code
- Test Case Generator
- Code Summarizer

LLM Integration (IBM Watsonx Granite)

- The SmartSDLC platform leverages IBM Watsonx Granite LLMs (e.g., Granite-20B-Code-Instruct) to power its AI-driven automation. Key roles of LLM integration:
    - Requirement Analysis: Converting natural language or PDF-based requirements into structured formats.
    - Code Generation: Producing optimized source code from prompts or requirement documents.
    - Test Case Creation: Generating unit tests, integration tests, and validation scenarios.
    - Code Summarization: Simplifying complex codebases into concise explanations for easier understanding.

Vector Search (Pinecone)

- SmartSDLC incorporates Pinecone as a vector database for efficient semantic search and context retrieval.
- Use cases within the platform:
    - Requirement Retrieval: Quickly find relevant sections of large PDF documents or historical project data.
    - Code Snippet Search: Enable similarity-based search across codebases for reuse and optimization.
    - Knowledge Base Querying: Improve chatbot assistant responses by retrieving the most contextually relevant information.

ML Modules (Forecasting and Anomaly Detection)

- In addition to LLM-driven automation, SmartSDLC uses machine learning modules to enhance software lifecycle intelligence:
    - Forecasting: Predicting project timelines, resource allocation, or potential bottlenecks based on historical data.

        - Anomaly Detection: Identifying irregularities in code quality, test case results, or system logs to flag potential defects early in the lifecycle.
    - These ML modules complement the LLM outputs by ensuring predictive insights and proactive risk management.

4 SMART (SDLC) – SETUP INSTRUCTIONS

1. Install Python 3.10

- Download Python 3.10 from the official website:
   Python 3.10 Download
- During installation:
    - ✅Check **"Add Python to PATH"**.
    - ✅Select **"Customize installation"** if you want to manage features manually.
- Verify installation:

python –version

2.SETUP VIRTUAL ENVIRONMENT(Recommended)

Creating a virtual environment keeps dependencies clean.

```
# Create venv
python -m venv venv

# Activate venv (Windows)
venv\Scripts\activate

# Activate venv (Mac/Linux)
source venv/bin/activate
```

3.Install Core Dependencies

FastAPI (Backend Framework)

Uvicorn (ASGI Server)

Streamlit (Frontend UI)

LangChain (LLM Orchestration)

PyMuPDF (for PDF Parsing)

Git & GitHub (Version Control)

4. IBM Watsonx AI & Granite Models

- Create an **IBM Cloud account**
- Get API credentials (API key)
- Install IBM Watsonx SDK (Python client)

5. Frontend Libraries

Depending on your UI needs, install additional libraries

5 Folder Structure

```
smartsdlc/
│── backend/                    # FastAPI backend services
│   ├── main.py                 # Entry point for FastAPI
│   ├── routes/                 # API endpoints
│   │   ├── requirements.py     # Endpoints for requirements processing
│   │   ├── codegen.py          # Endpoints for code generation
│   │   ├── testcases.py        # Endpoints for test case generation
│   │   ├── summarize.py        # Endpoints for code summarization
│   ├── services/               # Core business logic
│   │   ├── watsonx_service.py  # IBM Watsonx Granite LLM integration
│   │   ├── langchain_service.py # LangChain orchestration
│   │   ├── vector_service.py   # Pinecone vector search integration
│   │   ├── ml_forecasting.py   # Forecasting module
│   │   ├── ml_anomaly.py       # Anomaly detection module
│   ├── utils/                  # Helper utilities
│   │   ├── pdf_parser.py       # PDF & document processing (PyMuPDF)
│   │   ├── code_utils.py       # Code formatting & validation
│   │   ├── logger.py           # Logging config
│   ├── tests/                # Backend unit/integration tests
│   │   ├── test_codegen.py
│   │   ├── test_requirements.py
│   ├── __init__.py

│── frontend/                 # Streamlit frontend app
│   ├── app.py                # Main Streamlit app
│   ├── pages/                # Multi-page UI (optional)
│   │   ├── 1_Requirements.py
│   │   ├── 2_Code_Generation.py
│   │   ├── 3_Test_Cases.py
│   │   ├── 4_Code_Summary.py
│   ├── components/           # Reusable UI widgets
│   │   ├── chatbot.py        # Floating chatbot assistant
│   │   ├── file_uploader.py  # File upload UI
│   ├── assets/               # Images, CSS, logos
│   │   ├── logo.png
│   │   ├── styles.css

│── models/                   # AI/ML integration layer
│   ├── watsonx_model.py      # Granite-20B integration logic
│   ├── langchain_pipeline.py # LangChain pipelines
│   ├── vector_store.py       # Pinecone vector search setup
│   ├── ml_models.py          # ML model definitions

│── docs/                     # Documentation
│   ├── architecture.md       # System architecture diagrams
│   ├── setup_instructions.md # Setup guide
│   ├── api_endpoints.md      # API documentation

│── data/                     # Sample data & inputs
```

```
│   ├── sample_reqs.pdf
│   ├── sample_code.py

│── requirements.txt          # Python dependencies
│── README.md                 # Project overview
│── .env                      # Environment variables (API keys, etc.)
│── .gitignore                # Ignore venv, cache, etc.
```

6.Running the Application

1. Activate Virtual Environment

Make sure your virtual environment is active:

```
# Windows
venv\Scripts\activate

# Mac/Linux
source venv/bin/activate
```

2. Start the Backend (FastAPI + Uvicorn)

Go into the backend folder:

```
cd backend
```

Run the FastAPI server with Uvicorn:

```
uvicorn main:app --reload
```

- The backend will now be running at:
  - http://127.0.0.1:8000

- API docs are automatically available at:
  - **http://127.0.0.1:8000/docs**

3. Start the Frontend (Streamlit UI)

Open another terminal, activate the venv again, and run:

```
cd frontend
streamlit run app.py
```

4. Environment Variables (Important)

Before running, make sure you have a .env file in the root directory (smartsdlc/.env) with your API keys and configs:

```
WATSONX_API_KEY=your_ibm_watsonx_api_key
WATSONX_PROJECT_ID=your_ibm_project_id
PINECONE_API_KEY=your_pinecone_api_key
```

.

.

7.API documentation

Overview

This API powers the SmartSDLC platform, enabling automation across the Software Development Lifecycle (SDLC). It provides endpoints for requirement analysis, code generation, test case creation, bug fixing, and documentation generation.

The backend is implemented with **FastAPI**, which automatically generates interactive documentation via **Swagger UI** and **ReDoc**.

Base URL

http://localhost:8000/api/v1

Authentication

- Currently, endpoints are open for testing.
- Future versions will support **API Key / JWT authentication**.

Endpoints

1. Health Check

GET /health

- **Description**: Verifies if the API is running.
- **Response**:

```
{
  "status": "ok",
  "message": "API is up and running"
}
```

2. Upload Requirements (PDF/Text)

POST /requirements/upload

- **Description**: Uploads requirement documents for AI-based processing.
- **Request Body (multipart/form-data)**:
  - file: PDF or text file
- **Response**:

```
{
  "message": "File uploaded successfully",
  "file_id": "req_12345"
}
```

3. Generate Requirements

POST /requirements/generate

- **Description**: Extracts structured requirements using **IBM Watsonx + Granite Models**.
- **Request Body**:

```
{
  "file_id": "req_12345"
}
```

- **Response**:

```
{
  "requirements": [
    {"id": 1, "text": "The system shall allow user login"},
    {"id": 2, "text": "The system shall generate reports"}
  ]
}
```

4. Code Generation

POST /code/generate

- **Description**: Generates source code in selected language.
- **Request Body**:

```
{
  "requirements": ["User login", "Report generation"],
  "language": "python"
}
```

- **Response**:

```
{
  "code": "def login():\n    # user login code here"
}
```

5. Test Case Generation

POST /testcases/generate

- **Description**: Creates automated test cases from requirements/code.
- **Response**:

```
{
  "testcases": [
    {"id": "TC1", "description": "Verify user login"},
    {"id": "TC2", "description": "Verify report generation"}
  ]
}
```

6. Bug Fixing

POST /code/fix

- **Description**: Fixes bugs in submitted code.
- **Request Body**:

```
{
  "code": "print('Hello World'"
}
```

- **Response**:

```
{
  "fixed_code": "print('Hello World')",
  "notes": "Added missing closing parenthesis"
}
```

7. Documentation Generation

POST /docs/generate

- **Description**: Generates technical documentation.
- **Response**:

```
{
  "documentation": "This API handles requirement analysis, code generation, and test case creation..."
}
```

8.Authentication

Authentication in SmartSDLC

Authentication in **SmartSDLC** is the process of verifying the identity of users before they can access the system's features such as requirement extraction, code generation, bug fixing, and documentation.

SmartSDLC uses **JWT (JSON Web Token) based authentication** to ensure security.

How it Works

1. **User Registration** – A new user creates an account with a username and password.
2. **Login** – The user logs in with their credentials. If correct, the system issues a **JWT token**.
3. **Token Usage** – For every API request, the user must include this token in the **Authorization header**.

4. **Token Expiry** – Tokens are valid for a limited time (e.g., 1 hour). Once expired, the user must log in again or use a refresh token.
5. **Protected Endpoints** – Only users with valid tokens can access sensitive features like uploading requirements, generating code, or fixing bugs.

Why JWT?

Stateless**: No need to store sessions on the server.**

- **Lightweight**: Tokens are small and easy to pass in API requests.
- **Secure**: Each token is signed, so it cannot be tampered with.

Example Flow

- A developer signs in to SmartSDLC.
- The system issues a JWT token.
- When the developer calls the **"Generate Code"** API, the request includes the token.
- The backend verifies the token → if valid, it returns the generated code; if not, it denies access.

This way, **SmartSDLC ensures that only authorized users can access its AI-powered SDLC automation features**.

9.User Interface

1. **Dashboard**
   - Overview of the project status (requirements, generated code, test cases, bug reports, documentation).
   - Quick access to recent activities.

- AI insights and recommendations.

2. **Project Upload & Management**
    - **File Upload Panel**: Users can upload project files (e.g., PDF requirements, source code).
    - **Project List View**: Displays all active projects with progress tracking.
    - **Version Control Integration**: Shows versions or iterations of code and documentation.
3. **Requirement Analysis Interface**
    - Text box or PDF upload option for requirement documents.
    - AI-generated structured requirements displayed in a table or card view.
    - Option to edit/refine AI-suggested requirements manually.
4. **Code Generation Workspace**
    - Input fields for requirement selection.
    - Generated code preview with syntax highlighting.
    - Options to copy, download, or directly push code to a repo.
5. **Test Case Generator**
    - Auto-generated test cases displayed in table format.
    - Manual addition and editing allowed.
    - Export to testing frameworks (e.g., PyTest, JUnit).
6. **Bug Fixing & Debugging Panel**
    - Upload buggy code snippets or paste directly.
    - AI suggests bug fixes with explanations.
    - Side-by-side **before & after** code comparison.
7. **Documentation Assistant**
    - Automatically generated documentation (SRS, design docs, user manuals).
    - Options to export in multiple formats: PDF, DOCX, Markdown.

    - Inline editing for customization.

8. **Collaboration Tools**
    - Commenting system for team discussions on requirements/code.
    - AI chatbot assistant for clarifications.
    - Task assignments for team members.
9. **Integration with Streamlit (Frontend)**
    - Clean, card-based UI with collapsible sections.
    - Interactive widgets for file uploads, text inputs, and drop-downs.
    - Real-time rendering of AI responses.

Design Style

- **Frontend**: Built with **Streamlit** (minimal, interactive, fast prototypes).
- **UI Layout**: Card-based, grid sections, side navigation bar.
- **Styling**: Modern, lightweight, easy to navigate.
- **Accessibility**: AI suggestions are presented in editable fields so users stay in control.

10.Testing

1. AI-Powered Test Case Generation

- From **requirements** or **code snippets**, SmartSDLC auto-generates:
    - **Unit Tests** → function-level validation (e.g., Python → pytest, Java → JUnit).
    - **Integration Tests** → ensures modules interact correctly.
    - **System Tests** → end-to-end functionality check.

   - **Edge/Negative Tests** → AI predicts possible error scenarios.

 Example: If a requirement says *"The system must validate user login"*, SmartSDLC generates test cases like:

- Valid credentials → successful login.
- Invalid password → error message.
- SQL injection attempt → blocked.

2. Automated Test Execution

- Users can run generated test cases directly in the platform.
- Test reports show:
   - ✅Passed cases
   - ❌Failed cases (with explanation)
   - Suggested improvements

SmartSDLC can connect with **CI/CD tools** (GitHub Actions, Jenkins, GitLab CI) for continuous testing.

3. Bug Detection & Fix Suggestions

- If a test fails, SmartSDLC:
   - Highlights the buggy code section.
   - Suggests AI-powered fixes.
   - Provides reasoning for the fix.
   - Allows side-by-side comparison (before vs. after).

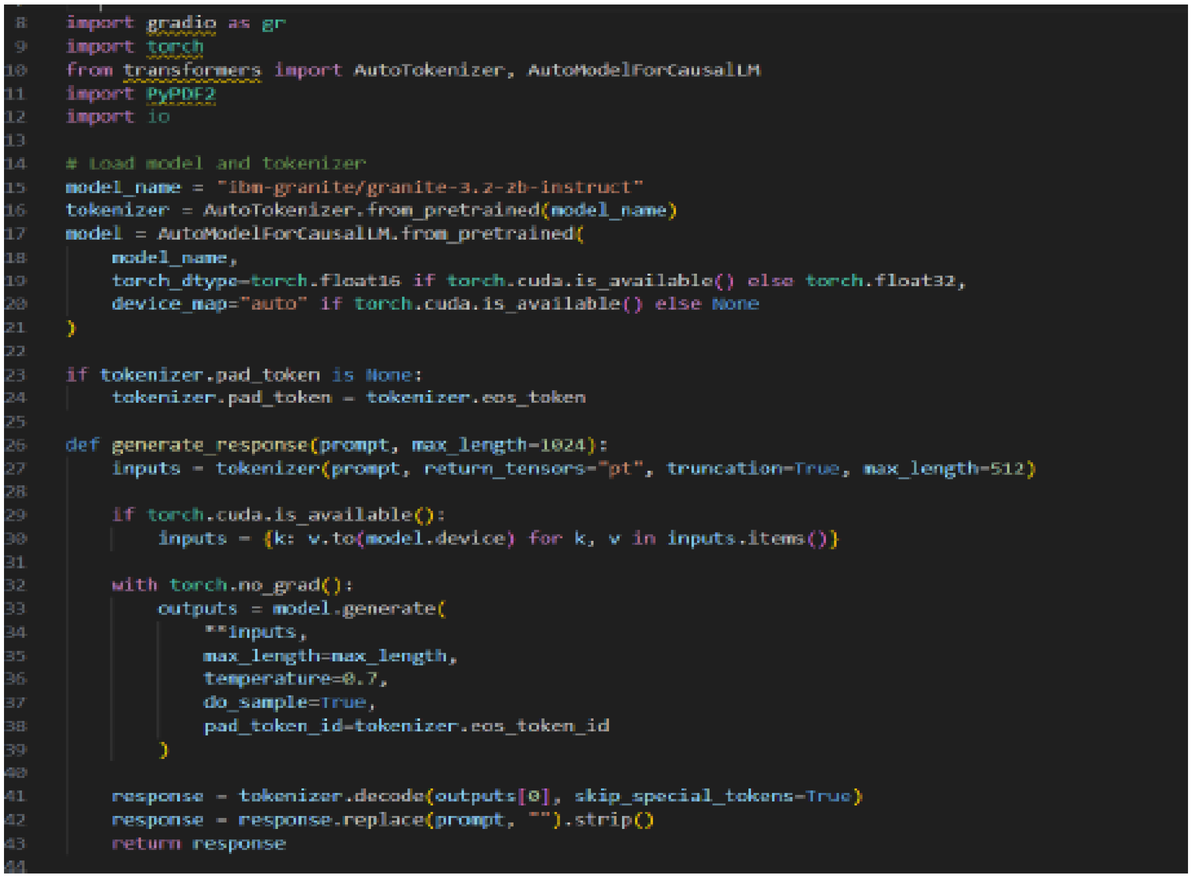
4. Test Documentation

- Automatically generates:
   - **Test Case Reports** (in tabular format).
   - **Traceability Matrix** (requirements → test cases).
   - Export in **PDF, DOCX, or Markdown**.

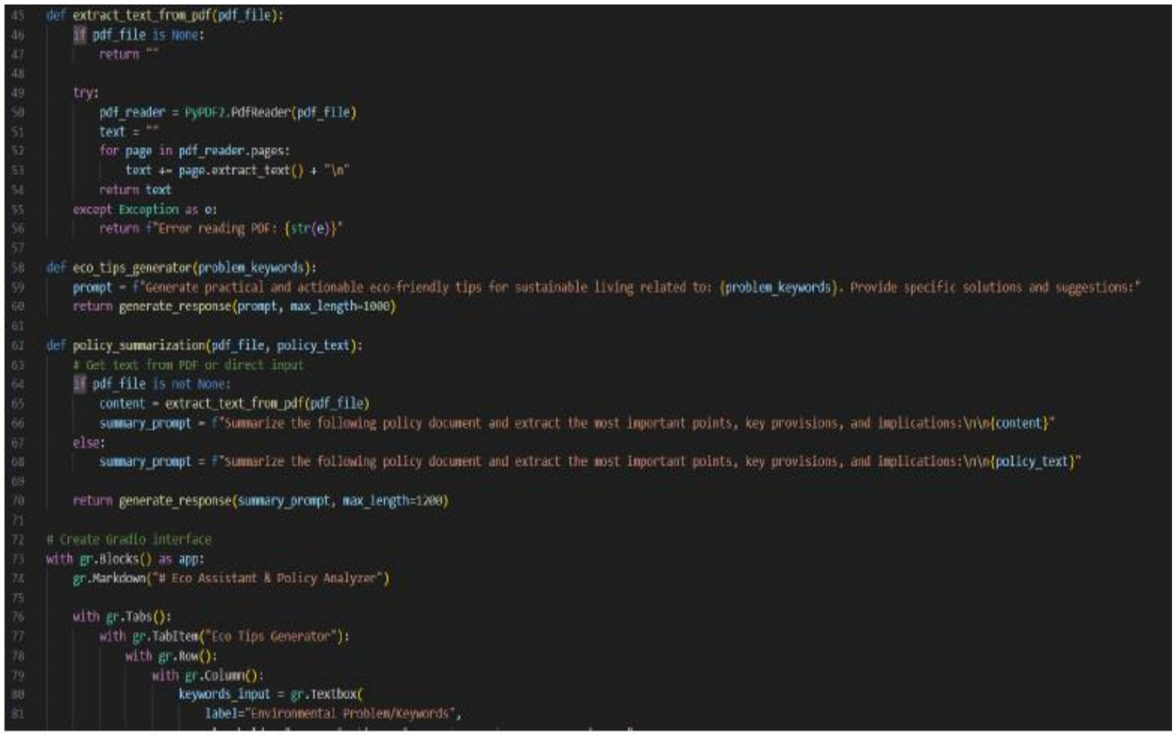
5. UI for Testing in SmartSDLC

- **Test Case Generator Panel**: Upload requirement/code → get AI-generated tests.
- **Execution Window**: Run all/selected test cases with live output.
- **Results Dashboard**: Pass/fail summary, coverage, suggestions.
- **Bug Fixing Assistant**: AI recommends corrections.

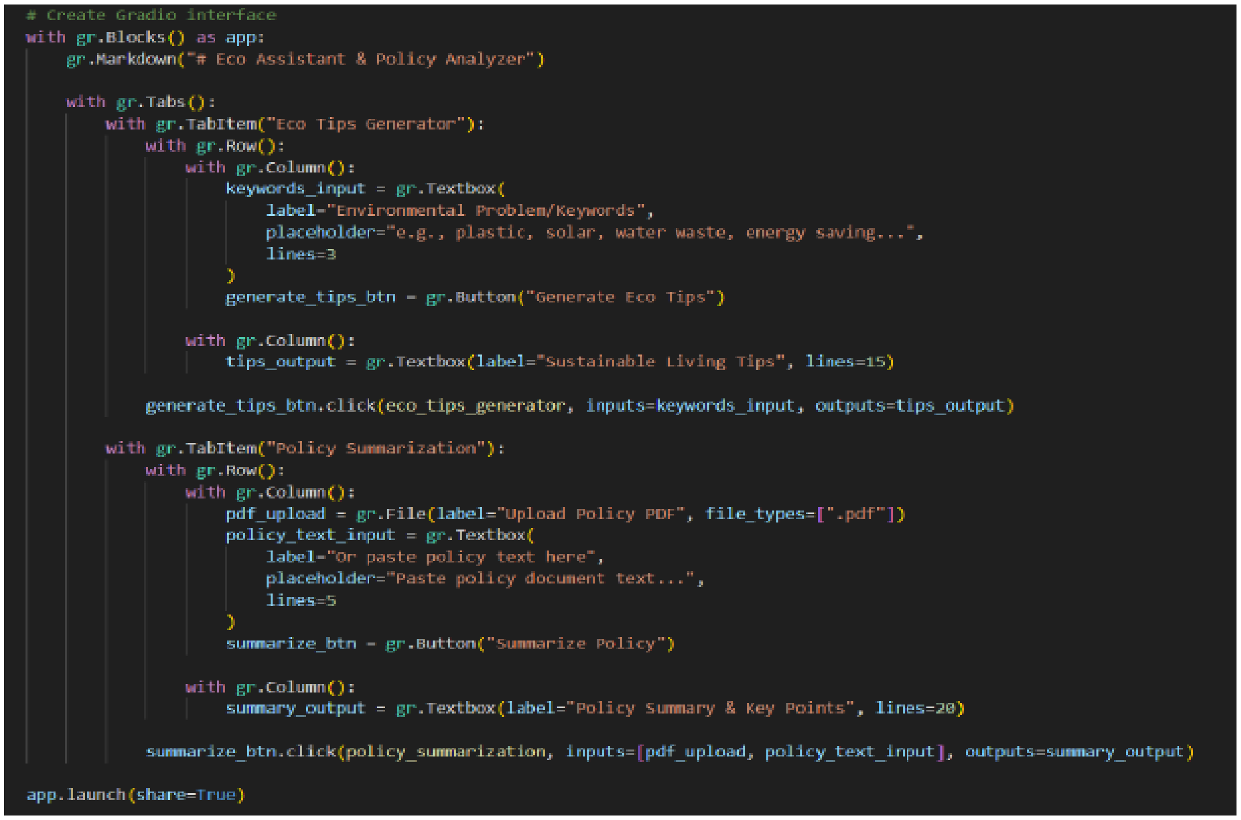
11. Screenshot



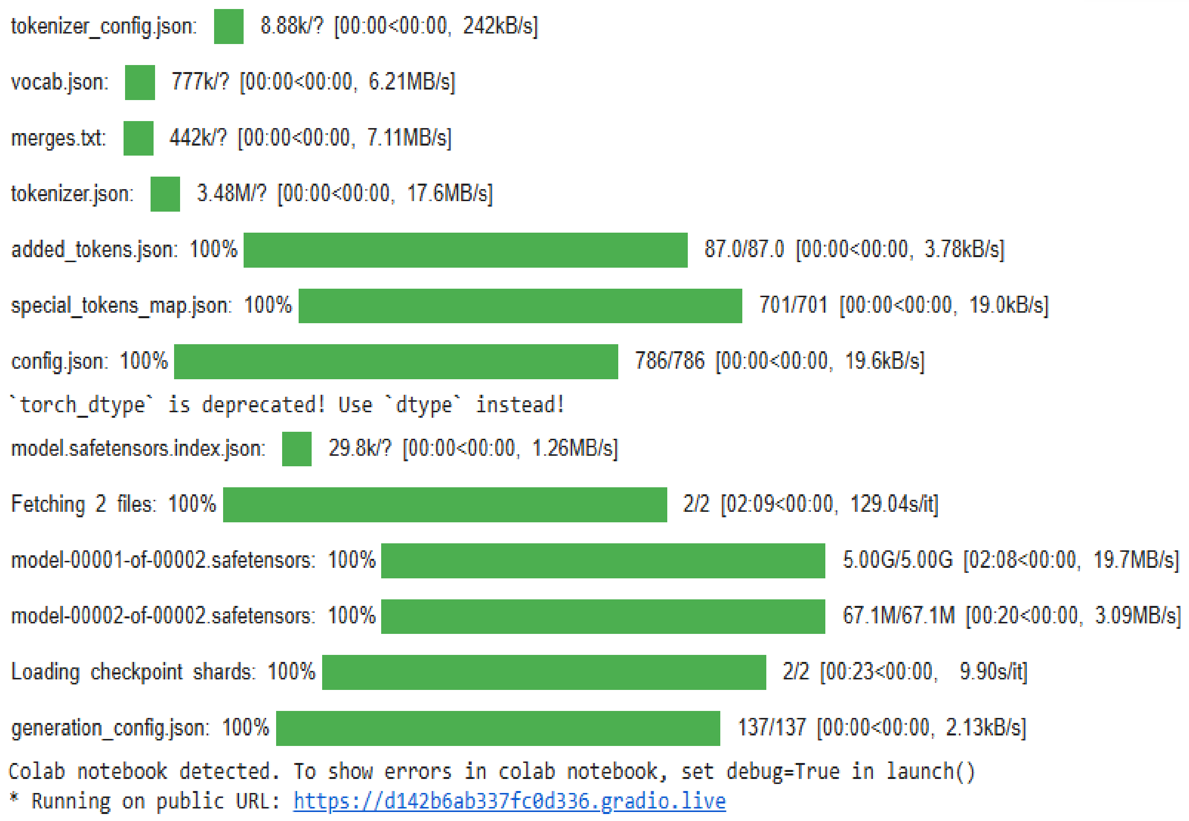
```python
import gradio as gr
import torch
from transformers import AutoTokenizer, AutoModelForCausalLM
import PyPDF2
import io

# 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
```



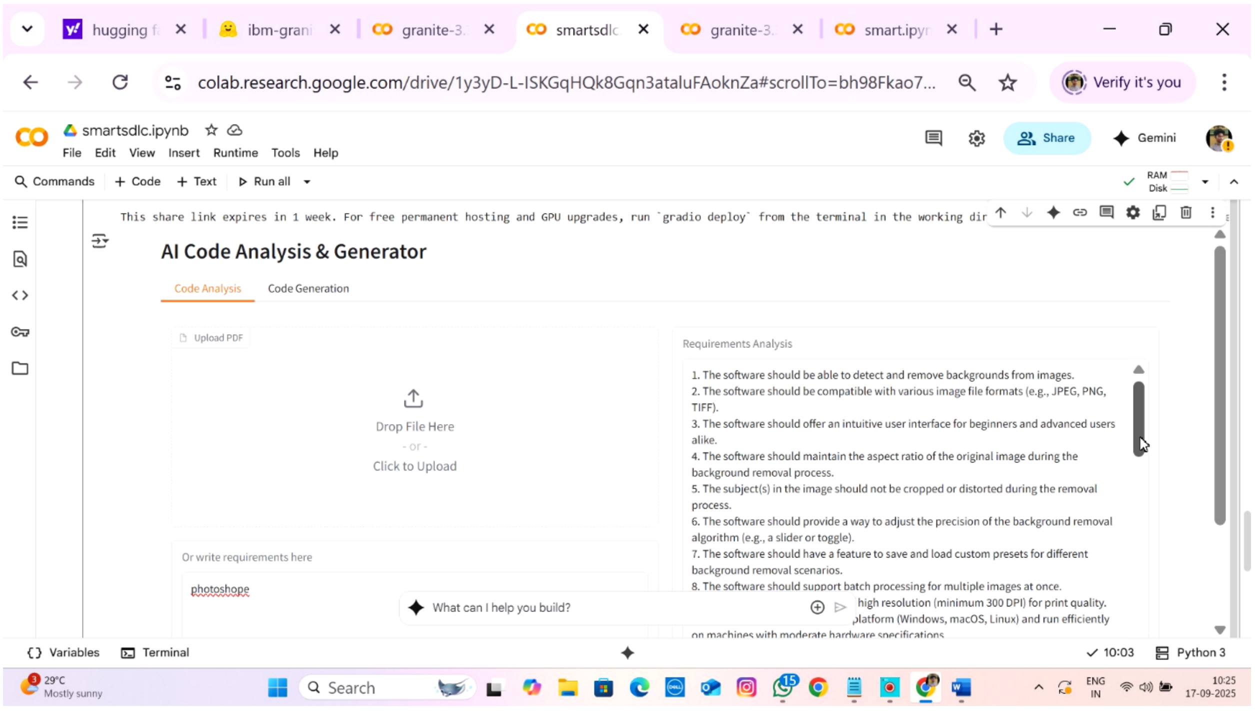
```python
def extract_text_from_pdf(pdf_file):
    if pdf_file is None:
        return ""

    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 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)

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():
        with gr.TabItem("Eco Tips Generator"):
            with gr.Row():
                with gr.Column():
                    keywords_input = gr.Textbox(
                        label="Environmental Problem/Keywords",
```



```python
# Create Gradio interface
with gr.Blocks() as app:
    gr.Markdown("# Eco Assistant & Policy Analyzer")

    with gr.Tabs():
        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)

        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)
```



```
tokenizer_config.json:  8.88k/?  [00:00<00:00,  242kB/s]
vocab.json:  777k/?  [00:00<00:00,  6.21MB/s]
merges.txt:  442k/?  [00:00<00:00,  7.11MB/s]
tokenizer.json:  3.48M/?  [00:00<00:00,  17.6MB/s]
added_tokens.json: 100%  87.0/87.0  [00:00<00:00,  3.78kB/s]
special_tokens_map.json: 100%  701/701  [00:00<00:00,  19.0kB/s]
config.json: 100%  786/786  [00:00<00:00,  19.6kB/s]
`torch_dtype` is deprecated! Use `dtype` instead!
model.safetensors.index.json:  29.8k/?  [00:00<00:00,  1.26MB/s]
Fetching 2 files: 100%  2/2  [02:09<00:00, 129.04s/it]
model-00001-of-00002.safetensors: 100%  5.00G/5.00G  [02:08<00:00,  19.7MB/s]
model-00002-of-00002.safetensors: 100%  67.1M/67.1M  [00:20<00:00,  3.09MB/s]
Loading checkpoint shards: 100%  2/2  [00:23<00:00,   9.90s/it]
generation_config.json: 100%  137/137  [00:00<00:00,  2.13kB/s]
Colab notebook detected. To show errors in colab notebook, set debug=True in launch()
* Running on public URL: https://d142b6ab337fc0d336.gradio.live
```


smartsdlc.ipynb
File Edit View Insert Runtime Tools Help
Commands + Code + Text Run all

```
This share link expires in 1 week. For free permanent hosting and GPU upgrades, run `gradio deploy` from the terminal in the working dir
```

AI Code Analysis & Generator

Or write requirements here

photoshope

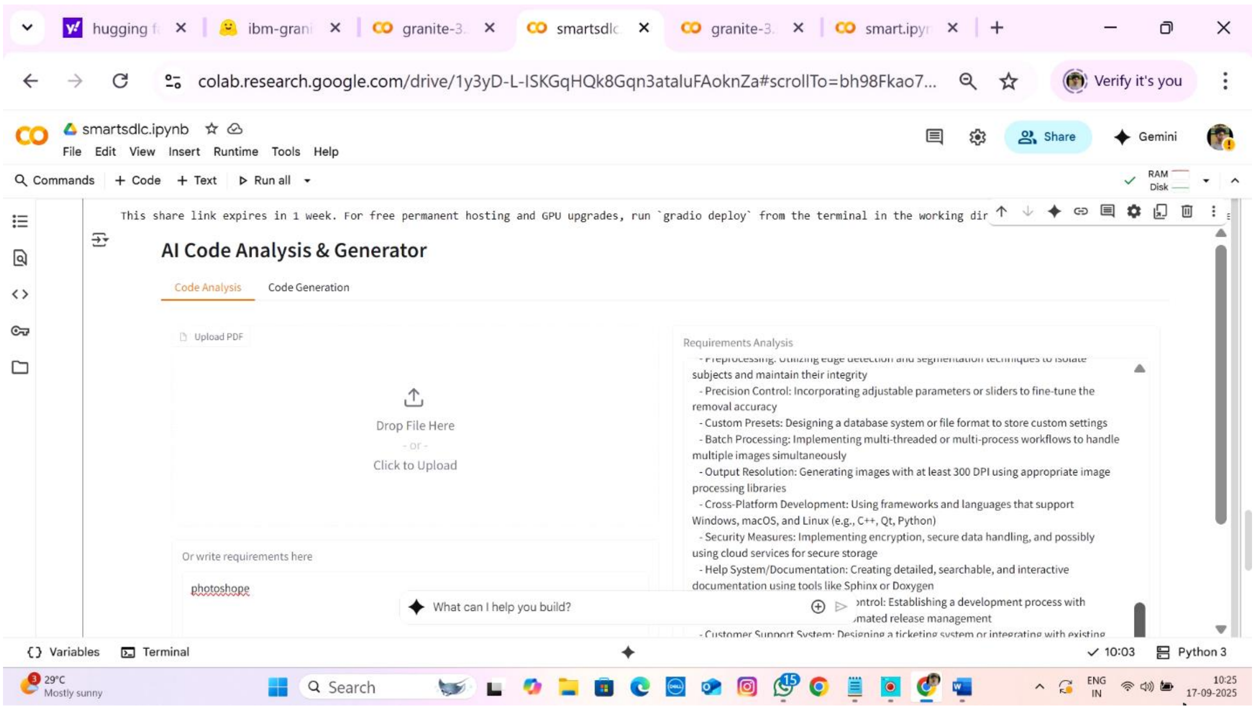
Requirements Analysis

1. The software should be able to detect and remove backgrounds from images.
2. The software should be compatible with various image file formats (e.g., JPEG, PNG, TIFF).
3. The software should offer an intuitive user interface for beginners and advanced users alike.
4. The software should maintain the aspect ratio of the original image during the background removal process.
5. The subject(s) in the image should not be cropped or distorted during the removal process.
6. The software should provide a way to adjust the precision of the background removal algorithm (e.g., a slider or toggle).
7. The software should have a feature to save and load custom presets for different background removal scenarios.
8. The software should support batch processing for multiple images at once.

high resolution (minimum 300 DPI) for print quality.

platform (Windows, macOS, Linux) and run efficiently on machines with moderate hardware specifications

12. Known Issues


smartsdlc.ipynb
File Edit View Insert Runtime Tools Help
Commands + Code + Text Run all

```
This share link expires in 1 week. For free permanent hosting and GPU upgrades, run `gradio deploy` from the terminal in the working dir
```

AI Code Analysis & Generator

Or write requirements here

photoshope

Requirements Analysis

- Preprocessing: Utilizing edge detection and segmentation techniques to isolate subjects and maintain their integrity
- Precision Control: Incorporating adjustable parameters or sliders to fine-tune the removal accuracy
- Custom Presets: Designing a database system or file format to store custom settings
- Batch Processing: Implementing multi-threaded or multi-process workflows to handle multiple images simultaneously
- Output Resolution: Generating images with at least 300 DPI using appropriate image processing libraries
- Cross-Platform Development: Using frameworks and languages that support Windows, macOS, and Linux (e.g., C++, Qt, Python)
- Security Measures: Implementing encryption, secure data handling, and possibly using cloud services for secure storage
- Help System/Documentation: Creating detailed, searchable, and interactive documentation using tools like Sphinx or Doxygen

ntrol: Establishing a development process with mated release management

- Customer Support System: Designing a ticketing system or integrating with existing

1. **Limited Accuracy in Requirement Extraction**
    - The system may not always extract requirements correctly from unstructured or poorly formatted PDF documents.
    - Complex sentences or domain-specific terminology can lead to incomplete or incorrect requirement generation.
2. **Dependency on External APIs**
    - Integration with IBM Watsonx AI and Granite models relies on stable internet and external API availability.
    - Any downtime, latency, or API quota limitations can affect the platform's functionality.
3. **Scalability Challenges**
    - The application may experience performance slowdowns when processing very large documents or handling multiple concurrent users.
    - Current deployment is not fully optimized for high-scale enterprise usage.
4. **Authentication and Security Limitations**
    - User authentication is basic (limited to standard methods).
    - No role-based access control (RBAC) or advanced encryption measures have been fully implemented yet.
5. **UI/UX Constraints**
    - The user interface (Streamlit-based) is functional but lacks advanced design flexibility.
    - Mobile responsiveness and accessibility features are limited.
6. **Partial Automation in SDLC Phases**
    - While requirement analysis, test case generation, and code suggestions are supported, phases like deployment, monitoring, and continuous integration are not fully automated.
7. **Error Handling & Logging**
    - Error messages may be generic, making it harder for non-technical users to understand issues.
    - Logging is minimal and requires improvement for debugging and monitoring.
8. **Testing Coverage**
    - Automated test case generation exists but is not comprehensive for all scenarios.
    - Generated test cases may require manual validation and refinement.

13. Future enhancement

1. **Advanced Requirement Extraction**
    - Improve NLP models to handle domain-specific documents and extract requirements with higher accuracy.
    - Support multiple document formats (Word, Excel, HTML) in addition to PDF.
2. **Enhanced Authentication & Security**
    - Implement Role-Based Access Control (RBAC) for different user levels (Admin, Developer, Tester, Client).
    - Add multi-factor authentication (MFA) and stronger data encryption.
3. **Scalability & Cloud Deployment**
    - Deploy on scalable cloud platforms (AWS, Azure, GCP) with auto-scaling and load balancing.
    - Containerization (Docker/Kubernetes) for better portability and performance.
4. **Improved User Interface (UI/UX)**
    - Redesign UI with modern frameworks (React, Angular) for better flexibility.
    - Ensure full mobile responsiveness and accessibility (WCAG standards).
5. **End-to-End SDLC Automation**
    - Extend automation to later phases like deployment, CI/CD, monitoring, and feedback integration.
    - Integration with tools like Jenkins, GitHub Actions, or GitLab CI.
6. **AI-Powered Code Review & Bug Fixing**
    - Enhance AI models to suggest optimized code snippets and automatically fix common coding issues.
    - Provide code quality scores and recommendations.
7. **Comprehensive Test Automation**
    - Expand auto-generated test cases to include integration testing, performance testing, and security testing.
    - Provide dashboards for real-time test coverage and defect tracking.
8. **Collaboration & Project Management Features**
    - Add features like task assignments, project timelines, and team collaboration boards.
    - Integration with tools like Jira, Trello, or Slack.
9. **Multi-Language & Multi-Domain Support**
    - Extend support for global languages in requirements and documentation.
    - Customize AI models for industries like healthcare, finance, and e-commerce.
10. **Advanced Analytics & Reporting**

- Provide insights into project progress, risks, and resource utilization.
- AI-driven predictions for project delays, cost overruns, and quality issues.

14 Conclusion

The SmartSDLC project demonstrates how Artificial Intelligence can be effectively integrated into the Software Development Lifecycle to improve efficiency, accuracy, and automation. By leveraging advanced technologies such as IBM Watsonx, Granite models, FastAPI, and Streamlit, the platform simplifies critical phases like requirement analysis, code generation, test case creation, and documentation.

Although the system currently faces challenges such as limited scalability, partial automation, and basic authentication mechanisms, it has established a strong foundation for AI-driven SDLC management. With planned enhancements in security, UI/UX, scalability, and end-to-end automation, SmartSDLC has the potential to evolve into a comprehensive and intelligent development assistant.

In summary, SmartSDLC is a step toward **next-generation software engineering**, where AI not only assists but actively drives the development process, reducing manual effort, improving quality, and accelerating project delivery.